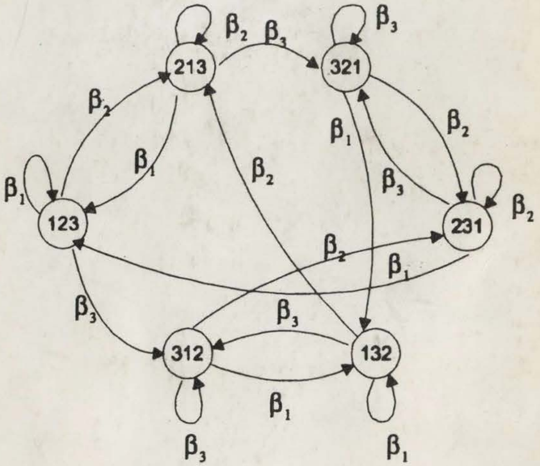
IOAN
TOMESCU

DATA STRUCTURES



EDITURA UNIVERSITĂȚII DIN BUCUREȘTI

IOAN TOMESCU

DATA STRUCTURES

EDITURA UNIVERSITĂȚII DIN BUCUREȘTI
– 1997 –

Referenți științifici: Conf. dr. **Octavian BÂSCĂ**
Conf. dr. **Horia Ioan GEORGESCU**

Redactor : **Ion MIHAI**
Tehnoredactare computerizată: **Constanța TITU**

ISBN: 973 – 575 – 147 – X

CONTENTS

Chapter 1. Structured data types

§ 1. Linear lists. Stacks and Queues

A linear list is a set of $n \geq 0$ nodes $X[1]$, $X[2]$,..., $X[n]$ whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes:

if $n > 0$, $X[1]$ is the first node;

when $1 < k < n$, the k- th node $X[k]$ is preceded by $X[k-1]$ and followed by X [k+1];

$X[n]$ is the last node.

The operations we might want to perform on linear lists include, for example, the following:

i) Gain acces to the k–th node of the list to examine and/or change the contents of its fields.

ii) Insert a new node just before the k– th node.

iii) Delete the k– th node.

iv) Combine two or more lists into a single list.

v) Split a linear list into two or more lists.

vi) Make a copy of a linear list.

vii) Determine the number of nodes in a list.

viii) Sort the nodes of the list into ascending order based on certain fields of the nodes.

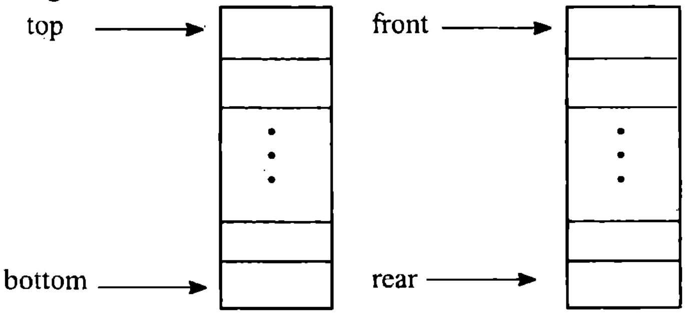
ix) Search the list for the occurrence of a node with a particular value in some field.

There are many ways to represent linear lists depending on the class of operations which are to be done most frequently. It appears to be impossible to design a simple representation method for linear lists in which all of these operations are efficient.

Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are frequently encountered, and we give them special names:

- A stack is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list. Stacks have been called push - down lists, last-in-first out (LIFO) lists,...
- A queue is a linear list for which all insertions are made at one end of the list; all deletions are made at the other end. Queues have been called also circular stores or first-in-first-out (FIFO) lists.

Stacks most frequently occur in connection with recursive algorithms.



Stack

Queue

We write $A \Leftarrow x$ (when A is a stack) to mean that the value x is inserted on top of stack A, or (when A is a queue) to mean that x is inserted at the rear of the queue.

The notation $x \Leftarrow A$ is used to mean that the variable x is set equal to the value at the top of stack A or at the front of queue A, and this value is deleted from A.

Notation $x \Leftarrow A$ is meaningless when A is empty.

§ 2 Sequential allocation

The simplest way to keep a linear list inside a computer is to put the list items in sequential locations, one node after the other:

$$\mathrm{LOC}(X[j+1]) = \mathrm{LOC}(X[j]) + c$$

where c is the number of words per node (usually $c = 1$).

We will assume that adjacent groups of c words form a single node.

In general LOC($X[j]$) = $L_0 + cj$, where L_0 is a constant, called the base address.

Sequential allocation is convenient for dealing with a stack.

We have a variable T called the stack pointer. When the stack is empty, we let $T = 0$.

```
· $X \Leftarrow Y$ (insert into stack):
        $T \leftarrow T + 1$; if $T > M$ then OVERFLOW;
            $X[T] \leftarrow Y$.
· $Y \Leftarrow X$ (delete from stack):
        if $T = 0$, then UNDERFLOW;
        $Y \leftarrow X[T]$; $T \leftarrow T - 1$ .
```

(We assumed that $X[1]$,..., $X[M]$ is the total amount of space allowed for the list).

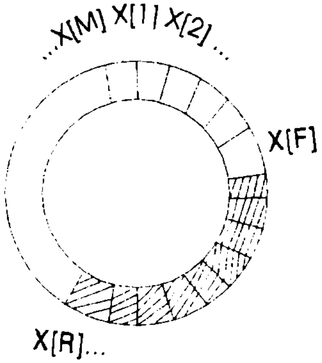
OVERFLOW means that the storage capacity has heen exceeded, and the program terminates.

For a queue we use two pointers F and R (for the front and rear of the queue). If $F = R$ then the queue is empty.

We can set aside M nodes $X[1]$,..., $X[M]$ arranged implicitly in a circle with $X[1]$ following $X[M]$. ($R = F = M$ initially).

Hence we use a queue as a circular queue.

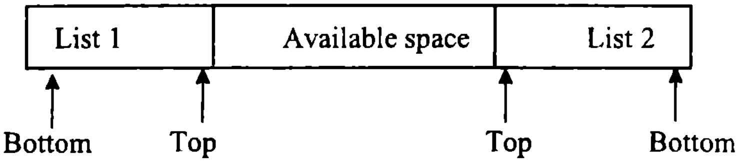
$$\text{Let } c = \begin{cases} 1, & \text{if queue is full} \\ 0, & \text{if queue is empty} \\ 2, & \text{otherwise} \end{cases}$$



```
• $X \Leftarrow Y$ (insert into queue):
1. if $c = 1$ then OVERFLOW;
```

```
    if $R = M$ then $R \leftarrow 1$, else $R \leftarrow R + 1$.
2. $X[R] \leftarrow Y$; if $R = F$ then $c \leftarrow 1$, else $c \leftarrow 2$.
```

```
• $Y \Leftarrow X$ (delete from queue):
1. if $c = 0$ then UNDERFLOW;
    if $F = M$ then $F \leftarrow 1$, else $F \leftarrow F + 1$.
2. $Y \leftarrow X[F]$; if $F = R$ then $c \leftarrow 0$, else $c \leftarrow 2$.
```

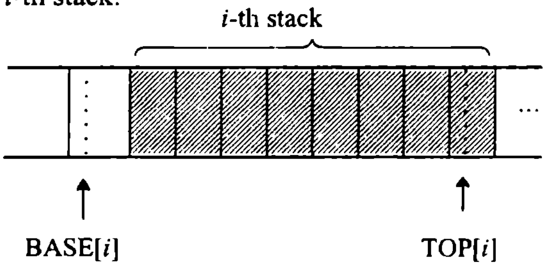


When there are just two variable size list, they can coexist together if we let the lists grow toward each other: List 1 expands to the right and List 2 to the left.

OVERFLOW will occur when the total size of both lists exhausts all memory space.

But there is no way to store three or more variable – size sequential lists in memory so that (a) OVERFLOW will occur only when the total size of all lists exceeds the total space, and (b) each list has a fixed location for its „bottom" element.

Suppose that we have n stacks; the insertion and deletion algorithms above become the following if BASE[i] and TOP[i] are link variables for the i-th stack:



```
• Insertion: TOP[$i$] $\leftarrow$ TOP [$i$] + $c$;
    if TOP [$i$] > BASE [$i$+1], then OVERFLOW; otherwise set
NODE [TOP[$i$]] $\leftarrow Y$.
```

• Deletion: if TOP $[i]$ = BASE $[i]$, then UNDERFLOW; otherwise set $Y \leftarrow$ NODE [TOP $[i]$], TOP $[i] \leftarrow$ TOP $[i] - c$.

These stacks are all to share the common memory area consisting of all locations L with $L_0 \leq L < L_\infty$

($L_\infty - L_0$ is a multiple of c).

We wight start out with all stacks empty, and BASE $[i]$ = TOP $[i]$ $= L_0 - c$, for all i.

We also set BASE $[n + 1] = L_\infty - c$.

Whenever a particular stack, except stack n, gets more items in it than it ever had before, OVERFLOW will occur.

When stack i overflows, there are three possibilities:

a) We find the smallest k for which

$i < k \geq n$ and TOP $[k]$ < BASE $[k + 1]$,

if any such k exist.

Set CONTENTS $(L + c) \leftarrow$ CONTENTS (L) for TOP $[k] + c > L \geq$ BASE $[i + 1] + c$ (this should be done for decreasing values of L to avoid losing information).

Set BASE $[j] \leftarrow$ BASE $[j] + c$

TOP $[j] \leftarrow$ TOP $[j] + c$, for $i < j \leq k$.

b) No k can be found as in a), but we find the largest k for which $1 \leq k < j$ and TOP $[k]$ < BASE $[k + 1]$.

Set CONTENTS $(L - c) \leftarrow$ CONTENTS (L),

for BASE $[k + 1] + c \leq L <$ TOP $[i] + c$

(for increasing values of L)

Set BASE $[j] \leftarrow$ BASE $[j] - c$, TOP $[j] \leftarrow$ TOP $[j] - c$, for $k < j \leq i$.

c) We have TOP $[k]$ = BASE $[k + 1]$ for all $k \neq i$. We cannot find space for the new stack entry, and we must give up.

By imagining a sequence of m insertion operations $a_1, a_2, \ldots, a_m$ where each a_i is an integer between 1 and n representing an insertion on top of stack a_i, we can regard each of the n^m possible specifications $a_1, \ldots, a_m$ as equally likely.

We can ask for the average number of times it is necessary to move a word from one location to another during the repacking operations as the entire table is built.

Starting with all available space given to the n–th stack, we find that the average number of move operations required is

$$\frac{1}{2}\left(1-\frac{1}{n}\right)\binom{m}{2}$$

(hence is essentially proportional to the square of the number of items in the tables) by counting the number of inversions in all n^m such strings.

• Algorithm *R* [Relocate sequential tables].

For $1 \le j \le n$ the information specified by BASE $[j]$ and TOP$[j]$ in accord with the given conventions is moved to new positions specified by NEWBASE $[j]$ and the values of BASE $[j]$ and TOP $[j]$ are suitably adjusted.

1 (Initialize) Set $j \leftarrow 1$ (Note that stack 1 never needs to be moved, so for efficiency the programmer should put the longest stack first if he knows which one will be largest).

2 (Find start of shift). Increase j in steps of 1 until finding either a) NEWBASE $[j] <$ BASE $[j]$: go to 3; or b) NEWBASE $[j] >$ BASE $[j]$: go to 4; or c) $j > n$: the algorithm terminates.

3 (Shift down) Set $\delta \leftarrow$ BASE $[j]$ – NEWBASE $[j]$. Set CONTENTS $(L - \delta) \leftarrow$ CONTENTS (L) for $L =$ BASE $[j] + c$, BASE $[j] + c + 1$,..., TOP $[j] + c - 1$. Set BASE $[j] \leftarrow$ NEWBASE $[j]$, TOP $[j] \leftarrow$ TOP $[j] - \delta$.

Go to 2.

4 (Find top of shift). Find the smallest $k \ge j$ for which NEWBASE $[k + 1] \le$ BASE $[k + 1]$. (Note that NEWBASE $[n+1]$ = BASE $[n + 1]$, so that such a k will always exist).

Then do step 5 for $t = k, k - 1$,..., j; finally set $j \leftarrow k$ and go to 2.

5 (shift up) Set

$\delta \leftarrow$ NEWBASE $[t]$ – BASE $[t]$

Set

CONTENTS $(L + \delta) \leftarrow$ CONTENTS (L) for $L =$ TOP $[t] +$ $+ c - 1$, TOP $[t] + c - 2$,..., BASE $[t] + c$.

Set BASE $[t] \leftarrow$ NEWBASE $[t]$

TOP $[t] \leftarrow$ TOP $[t] + \delta$.

Notation. Let $f, g : \mathbf{N} \to \mathbf{R}$

• We write $f(n) = O(g(n))$ if there exist $C > O$ and $n_0 \in \mathbf{N}$ such that

$$\left|\frac{f(n)}{g(n)}\right| < C$$

for all $n > n_0$

- $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

- $f(n) \sim g(n)$ (asimptotically equal) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

§ 3. Linked Allocation

Each node contains a link to the next node of the list:

Sequential allocation:

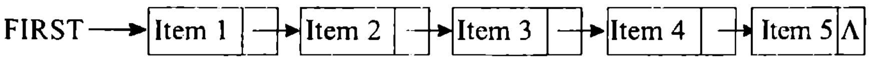
Address	Contents
$L_o + c$:	Item 1
$L_o + 2c$:	Item 2
$L_o + 3c$:	Item 3
$L_o + 4c$:	Item 4
$L_o + 5c$:	Item 5

Linked allocation:

Address	Contents	
A:	Item 1	B
B:	Item 2	C
C:	Item 3	D
D:	Item 4	E
E:	Item 5	Λ

Λ is the null link.

Links are often shown simply by arrows and the liked table above might be shown as follows:



Here FIRST is a link variable pointing to the first node of the list.
Comparisons between sequential and linked allocation:

- linked allocation takes up additional memory space for the links.
- it is easy to delete an item from within a linked list. For sequential allocation such a deletion implies moving a large part of the list.
- it is easy to insert an item into a list when the linked scheme is being used.

By comparison, this operation would be extremely time – consuming in a long sequential table.

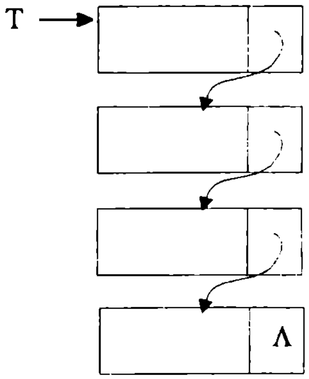
- references to random parts of the list are much faster in the sequential case.
- the linked scheme makes it easier to join two lists together or to break one apart.

We shall assume that a node has one word and that it is broken into two fields

INFO and LINK: | INFO | LINK |

The use of linked allocation implies the existence of a list of available space: AVAIL list (AVAIL stack).

„$X \Leftarrow$ AVAIL" : if AVAIL = Λ, then OVERFLOW;
otherwise $X \leftarrow$ AVAIL, AVAIL $\leftarrow$ LINK (AVAIL).
„AVAIL $\Leftarrow X$" : LINK (X) $\leftarrow$ AVAIL, AVAIL $\leftarrow X$.



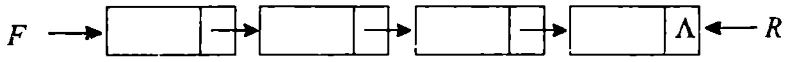
A linked stack

Insertion:

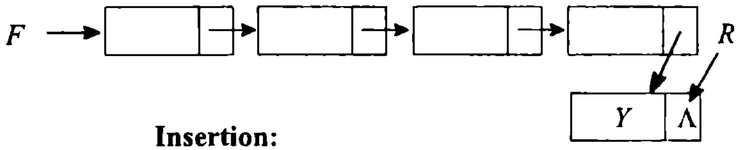
$P \Leftarrow$ AVAIL, INFO (P) $\leftarrow Y$,
LINK(P) $\leftarrow T$, $T \leftarrow P$.

Deletion:

If $T = \Lambda$, then UNDERFLOW;
otherwise set $P \leftarrow T$, $T \leftarrow$ LINK(P),
$Y \leftarrow$ INFO(P), AVAIL $\Leftarrow P$.



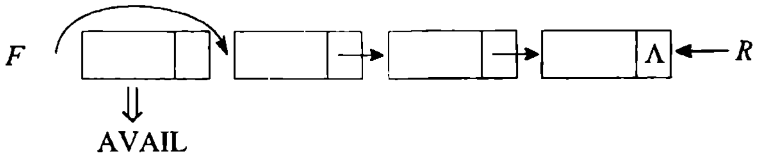
A linked queue



Insertion:

$P \Leftarrow$ AVAIL, INFO(P) $\leftarrow Y$, LINK(P) $\leftarrow \Lambda$; if $F = \Lambda$ then $R \leftarrow F \leftarrow P$, else LINK ($R$) $\leftarrow P$, $R \leftarrow P$.

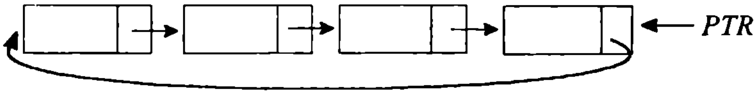
(By definition an empty queue is represented by $F = \Lambda$)



Deletion

If $F = \Lambda$, then UNDERFLOW; otherwise set $P \leftarrow F$, $F \leftarrow \text{LINK}(P)$, $Y \leftarrow \text{INFO}(P)$, AVAIL $\Leftarrow P$.

§ 4. Circular lists



A circular list has the property that its last node links back to the first instead to be Λ.

a) *Insert Y at left*

$P \Leftarrow$ AVAIL, $\text{INFO}(P) \leftarrow Y$, if $PTR = \Lambda$, then $PTR \leftarrow \text{LINK}(P) \leftarrow P$; otherwise $\text{LINK}(P) \leftarrow \text{LINK}(PTR)$, $\text{LINK}(PTR) \leftarrow P$.

b) *Insert Y at right*

Insert Y at left, then $PTR \leftarrow P$.

c) *Set T to left node and delete*

If $PTR = \Lambda$, then UNDERFLOW; otherwise $P \leftarrow \text{LINK}(PTR)$, $Y \leftarrow \text{INFO}(P)$, $\text{LINK}(PTR) \leftarrow \text{LINK}(P)$, if $PTR = P$, then $PTR \leftarrow \Lambda$, AVAIL $\Leftarrow P$.

$(a) + (c) =$ a stack

$(b) + (c) =$ a queue

It is convenient to „erase" a list, i.e., to put an entire circular list onto the AVAIL stack at once:

If $PTR \neq \Lambda$, then AVAIL $\leftrightarrow \text{LINK}(PTR)$.

(The ,,$\leftrightarrow$ '' operation denotes interchange, i.e., $P \leftarrow$ AVAIL, AVAIL $\leftarrow$ LINK(PTR), LINK (PTR) $\leftarrow P$).

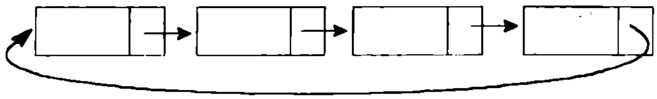
If PTR_1 and PTR_2 point to disjoint circular lists L_1 and L_2, respectively, we can insert the entire list L_2 at the right of L_1:

If $PTR_2 \neq \Lambda$ then
(if $PTR_1 \neq \Lambda$, then LINK (PTR_1) $\leftrightarrow$ LINK (PTR_2)
set PTR$_1 \leftarrow$ PTR$_2$, PTR$_2 \leftarrow \Lambda$).

We can put a special, recognizable node into each circular list, as a convenient stopping place.

This special node is called the list head.

List head



On this way the circular list will then never be empty. The references to such lists are usually made via the list head, which is often in a fixed memory location.

We will consider the two operations of addition and multiplication of polynomials in the variables x, y and z.

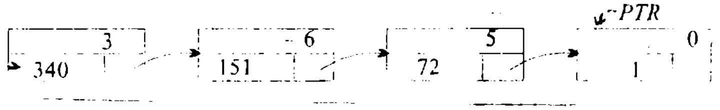
Let us suppose that a polynomial is represented as a list in which each node stands for one nonzero term, and has the two-word form

COEF	
± ABC	LINK

where COEF is the coefficient of the term $x^A y^B z^C$.

The nodes of the list always appear in decreasing order of the ABC field and the list head links to the largest value of ABC (the order is the lexicographic order).

For exemple, the polynomial $3x^3y^4 - 6xy^5z + 5y^7x^2$ would be represented thus:



Algorithm A (Addition of polynomials)

This algorithm adds polynomial P to polynomial Q, assuming that P and Q are pointer variables pointing to polynomials having the form above.

The list P will be unchanged, the list Q will retain the sum. Pointer variables P and Q return to their starting points at the conclusion of this algorithm; auxiliary pointer variables $Q1$ and $Q2$ are also used. The pointer variable $Q1$ follows the pointer Q around the list.

1. [Initialize]. Set $P \leftarrow$ LINK(P), $Q1 \leftarrow Q$, $Q \leftarrow$ LINK (Q) (Now both P and Q point to the leading term of the polynomial. The variable $Q1$ will be „one step behind" Q, in the sense that Q = LINK($Q1$)).

2. [$ABC(P) : ABC(Q)$]. If $ABC(P) < ABC(Q)$, set $Q1 \leftarrow Q$ and $Q \leftarrow$ LINK(Q) and repeat this step. If $ABC(P) = ABC(Q)$, go to step 3. If $ABC(P) > ABC(Q)$, go to step 5.

3. [Add coefficients] (We have found terms with equal exponents) If $ABC(P) < 0$, the algorithm terminates. Otherwise set COEF(Q) $\leftarrow$ COEF(Q) + COEF(P). Now if COEF(Q) = 0, go to 4; otherwise, set $Q1 \leftarrow Q$, $P \leftarrow$ LINK(P), $Q \leftarrow$ LINK(Q), and go to 2.

4. [Delete zero term]. Set $Q2 \leftarrow Q$, LINK($Q1$) $\leftarrow Q \leftarrow$ LINK(Q), and AVAIL $\Leftarrow Q2$.

(A zero term created in step 3 has been removed from polynomial Q).

Set $P \leftarrow$ LINK(P) and go to 2.

5. [Insert new term] (Polynomial P contains a term that is not present in polynomial Q, so we insert it in polynomial Q).

Set $Q2 \Leftarrow$ AVAIL, COEF($Q2$) $\leftarrow$ COEF(P), $ABC(Q2) \leftarrow ABC(P)$, LINK($Q2$) $\leftarrow Q$, LINK($Q1$) $\leftarrow Q2$, $Q1 \leftarrow Q2$, $P \leftarrow$ LINK(P), and return to step 2.

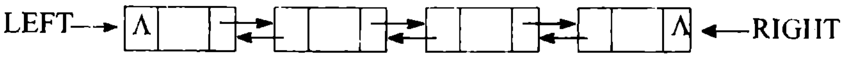
Algorithm M (Multiplication of polynomials)

This algoritm, analogous to algorithm A, replaces polynomial Q by polynomial Q + polynomial $M \times$ polynomial P.

1. [Next multiplier] Set $M \leftarrow$ LINK(M). If $ABC(M) < 0$, the algorithm terminates.

2. [Multiply cycle] Perform algorithm A, except wherever the notation „$ABC(P)$" appears in that algorithm, replace it by „if $ABC(P) < 0$ then -1, otherwise, $ABC(P) + ABC(M)$"; wherever „COEF(P)" appears in that algoritm, replace it by „COEF(P) $\times$ COEF(M)". Then go back to step 1.

§ 5. Doubly linked lists

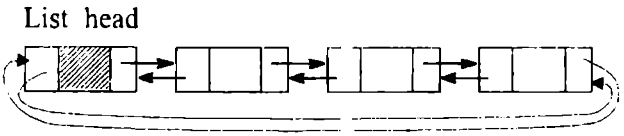


For even greater flexibility in the manipulation of linear lists, we can include two links in each node, pointing to the items on either side of that node.

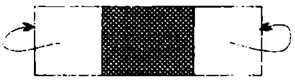
Here LEFT and RIGHT are pointer variables to the left and right of the list.

Each node includes two links, called LLINK and RLINK.

When a list head node is present, we have a typical diagram of a doubly linked list:



If the list is empty, both link fields of the



list head point to the head itself.

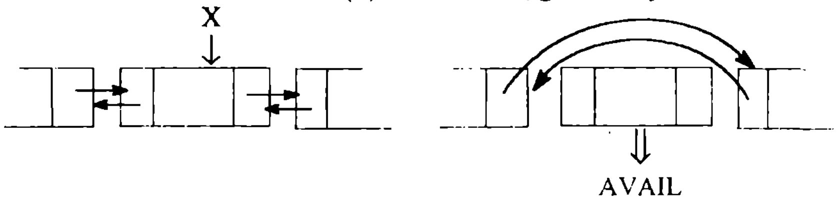
The list representation clearly satisfies the condition

RLINK (LLINK(X)) = LLINK(RLINK (X)) = X

if X is the location of any node in the list (including the head).

A doubly linked list usually takes more memory space than a singly linked one does, but the additional operations that can now be performed efficiently are often more than ample compensation for this extra space requirement.

One can delete NODE (X) from the list, given only the value of X.



RLINK(LLINK(X)) $\leftarrow$ RLINK(X). LLINK(RLINK(X)) $\leftarrow$ LLINK(X). AVAIL $\Leftarrow X$.

The insertion of a node adjacent to NODE (X) at either the left or the right is easy.

$P \Leftarrow$ AVAIL, LLINK(P) $\leftarrow X$. RLINK(P) $\leftarrow$ RLINK(X), LLINK (RLINK(X)) $\leftarrow P$, RLINK(X) $\leftarrow P$

do such an insertion to the right of NODE (X); by interchanging left and right we get the corresponding algorithm for insertion to the left.

Arrays and lists. A generalization of a linear list is a two – dimensional or higher – dimensional array of information. For example, consider the case of an $m \times n$ matrix

$$\begin{pmatrix} A[1,1], A[1,2], \dots, A[1,n] \\ A[2,1], A[2,2], \dots, A[2,n] \\ \vdots \\ A[m,1], A[m,2], \dots, A[m,n] \end{pmatrix}$$

In this two – dimensional array, each node $A[j,k]$ belongs to two linear lists: the „row j" list $A[j,1]$, $A[j,2]$,..., $A[j,n]$, and the „column k" list $A[1,k]$, $A[2,k]$,..., $A[m,k]$.

Similar remarks apply to higher – dimensional arrays of information.

Sequential allocation. When an array is stored in sequential memory locations, storage is usually allocated so that

LOC($A[J, K]$) $= a_0 + a_1 J + a_2 K$,

where a_0, a_1, and a_2 are constants.

The most natural (and most commonly used) way to allocate storage is to let the array appear in memory in the „lexicographic order" of its indices.

In general, given a k-dimensional array with c-word elements $A[I_1,I_2,\dots,I_k]$ for $0 \le I_1 \le d_1$, $0 \le I_2 \le d_2$,..., $0 \le I_k \le d_k$, we can store it in memory as

$$\text{LOC}(A[I_1, I_2, \dots, I_k]) = \text{LOC}(A[0,0,\dots,0]) + c(d_2+1)\dots(d_k+1)I_1 + \dots$$

$$\dots + c(d_k+1)I_{k-1} + cI_k = \text{LOC}(A[0,0,\dots,0]) + \sum_{r=1}^{k} a_r I_r,$$

where $a_r = c \prod_{s=r+1}^{k} (d_s + 1)$.

To see why this formula works, observe that a_r is the amount of memory needed to store the subarray $A[I_1,\ldots,I_r, J_{r+1},\ldots,J_k]$ if $I_1,\ldots,I_r$ are constant and $J_{r+1},\ldots,J_k$ vary through all values $0 \le J_{r+1} \le d_{r+1},\ldots, 0 \le J_k \le d_k$; hence by precisely this amount when I_r changes by 1.

The above method for storing arrays is generally suitable when the array has a complete rectangular structure, i.e., when all elements $A[I_1,I_2,\ldots,I_k]$ are present.

There are many situations in which this is not the case; most common among these is the triangular matrix, where we want to store only the entries $A[j,k]$ for,say, $0 \le k \le j \le n$:

$$\begin{pmatrix} A[0,0] & & & \\ A[1,0] & A[1,1] & & \\ \vdots & & & \\ A[n,0] & A[n,1] & \ldots & A[n,n] \end{pmatrix}$$

We may know that all other entries are zero, or that $A[j,k] = A[k,j]$, so only half of the values need to be stored.

If we want to store the lower triangular matrix in $(n+1)(n+2)/2$ consecutive memory positions, we can now ask for an allocation arrangement of the form

$$\text{LOC}(A[j, k]) = a_0 + f_1(J) + f_2(K)$$

where f_1 and f_2 are functions of one variable.

We have in fact the formula

$$\text{LOC}(A[J, K]) = \text{LOC}(A[0, 0]) + cJ(J+1)/2 + Kc$$

The generalization of triangular matrices in higher dimensions is called a tetrahedral array. A k-dimensional tetrahedral array $A[i_1,i_2,\ldots,i_k]$ satisfies $0 \le i_k \le \ldots \le i_2 \le i_1 \le n$.

If A is stored in lexicographic order of the indices then

$$\text{LOC}(A[I_1,\ldots, I_k]) = \text{LOC}(A[0, 0,\ldots, 0]) + c\sum_{r=1}^{k} \binom{I_r + k - r}{k - r + 1}$$

To see why this formula holds, observe that $c\binom{I_r + k - r}{k - r + 1}$ is the amount of memory needed to store the subarray $A(I_1,\dots,I_{r-1}, J_r,\dots,J_k)$ if $I_1,\dots,I_{r-1}$ are constant and $J_r,\dots,J_k$ vary through all values $0 \le J_k \le \dots \le J_r \le I_r - 1$, i.e. the difference of locations between $A[I_1,\dots, I_{r-1}, 0,\dots,0]$ and $A[I_1,\dots, I_{r-1}, I_r, 0,\dots,0]$. The number of increasing words $J_k \dots J_r$ of length $k - r + 1$ with letters in an alphabet of cardinality I_r is precisely the number of combinations with repetition of a set with I_r elements taken $k - r + 1$ at a time, i.e., $\binom{I_r + k - r}{k - r + 1}$.

Linked allocation. For higher – dimensional arrays of information the nodes can contain k link fields, one for each list the node is in.

Sparse matrices are matrices of large order in which most of the elements are zero.

The goal is to operate on these matrices as though the entire matrix were present, but to save great amounts of memory space because the zero entries need not be represented.

The representation we will discuss consists of circularly linked lists for each row and column. Each node of the matrix contains three words and five fields:

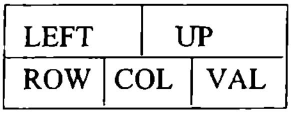
ROW	UP
COL	LEFT
VAL	

Here ROW and COL are the row and column indices of the node; VAL is the value stored at that part of the matrix; LEFT and UP are links to the next nonzero entry to the left in the row, or upward in the column, respectively.

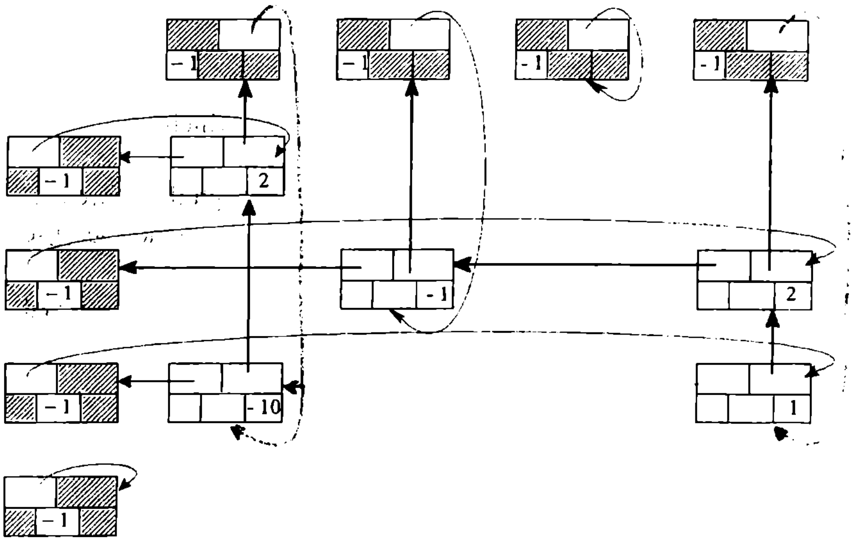
There are special list head nodes, BASEROW [i] and BASECOL [j], for every row and column. These nodes are identified by
COL(LOC(BASEROW[i])) < 0 and ROW(LOC(BASECOL [j])) < 0

As usual in a circular list, the LEFT link in BASEROW [i] is the location of the rightmost value in that row, and UP in BASECOL [j] is the lowest value in that column.

If the nodes are illustrated in the format



the matrix $\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 \\ -10 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ would be represented as shown below:



List heads appear at the left and the top.

The amount of time taken to access a random element $A[j,k]$ is also quite reasonable since most matrix algorithms proceed by walking sequentially through a matrix, instead of accessing elements at random.

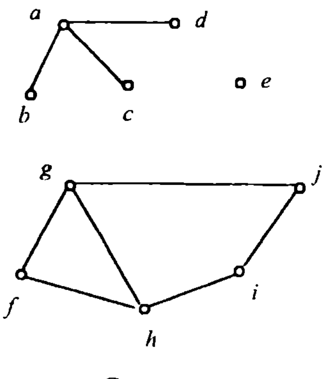
§ 6. Graphs and trees

A graph $G = (V,E)$ is a combinatorial structure consisting of a set of vertices V and a set of edges E. Unless otherwise stated, both are assumed to be finite. Each edge is associated with two vertices called its end points.

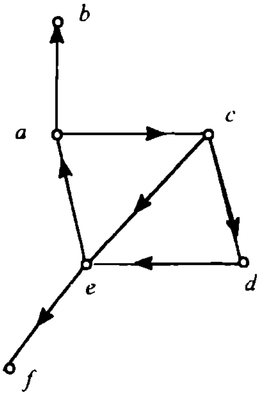
If these two end points have the same relation to the edge, the edge has no natural orientation and is considered undirected. In this case E is a set of unordered pairs of vertices.

If not, we may consider one of end points as the start vertex and the other as the finish vertex, and in this case the edge is considered directed. For directed graphs E is a subset of the cartesian product $V \times V$.

Usually, when we draw a representation of G, the vertices are represented by points and the edges are represented by lines, not necessarily straight. If the edges are directed, we add an arrowhead to specify its direction. The vertex set and edge set of a graph G are also denoted by $V(G)$ and $E(G)$, respectively.



G₁



G₂

The graph G_1 is undirected. It has

$V(G_1) = \{a,b,c,d,e,f,g,h,i,j\}$ and

$E(G_1) = \{ab, ac, ad, fg, fh, gh, gj, hi, ji\}$

G_2 is directed with $V(G_2) = \{a,b,c,d,e,f\}$ and $E(G_2) = \{(a,b), (a,c), (c,e), (e,a), (c,d), (d,e), (e,f)\}$.

The degree of a vertex v, denoted $d(v)$ is the number of edges containing v. In case of a directed graph we may also speak of the indegree $d^-(v)$, and outdegree $d^+(v)$, which are defined in the natural way.

For example, for the graph G_1 we have $d(a) = 3$, $d(e)=0$ (e is an isolated vertex), $d(f) = 2$ etc.

For G_2 we get $d^+(b)=1$ and $d^-(b)=0$; $d^-(c)=1$ and $d^+(c)=2$ etc.

Two vertices joined by an edge (directed or not) are called adjacent.

A walk in a graph is a sequence of vertices $x_1, x_2,\ldots,x_r$ such that x_i is adjacent with x_{i+1} for $1 \le i \le r-1$. For directed graphs (or digraphs) this condition becomes $(x_i, x_{i+1}) \in E(G)$ for every $1 \le i \le r-1$.

If all vertices $x_1,\ldots,x_r$ are pairwise distinct the walk is called a path.

For undirected graphs if $x_1 = x_r$ and no edge is used twice, the walk $x_1,x_2,\ldots,x_r$ $(=x_1)$ is called a cycle.

If the edges of a cycle are directed arbitrarily we obtain a cycle in a digraph.

If every two adjacent edges of a cycle in a digraph have not opposite orientations, the cycle is called a circuit.

An undirected graph is said to be connected if for every two vertices a and b there exists a path: $a,\ldots,b$ connecting them.

If a graph G is not connected, then it has $r \ge 2$ connected components that are defined as maximal connected subgraphs of G (maximality refers to set – inclusion).

For example, G_1 has three connected components, $C_1=\{e\}$, $C_2=\{a, b, c, d\}$ and $C_3=\{f, g, h, i, j\}$.

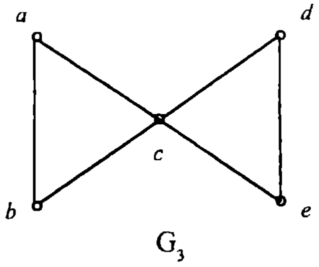
A directed graph is said to be strongly connected if for every two vertices a and b there exists a directed path from a to b. G_2 has 3 strongly connected components: $\{b\}$, $\{f\}$, $\{a,c,d,e\}$.

The connected (strongly connected, resp.) components of a graph G induce a partition of $V(G)$.

Examples

- b,a,d,a,c is a walk for G_1 which is not a path.
- f,g,h,i,j is a path for G_1.
- e,a,c,e,f is a walk for G_2 which is not a path.
- a,c,d,e,f is a path for G_2.
- f,g,h,f and g,h,i,j,g are cycles of G_1.
- e,c,d,e is a cycle of G_2 which is not a circuit.
- e,a,c,d,e is a circuit of G_2.

Note that a cycle can use twice the same vertex. G_3 is in fact a cycle *a, c, e, d, c, b, a* which uses twice vertex *c*. This cycle is in fact the sum of two elementary cycles: *a, c, b, a* and *c, d, e, c*.

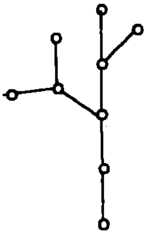


G_3

The *order* of a graph G is by definition $|V(G)|$, the number of its vertices and the size is $|E(G)|$, the number of its edges.

A connected graph that contains no cycle is called a *tree*.

This notion referees mainly to undirected graphs (see for example graph T). The following theorem gives three characterizations of trees:



T

Theorem 1. *Let G be a graph of order $n \geq 3$. The following conditions are equivalent:*

(a) *G is a connected graph without cycles;*

(b) *G is cycle – free maximal (if any missing edge is added to G, a cycle appears);*

(c) *G is connected minimal (if any edge is deleted from G, the connectivity of G is destroyed).*

Proof: We shall prove that (a) $\Rightarrow$ (b) $\Rightarrow$ (c) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): We assume that G is connected and cycle – free. Let $e = ab \notin E(G)$, where $a \neq b$. There is a path in G between a and b; if we add e this path with e form *a* cycle.

(b) $\Rightarrow$ (c): If G would not be connected then by joining by a new edge two vertices belonging to different connected components then no cycle appears, which contradicts (b). If $e \in E(G)$ and $G-e$ is still connected then there exists a path joining the extremities of e in G, hence G contains a cycle, which contradicts (b) again.

(c) $\Rightarrow$ (a): If G would contain a cycle and e is any edge of this cycle then $G-e$ is still connected, which contradicts (c). □

Theorem 2. *If G is a tree of order n, then G has $n-1$ edges.*

Proof: First we shall prove that G contains at least one (in fact two) vertices of degree one (terminal vertices).

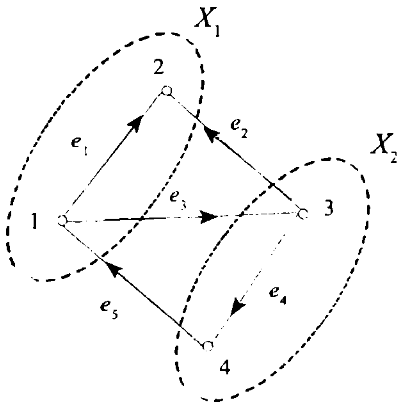
Suppose, to the contrary, that $d(v) \geq 2$ for every $v \in V(G)$. In this case consider a path P of maximum length (= number of edges) of G and let x be an extremity of P. Vertex x has degree at least 2, hence it must be adjacent to at least another vertex of P (by the maximality of P) which produces a cycle in G, a contradiction.

Now the property that G has $n-1$ edges follows easily by induction: it is true for $n=1$ and if we suppose that it is true for all trees of order at most $n-1$ let G be a tree of order n. If x is a terminal vertex of G, then $G-x$ is also a tree of order $n-1$ and the induction hypothesis applies to $G-x$. □

If G has m edges we shall associate to G a vectorial subspace of $\mathbf{R}^m$, denoted by $Z(G)$ and called the cycle space in the following manner:

First we orient in an arbitrary manner the edges of G and any cycle c of G is associated with *a* vector $(c_1, c_2, \ldots, c_m) \in \mathbf{R}^m$ where

$$c_i = \begin{cases} 1, \text{if edge } e_i \in c \text{ has the same orientation as } c; \\ -1, \text{if } e_i \in c \text{ has opposite orientation to } c; \\ 0, \text{otherwise} \end{cases}$$



G

For example cycle 1,2,3,1 is associated to

$c_1 = (1,-1,-1,0,0)$;

$1,3,4,1 \rightarrow (0,0,1,1,1) = c_2$;

$1,2,3,4,1 \rightarrow (1,-1,0,1,1) = c_3$

It is clear that c_1, c_2, c_3 are not linearly independent, since $c_3 = c_1 + c_2$.

Now $Z(G)$ is defined as the vectorial subspace of $\mathbf{R}^m$ spanned by all vectors associated with cycles of G.

Consider now an ordered partition of $V(G)$: $V(G) = X_1 \cup X_2$.

The cocycle (or cutset) ω induced by this partition consists of the set of directed edges (a,b) where $a \in X_1$ and $b \in X_2$ and of the set of directed edges (c,d), where $c \in X_2$ and $d \in X_1$. To ω we associate a vector of $\mathbf{R}^m$, $(\omega_1,\omega_2,\ldots,\omega_m)$, where

$$\omega_i = \begin{cases} 1, \text{if } e_i = (a,b) \text{ and } a \in X_1 \text{ and } b \in X_2; \\ -1, \text{if } a \in X_2 \text{ and } b \in X_1; \\ 0, \text{otherwise.} \end{cases}$$

If $X_1=\{1,2\}$ and $X_2=\{3,4\}$, the cocycle $\omega = (0,-1,1,0,-1)$.

The cocycle space of G, denoted by $U(G)$ is defined as the vectorial subspace of $\mathbf{R}^m$ spanned by all vectors associated with cocycles of G.

If $<,>$ denotes the scalar (or inner) product of vectors in $\mathbf{R}^m$, one can easily check that $<c_1,\omega> = <c_2,\omega> = <c_3,\omega> = 0$. This is a general property for graphs.

Theorem 3. *For any graph G, spaces Z(G) and U(G) are orthogonal.*

Proof: We shall prove that for any cycle c and cocycle ω we have $<c,\omega> = 0$.

Let $X_1 \cup X_2$ be the ordered partition of $V(G)$ inducing ω. It is clear that only edges of c joining a vertex of X_1 with a vertex of X_2 will produce non-vanishing components of $<c,\omega>$. If c goes from X_1 to X_2 this non-vanishing component of the scalar product is always equal to 1 ($1 \cdot 1$ if the corresponding edge of the cycle is from X_1 towards X_2 or $(-1)\cdot(-1)$ if the directed edge has an opposite orientation), and if c goes from X_2 to X_1 this component is equal to -1 ($1\cdot(-1)$ or $(-1)\cdot 1$). Hence $<c,\omega>$ equals the difference between the number of times whenever c passes from X_1 to X_2 and the number of times whenever c passes from X_2 to X_1, which is zero. □

The dimension of $Z(G)$ is denoted by $\mu(G)$ and is called the cyclomatic number of G and the dimension of $U(G)$ is denoted by $\lambda(G)$, the cocyclomatic number of G.

Theorem 4. (Kirchhoff). *Let G be a graph having n vertices, m edges and p connected components. Then* $\mu(G) = m-n+p$ *and* $\lambda(G) = n-p$.

Proof: Suppose first that G is connected ($p = 1$).

Since G is connected, from Theorem 1(c) it follows that there exists a spanning tree T of G (such that $V(T) = V(G)$ and $E(T) \subset E(G)$). T has exactly $n-1$ edges. We shall denote the remaining edges of G by $e_1, e_2, \ldots, e_{m-n+1}$. From Theorem 1 ($b$) it follows that $T+e_i$ contains a cycle c_i for every $1 \le i \le m-n+1$. These $m-n+1$ cycles are linearly independent because each contains an edge that is not contained by the other cycles. It follows that $\mu(G) = \dim Z(G) \ge m-n+1$.

If $E(T) = \{f_1, \ldots, f_{n-1}\}$, it follows that $T-f_i$ has exactly two connected components, C_i^1 and C_i^2 which induce a partition of $V(G)$, hence a cocycle ω_i of G for every $1 \le i \le n-1$. These $n-1$ cocycles are linearly independent by the same argument as for cycles. Hence $\lambda(G) = \dim U(G) \ge n-1$.

It follows that $\mu(G) + \lambda(G) \ge m$. Because spaces $Z(G)$ and $U(G)$ are orthogonal it follows that $\mu(G) + \lambda(G) \le m$ since both are subspaces of $\mathbf{R}^m$. We deduce that all inequalities are equalities and $\mu(G) = m-n+1$ and $\lambda(G) = n-1$ hold.

If G is not connected it has $p \ge 2$ connected components $C_1, C_2, \ldots, C_p$, containing respectively $n_1, \ldots, n_p$ vertices ($n_1 + \ldots + n_p = n$) and $m_1, \ldots, m_p$ edges ($m_1 + \ldots + m_p = m$).

By performing the same algorithm we find in each component C_i, $m_i - n_i + 1$ linearly independent cycles and $n_i - 1$ linearly independent cocycles.

Hence the number of these cycles is equal to $\sum_{i=1}^{p} (m_i - n_i + 1) =$

$= m - n + p$ and of cocycles is $\sum_{i=1}^{p} (n_i - 1) = n - p$ and all these cycles (resp. cocycles) are linearly independent. Since $\mu(G) + \lambda(G) \le m$ it follows, as above, that $\mu(G) = m-n+p$ and $\lambda(G) = n-p$. □

Theorem 5. *Let G be a graph of order $n \ge 3$. The following conditions are equivalent:*

(a) *G is a connected graph without cycles;*

(d) *G is cycle $-$free and has $n-1$ edges;*

(e) *G is connected and has $n-1$ edges;*

(f) *There is a unique path between every pair of distinct vertices of G.*

Proof: We shall prove that (a) $\Rightarrow$ (d) $\Rightarrow$ (e) $\Rightarrow$ (f) $\Rightarrow$ (a).

(a) $\Rightarrow$ (d): $\mu(G) = 0$, i.e., $m-n+1=0$, or $m=n-1$.

(d) $\Rightarrow$ (e): $\mu(G) = 0$ implies that $(n-1)-n+p=0$, or $p=1$, hence G is connected.

(e) $\Rightarrow$ (f): $\mu(G) = (n-1)-n+1=0$, hence G is cycle–free. Because G is connected there exists at least a path between every pair of distinct vertices of G. If for two vertices $x,y \in V(G)$ there exist two paths P_1 and P_2 between x and y ($P_1 \neq P_2$) this implies that G has at least a cycle, which contradicts the property that G is without cycles.

(f) $\Rightarrow$ (a): If G would contain a cycle C we contradict (*f*) since between every pair of distinct vertices of C there are two distinct paths. □

From theorems 1 and 5 we deduce six different characterizations of trees: (a) – (f).

§ 7. Binary trees

The binary trees are the most important nonlinear structures arising in computer algorithms. Generally speaking, tree structure means a branching relationship between nodes, much like that found in the trees of nature.

In the sequel we shall use another notion of tree. In graph-theoretic literature trees as defined below are usually called ordered rooted trees.

Trees consist of internal nodes (branching points) and terminal nodes (or leaves).

Let $V = \{v_1, v_2, \ldots\}$ be an infinite set of internal nodes and let $B = \{b_1, b_2, \ldots\}$ be an infinite set of leaves. We define the set of trees over V and B inductively.

Definition

a) Each element $b_i \in B$ is *a tree*. Then b_i is also *root of the tree.*

b) If $T_1,\ldots,T_m$ ($m \geq 1$) are trees with pairwise disjoint sets of internal nodes and leaves and $v \in V$ is a new node then the $(m+1)$ – tuple $T = < v, T_1,\ldots T_m >$ is a tree. Node v is *the root* of the tree, $\rho(v) = m$ is its *degree* and T_i is the i–th *subtree of T*.

We always draw trees with the root at the top and the leaves at the bottom. Internal nodes are drawn as circles and leaves are drawn as rectangles, but they can be drawn as circles also.

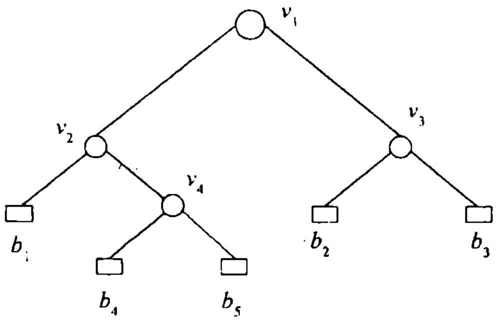


Fig. 1

We use the following terms when we talk about trees. Les T be a tree with root v and subtrees T_i, $1 \le i \le m$. Let w_i = root (T_i). Then w_i is the i–th son of v and v is the father of w_i.

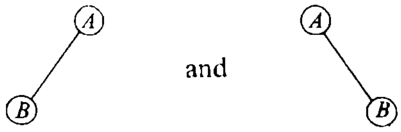
Descendant (ancestor) denotes the reflexive, transitive closure of relation son (father). w_j is a brother of w_i, $j \neq i$. In the tree above, b_1 and v_4 are brothers, v_1 is father of v_2 and v_3 and b_5 is descendant of v_2.

The depth or level of a node v of a tree T is defined as follows: If v is the root of T then depth $(v, T) = 0$. If v is not the root of T then v belongs to T_i for some i. Then depth $(v, T) = 1 +$ depth (v, T_i). We mostly drop the second argument of depth when it is clear from the context.

The height of a tree T is defined as follows: height (T) = max $\{$depth $(b, T) : b$ is a leaf of $T\}$. In our example, we have depth $(v_3) = 1$, depth $(v_4) = 2$, depth $(b_5) = 3$ and height $(T) = 3$. Tree T is a binary tree if all internal nodes of T have degree 1 or 2 and T is called complete if all internal nodes of T have degree exactly 2.

Our example tree is a complete binary tree. *A* complete binary tree with n internal nodes has $n + 1$ leaves. The 1st (2nd) subtree is also called left (right) subtree.

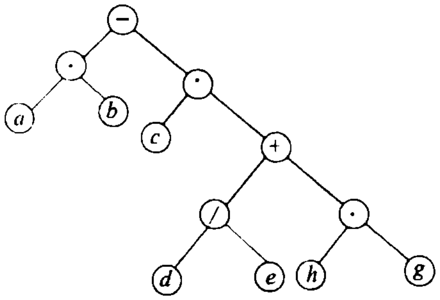
Note that the degree of a vertex (internal node or leaf) of a tree is the number of subtrees of that vertex and leafs have degree zero. Also a binary tree is not a tree from graphtheoretic literature. For example, the binary trees below are distinct (the root has an empty left subtree in one



case and a nonempty left subtree in the other), although as trees they would be identical. Algebraic formulas provide us with another example of tree structure. The arithmetic expression

$$ab - c\,(d/e + hg)$$

may be represented as a binary tree:



Traversing binary trees. Information can be stored in the leaves and nodes of a tree. In some applications we use only one possibility. A binary tree is realized by three arrays LLINK, RLINK and CONTENT or equivalently by records with three fields, and a link variable T which is a pointer to the tree. If the tree is empty, T = Λ; otherwise T is the address of the root of the tree, and LLINK (T), RLINK (T) are pointers to the left and right, respectively, subtrees of the root. These rules recursively define the memory representation of any binary tree.

For example, the binary tree in figure 1 is represented by

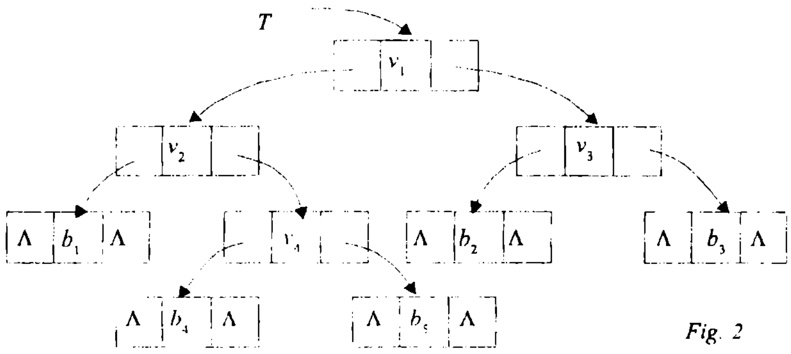


Fig. 2

Systematic exploration of a tree is needed frequently. A binary tree consisting of three components (a root, a left subtree and a right subtree), three methods of tree traversal come naturally:

Preorder traversal: visit the root, traverse the left subtree, traverse the right subtree: Root, *L, R*.

Postorder traversal: traverse the left subtree, traverse the right subtree, visit the root: *L, R,* Root.

Symmetric traversal: traverse the left subtree, visit the root, traverse the right subtree: *L*, Root, *R*.

These three methods are defined recursively; when the binary tree is empty, it is traversed by doing nothing. Symmetrical variants are obtained by interchanging *L* and *R*.

If we apply these definitions to the binary tree from fig. 1, we find that the vertices in preorder are $v_1v_2b_1v_4b_4b_5v_3b_2b_3$, in postorder are $b_1b_4b_5v_4v_2b_2b_3v_3v_1$ and in symmetric order: $b_1v_2b_4v_4b_5v_1b_2v_3b_3$.

In order to traverse a binary tree we usually make use of an auxiliary stack, as in the following algorithm:

Algorithm T (Traverse binary tree in symmetric order). Let *T* be a pointer to a binary tree having a representation as in fig. 2; this algorithm visits all the nodes of the binary tree in symmetric order, making use of an auxiliary stack *A*.

1. [Initialize]. Set stack *A* empty, and set the link variable $P \leftarrow T$.

2. [$P = \Lambda$?] If $P = \Lambda$, go to step **4.**

3. [Stack $\Leftarrow P$]. (Now *P* points to a nonempty binary tree which is to be traversed). Set $A \Leftarrow P$, i.e., push the value of *P* onto stack *A*. Then set $P \leftarrow$ LLINK (*P*) and return to step **2**.

4. [$P \Leftarrow$ Stack]. If stack *A* is empty, the algorithm terminates; otherwise set $P \Leftarrow A$.

5. [Visit NODE (*P*)]. Visit NODE (*P*). Then set $P \leftarrow$ RLINK (*P*) and return to step **2**.

In the final step of this algorithm, the word "visit" means we do whatever activity is intended as the tree is being traversed.

When we get to step **3**, we want to traverse the binary tree whose root is indicated by pointer *P*. The idea is to save *P* on a stack and then to traverse the left subtree; when this has been done, we will get to step **4** and will find the old value of *P* on the stack again. After visiting the root, NODE (*P*) in step **5**, the remaining job is to traverse the right subtree.

Let us prove that algorithm T traverses a binary tree of n nodes in symmetric order, by using induction on n. We shall prove a slightly more general result:

Starting at step **2** with P a pointer to a binary tree of n nodes and with the stack A containing A [1] ... A [m] for some $m \geq 0$, the procedure of steps **2–5** will traverse the binary tree in question in symmetric order and will then arrive at step **4** with stack A returned to its original value A [1] ... A [m].

This statement is obviously true when $n = 0$, because of step **2**. If $n > 0$, let P_0 be the value of P upon entry to step **2**. Since $P_0 \neq \Lambda$, we will perform step **3**, which means that stack A is changed to A [1] ... A [m] P_0 and P is set to LLINK (P_0). Now the left subtree has less than n nodes, so by induction we will traverse the left subtree in symmetric order and will ultimately arrive at step **4** with A [1] ... A [m]P_0 on the stack. Step **4** returns the stack to A [1] ... A [m] and sets $P \leftarrow P_o$. Step **5** now visits NODE (P_0) and sets $P \leftarrow$ RLINK (P_0). Now the right subtree has less than n nodes, so by induction we will traverse the right subtree in postorder and arrive at step **4** as required. The tree has been traversed in symmetric order and the proof is complete. □

An almost identical algorithm may be formulated which traverses binary trees in order Root, L, R by visiting NODE (P) between steps **2** and **3**, instead of between steps **4** and **2**.

If P points to a node of a binary tree, let $P\$$ be the address of successor of NODE (P) in symmetric order and $\$P$ be the address of predecessor of NODE (P) in the same order. If there is no such successor or predecessor of NODE (P), the value LOC (T) is generally used, where T is a pointer to the tree in question.

There is an important alternative to the memory representation of binary trees given in figure 2, which is somewhat analogous to the difference between circular lists and straight one-way lists. It can be proved easily by induction on n that for every binary tree with n nodes represented as in fig. 2 there are always exactly $n + 1$ Λ links (counting T when it is null) and there are n non-null links, counting T, hence there are more null links than other pointers.

In the method called threaded tree representation terminal links are replaced by "threads" to other parts of the tree, as an aid to traversing the tree.

The threaded tree equivalent to the representation in fig. 2 is:

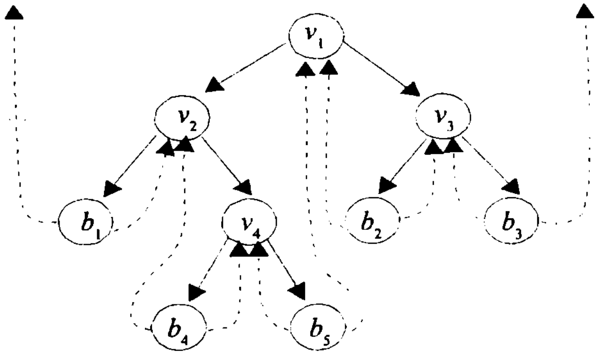


Fig. 3

Here dotted lines represent the threads, which go to a higher node of the tree.

In the memory representation of a threaded binary tree it is necessary to distinguish between the dotted and solid links; this is done by two additional one-bit fields in each node, LTAG and RTAG. The threaded representation may be precisely defined as follows:

Unthreaded representation	Threaded representation
LLINK $(P) = \Lambda$	LTAG $(P) = -$, LLINK $(P) = \$P$
LLINK $(P) = Q \neq \Lambda$	LTAG $(P) = +$, LLINK $(P) = Q$
RLINK $(P) = \Lambda$	RTAG $(P) = -$, RLINK $(P) = P\$$
RLINK $(P) = Q \neq \Lambda$	RTAG $(P) = +$, RLINK $(P) = Q$

Hence each new thread link points directly to the predecessor or successor of the node in question, in symmetric order.

For threaded trees we shall use a list head for the tree, with LLINK (HEAD) = T, RLINK (HEAD) = HEAD, RTAG (HEAD) = +, where T is the pointer to the tree and HEAD denotes the address of the list head.

If the tree is nonempty then LTAG (HEAD) = +; otherwise we have LLINK (HEAD) = HEAD, LTAG (HEAD) = –.

In accordance with these conventions, the computer representation for tree in fig. 1, as a threaded tree is shown in fig. 4.

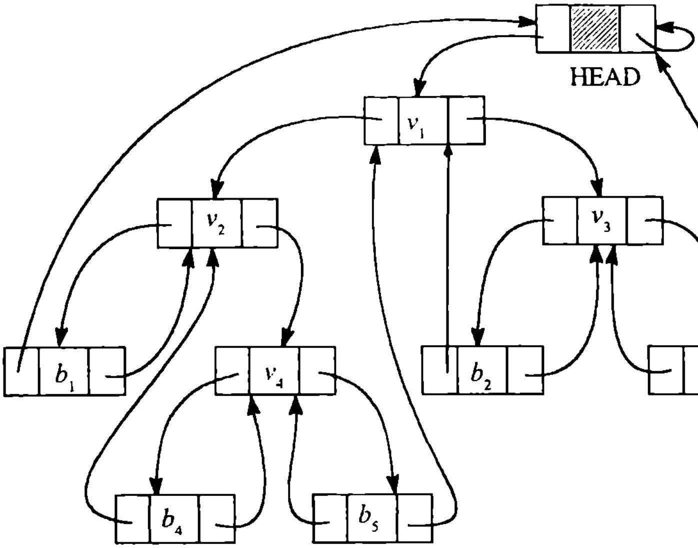


Fig. 4

The advantage of threaded trees is that the traversal algorithms become simpler.

The following algorithm calculates *P*$, given *P*:

Algorithm *S* (Symmetric successor in a threaded binary tree). If P points to a node of a threaded binary tree, this algorithm sets $Q \leftarrow P\$$.

1. [RLINK (*P*) a thread?] Set $Q \leftarrow$ RLINK (*P*). If RTAG (*P*) = –, terminate the algorithm.

2. [Search to left]. If LTAG (*Q*) = +, set $Q \leftarrow$ LLINK (*Q*) and repeat this step. Otherwise the algorithm terminates.

Note that no stack is needed here to accomplish what was done using a stack in Algorithm *T*. Threaded trees grow almost as easily as ordinary ones do.

Algorithm I (Insertion into a threaded binary tree). This algorithm attaches a single node, NODE (*Q*), as the right subtree of NODE (P), if the right subtree is empty, and it inserts NODE (Q) between NODE (P) and NODE (RLINK (P)) otherwise.

1. [*Adjust tags and links*]. Set *RLINK* (Q) ← RLINK (P), RTAG (Q) ← RTAG (P), RLINK (P) ← Q, RTAG (P) ← +, LLINK (Q) ← P, LTAG (Q) ← –.

2. [Was RLINK (P) a thread?]. If RTAG (Q) = +, set LLINK ($Q\$$) ← Q. (Here $Q\$$ is determined by Algorithm **S**).

§ 8. Huffman's algorithm

The vertices of a binary tree have a natural correspondence to words over an alphabet with 2 letters, say $M = \{0,1\}$. For example, vertex H of fig. 5 corresponds to the word 100, because we take first the right son (1), then the left (0), and finally the left (0).

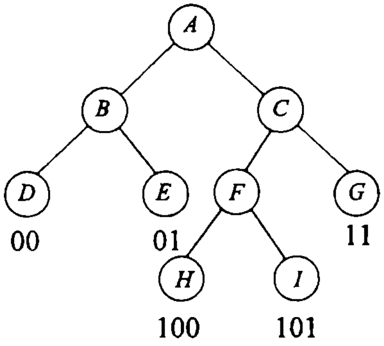


Fig. 5.

Thus, for every vertex of the tree there exists a unique word over M. This is true whether the vertex is terminal (like H) or not (like F, which corresponds to 10). However, there may be words over M which do not correspond to vertices of the tree. Let $l(v)$ (the level of vertex v) be the length of the path from the root to v (or the depth of v): it is equal to the number of letters in the word which corresponds to v.

The set of words which correspond to the terminal vertices of a binary tree forms a prefix (sometimes called instantaneous) code; that is, no word in the code is the beginning of another. Thus, if a sequence of letters is formed by concatenation of words of the code, where repetitions are allowed, the sequence can be decomposed by reading the sequence from left to right and marking off a word as soon as a word of the code is recognized. For example, the code which corresponds to the tree of fig. 5 is {00, 01, 100, 101, 11}. Now, the sequence 000110110100 is easily decomposed, from left to right into 00, 01, 101, 101, 00.

We shall discuss a construction of a binary tree which is optimal in a sense to be discussed shortly. We shall present it as a communication problem, both because it is a natural application, and because historically it was the context of its invention by Huffman. Later we shall point out two more applications: in sorting and in searching.

Assume that we have L basic messages to be transmitted over a communication channel which transfers letters of $M = \{0,1\}$, one at a time. We assume that each letter of M requires the same time to transmit. Also assume that these messages appear one after the other with probabilities $p_1, p_2, \dots, p_L$ and the next message to be sent is chosen with these probabilities, independent of the previous messages. Our purpose is to find a prefix code $C = \{w_1, w_2, \dots, w_L\}$ over M with a vector of word lengths $(l_1, l_2, \dots, l_L)$ such that the average word length $\sum_{i=1}^{L} p_i l_i$ will be minimum. Here $l(w_i) = l_i$ is the length (number of letters) in w_i.

Sorting by merging accesses data in a purely sequential manner. Hence it is very appropriate for sorting with secondary memory, such as disks and tapes. In this context the following problem is of interest. Very often we do not start with sorted sequences of length one but are given n sorted sequences $S_1, \dots, S_n$ of length $w_1, \dots, w_n$ respectively. The problem is to find the optimal order of merging these sequences into a single sequence. Here we assume that it costs $x + y$ time units to merge a sequence of length x with a sequence of length y.

Any merging pattern can be represented as a binary tree with n leaves.

The n leaves represent the n initial sequences and the $n-1$ internal nodes represent the sequences obtained by merging.

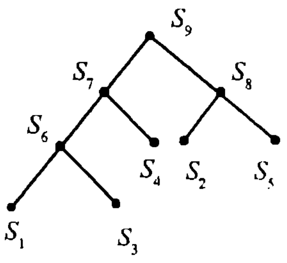
Tree represents the following merging pattern:

$S_6 \leftarrow$ Merge (S_1, S_3)

$S_7 \leftarrow$ Merge (S_6, S_4)

$S_8 \leftarrow$ Merge (S_2, S_5)

$S_9 \leftarrow$ Merge (S_7, S_8)



Here the sum to be minimized is $\sum_{i=1}^{n} w_i d(S_i)$, where w_i is the length of S_i and $d(S_i)$ is its depth (or level) in the tree.

Another application of the Huffman tree to searching problems is the following: in certain cases, data (keys) are stored in the leaves of a binary tree. The question of how to construct this tree when the probabilities of the various data are given and when we want to minimize the average search time is identical with the problem which the Huffman construction solves.

Let us first demonstrate Huffman's construction by means of an example. We shall assume that $p_1 \geq p_2 \geq \ldots \geq p_L$. Let our vector of probabilities be (0.6, 0.2, 0.05, 0.05, 0.03, 0.03, 0.03, 0.01). We shall write it as our top row (see fig. 6). We add the last (and therefore least) two numbers, and put the result (0.04 in our case) in its proper place. We repeat this operation until we get a vector with only two probabilities.

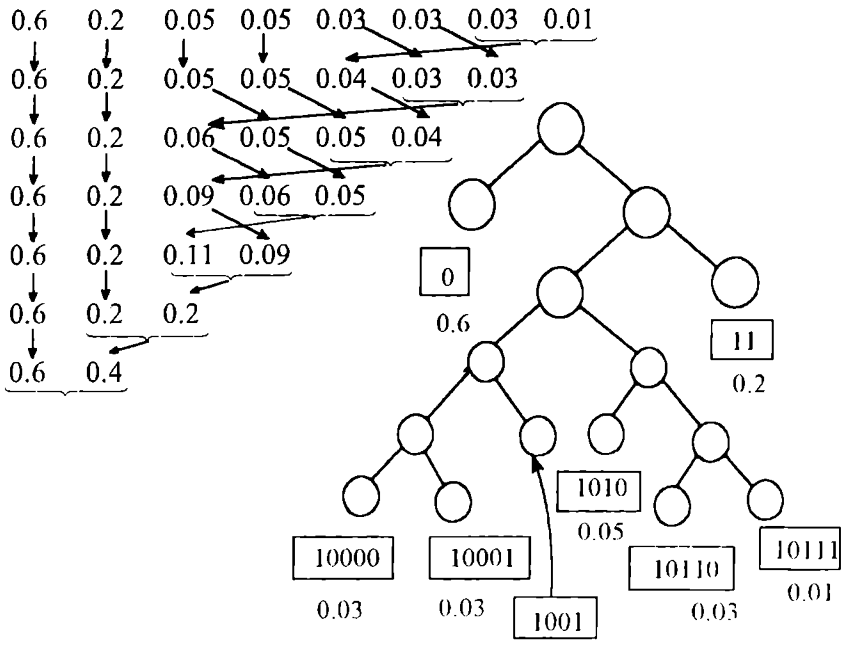


Fig. 6

We have obtained an optimal binary tree.

The fact that the Huffman construction is in terms of probabilities does not matter, since the fact that $p_1 + p_2 + \ldots + p_L = 1$ is never used in the construction or its validity proof.

Definition: Let T be a binary tree with n leaves $v_1,\ldots,v_n$; let CONT: $\{v_1, v_2,\ldots,v_n\} \to \{w_1,\ldots,w_n\}$ be a bijection and let d_i be the depth of leaf v_i. Then Cost $(T) = \sum_{i=1}^{n} d_i$ CONT(v_i) is called the cost of tree T with respect to labelling CONT.

In the case of sorting by merging tree T is a merging pattern, the leaves of T are labelled by the n initial sequences, respectively their lengths (weights). In our example above sequence S_1 is merged three times into larger sequences: with S_3, then as a part of S_6 with S_4 and then as a part of S_7 with S_8. Also three is the depth of the leaf labelled S_1.

In general, a leaf v of depth d is merged d times into larger sequences for a total cost of d CONT(v). Thus the cost of a merging pattern T is as given in the definition above. We want to find the merging pattern of minimal cost.

Definition: Tree T with labelling CONT is optimal if Cost $(T) \leq$ Cost (T') for any other tree T' and labelling CONT$'$.

Theorem. *If* $0 \leq w_1 \leq w_2 \leq \ldots \leq w_n$ *then an optimal tree T and labelling* CONT *can be found in linear time.*

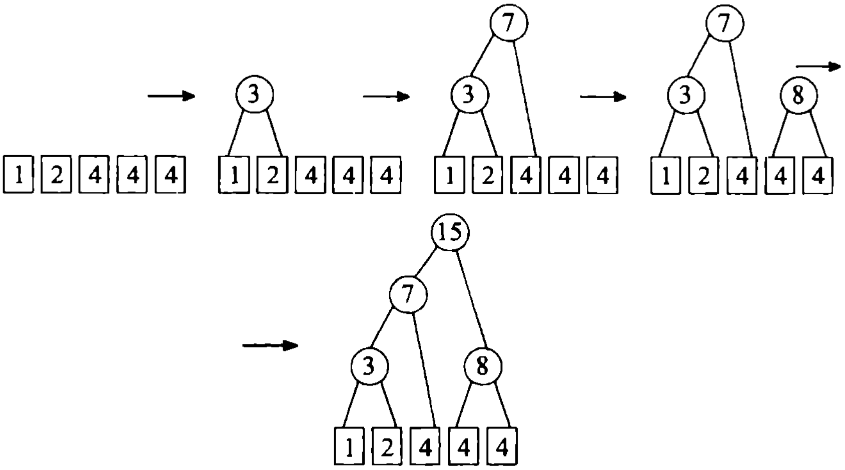
Proof: Huffman's algorithm can be summarized as follows: We construct tree T in a bottom-up fashion. We start with a set $V = \{v_1,\ldots,v_n\}$ of n leaves and labelling CONT $(v_i) = w_i$ for $1 \leq i \leq n$ and an empty set I of internal nodes and set k to zero; k counts the number of internal nodes constructed so far.

while $k < n - 1$

do select $x_1, x_2 \in I \cup V$ with the two smallest values of CONT; ties are broken arbitrarily; construct a new node x with CONT $(x) =$ = CONT (x_1) + CONT (x_2) and add x to I; $k \leftarrow k + 1$; delete x_1 and x_2 from $I \cup V$

od

For $n = 5$ and $\{w_1,\ldots,w_5\} = \{1, 2, 4, 4, 4\}$ we start with 5 leaves of weight 1, 2, 4, 4, 4. In the first step we combine the leaves of weight 1 and 2 and obtain an internal node with weight (content) 3 and so on:



Let T_{opt} with labelling CONT_{opt} be an optimal tree. Let $\{y_1,\ldots,y_n\}$ be the set of leaves of T_{opt}. Assume w.l.o.g. that $\text{CONT}_{\text{opt}}(y_i) = w_i$ for $1 \le i \le n$. Let d_i^{opt} be the depth of leaf y_i in tree T_{opt}.

Lemma 1. If $w_i < w_j$ then $d_i^{opt} \ge d_j^{opt}$ for all i, j.

Proof: Assume otherwise, say $w_i < w_j$ and $d_i^{opt} < d_j^{opt}$ for some i and j. If we interchange the labels of leaves y_i and y_j then we obtain a tree with cost

$$\text{Cost}\,(T_{\text{opt}}) - d_i^{opt}\,w_i - d_j^{opt}\,w_j + d_j^{opt}\,w_i + d_i^{opt}\,w_j =$$

$= \text{Cost}\,(T_{\text{opt}}) - (w_j - w_i)(d_j^{opt} - d_i^{opt}) < \text{Cost}\,(T_{\text{opt}})$, a contradiction. □

Lemma 2. *There is an optimal tree in which the leaves with content w_1 and w_2 are brothers.*

Proof: Let y be a node of maximal depth in T_{opt} and let y_i and y_j be its sons. Then y_i and y_j are leaves. Assume w.l.o.g. that CONT_{opt}

$(y_i) \leq \mathrm{CONT}_{\mathrm{opt}}(y_j)$. From lemma 1 it follows that either $\mathrm{CONT}(y_i) = w_1$ or $d_i \leq d_1$ and hence $d_i = d_1$ by the choice of y. In either case we may exchange leaves y_1 and y_i without affecting cost of the tree. This shows that there is an optimal tree such that y_1 is a son of y. Similarly, we infer from lemma 1 that either $\mathrm{CONT}(y_j) = w_2$ or $d_j \leq d_2$ and hence $d_j = d_2$. In either case we may exchange leaves y_2 and y_j without affecting cost. In this way we obtain an optimal tree in which y_1 and y_2 are brothers. □

Lemma 3. *The Huffman algorithm constructs an optimal tree.*

Proof: (by induction on n). The claim is obvious for $n \leq 2$. Let us assume that $n \geq 3$ and let T_{alg} be the tree constructed by our algorithm for weights $w_1 \leq w_2 \leq \ldots \leq w_n$. The algorithm combines weights w_1 and w_2 first and constructs a node of weight (content) $w_1 + w_2$. Let T'_{alg} be the tree constructed by our algorithm for set (in fact a multiset) $\{w_1 + w_2, w_3, w_4, \ldots, w_n\}$ of weights. Then

$$\mathrm{Cost}\,(T_{\mathrm{alg}}) = \mathrm{Cost}\,(T'_{\mathrm{alg}}) + w_1 + w_2$$

because T_{alg} can be obtained from T'_{alg} by replacing a leaf of weight $w_1 + w_2$ by an internal node with two leaf sons of weight w_1 and w_2, respectively. Also T'_{alg} is optimal for the set of $n - 1$ weights $w_1 + w_2, w_3, \ldots, w_n$ by induction hypothesis.

Let T_{opt} be an optimal tree satisfying lemma 2, i.e. the leaves with content w_1 and w_2 are brothers in T_{opt}. Let T' be the tree obtained from T_{opt} by replacing leaves w_1 and w_2 and their father by a single leaf of weight $w_1 + w_2$. Then

$$\mathrm{Cost}\,(T_{\mathrm{opt}}) = \mathrm{Cost}\,(T') + w_1 + w_2 \geq \mathrm{Cost}\,(T'_{\mathrm{alg}}) + w_1 + w_2 =$$
$= \mathrm{Cost}\,(T_{\mathrm{alg}})$, since $\mathrm{Cost}\,(T') \geq \mathrm{Cost}\,(T'_{\mathrm{alg}})$ by induction hypothesis. It follows that $\mathrm{Cost}\,(T_{\mathrm{alg}}) = \mathrm{Cost}\,(T_{\mathrm{opt}})$. □

It remains to analyse the run time of the algorithm.

Lemma 4. *Let $z_1, z_2, \ldots, z_{n-1}$ be the internal nodes created by the algorithm in this order. Then* $\mathrm{CONT}\,(z_1) \leq \mathrm{CONT}\,(z_2) \leq \ldots \leq \mathrm{CONT}\,(z_{n-1})$. *Furthermore, we always have $V = \{v_i, \ldots v_n\}$, $I = \{z_j, \ldots, z_k\}$ for some $i \leq n + 1$, $j \leq k + 1 \leq n$ when entering the body of the loop.*

Proof: (by induction on k). The claim is true when $k = 0$. In each iteration of the loop we increase k by one and $i + j$ by two. Also $\mathrm{CONT}\,(z_{k+1}) \geq \mathrm{CONT}\,(z_k)$ is immediately from the construction. □

This lemma suggests a linear time implementation. We keep the elements of V and I in two separate sets both ordered according to CONT. Since $w_1 \leq \ldots \leq w_n$ a queue will do for V and since CONT $(z_1) \leq \ldots \leq$ CONT (z_{n-1}) another queue will do for I. It is then easy to select $x_1, x_2 \in I \cup V$ with the two smallest values of CONT by comparing the first two front elements of the queues. Also x_1, x_2 can be deleted in time O(1) and the newly created node can be added to the I – queue in constant time.

Huffman's algorithm can be generalized to non – binary trees (i.e., to m – ary trees, where $m \geq 3$).

§ 9. Marking algorithms for non-available memory

A (general) list is a linear list whose elements may contain pointers to other lists. The common operations we wish to perform on lists are the usual ones desired for linear lists: creation, insertion, deletion, concatenation, splitting, etc. For these purposes any of the three basic techniques for representing linked linear lists in memory - straight, circular, or double linkage can be used. The rest of this paragraph will be devoted to the problem of maintaining the list of available space in the memory.

The garbage - collection technique requires a new one-bit field in each node called the "mark bit". Garbage collection generally proceeds in two phases. We assume that the mark bits of all nodes are initially zero (or we set them all to zero). Now the first phase marks all the nongarbage nodes, starting from those which are immediately accessible to the main program. The second phase makes a sequential pass over the entire memory pool area, putting all unmarked nodes onto the list of free space. The most interesting feature of garbage collection is the fact that while this algorithn is running, there is only a very limited amount of storage available which we can use to control our marking algorithm. Hence it runs very slowly when nearly all the memory space is in use.

Algorithm A (Marking). Let the entire memory used for list storage be NODE (1), NODE (2), ..., NODE (M), and suppose that these nodes either are atoms or contain two link fields ALINK and BLINK.

Assume that all nodes are initially unmarked. The purpose of this algorithm is to mark all of the nodes which can be reached by a chain of ALINK and/or BLINK pointers in nonatomic nodes, starting from a set of "immediately accessible" nodes.

1. Mark all nodes that are "immediately accessible", i.e., the nodes pointed to by certain fixed locations in the main program which are used as a source for all memory accesses. Set $K \leftarrow 1$.

2. Set $K1 \leftarrow K + 1$. If NODE (K) is an atom or unmarked, go to **3**. Otherwise, if NODE (ALINK (K)) is unmarked, mark it, and if it is not an atom, set $K1 \leftarrow$ min ($K1$, ALINK (K)). Similarly, if NODE (BLINK (K)) is unmarked, mark it, and if it is not an atom, set $K1 \leftarrow$min ($K1$, BLINK (K)).

3. Set $K \leftarrow K1$. If $K \leq M$, return to **2**; otherwise the algorithm terminates.

Throughout this algorithm and the ones which follow in this section, we will assume for convenience that the nonexistent node NODE (Λ) is marked and also min ($K1$, Λ) = $K1$.

Algorithm **A** is very slow when n is large. Another marking algorithm follows all paths and record branch points on a stack:

Algorithm B. This algorithm achieves the same effect as Algorithm **A**, using STACK [1] , STACK [2], ..., as auxiliary storage.

1. Let T be the number of immediately accessible nodes; mark them and place pointers to them in STACK [1], ..., STACK [T].

2. If $T = 0$, the algorithm terminates.

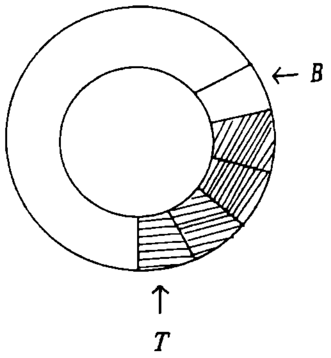
3. Set $K \leftarrow$ STACK [T], $T \leftarrow T - 1$.

4. If NODE(K) is an atom, return to **2**.

Otherwise, if NODE (ALINK(K)) is unmarked, mark it and set $T \leftarrow T + 1$, STACK [T] $\leftarrow$ ALINK(K); if NODE (BLINK(K)) is unmarked, mark it and set $T \leftarrow T + 1$, STACK [T] $\leftarrow$ BLINK (K). Return to **2**.

Algorithm **B** has an execution time proportional to the number of cells it marks; but it can be not really usable for memory marking because there is no place to keep the stack. A better alternative is possible, using a fixed stack size, and combining algorithms **A** and **B**:

Algorithm C. This algorithm achieves the same effect as algorithms **A** and **B**, using an auxiliary table of H cells, STACK [0], STACK



[1], ..., STACK [H–1].In this algorithm, the action "insert X on the stack" means the following:

"Set $T \leftarrow (T + 1) \bmod H$, and STACK [$T$] $\leftarrow X$. If $T = B$, set $B \leftarrow (B + 1) \bmod H$ and $K1 \leftarrow \min$ ($K1$, STACK [B])".

(T points to the current top of the stack, but B points one place below the current bottom).

1. Set $T \leftarrow H - 1$, $B \leftarrow H–1$, $K1 \leftarrow M + 1$. Mark all the immediately accessible nodes, and successively insert their locations onto the stack (as just described above).

2. If $T = B$ (stack empty) go to **5**.

3. Set $K \leftarrow$ STACK [T], $T \leftarrow (T–1) \bmod H$.

4. If NODE (K) is an atom, return to **2**.

Otherwise, if NODE (ALINK (K)) is unmarked, mark it and insert ALINK(K) on the stack. Similarly, if NODE (BLINK(K)) is unmarked, mark it and insert BLINK (K) on the stack. Return to **2**.

5. If $K1 > M$, the algorithm terminates. (The variable $K1$ represents the smallest location where there is a possibility of a new link to a node that should be marked). Otherwise, if NODE ($K1$) is unmarked, increase $K1$ by 1 and repeat this step. If NODE ($K1$) is marked, set $K \leftarrow K1$, increase $K1$ by 1, and go to **4**.

This algorithm and algorithm **B** can be improved if X is never put on the stack when NODE (X) is an atom.

Algorithm **C** is essentially algorithm A when $H = 1$ and algorithm **B** when $H = M$; it is more efficient when H becomes larger.

Algorithm E. Assume that a collection of nodes is given having the following fields:

MARK (a 1-bit field, initially zero in each node), ATOM (another 1-bit field), ALINK (a pointer field), BLINK (a pointer field).

When ATOM = 0, the ALINK and BLINK fields may contain Λ or a pointer to another node of the same format; when ATOM = 1, the contents of the ALINK and BLINK fields are irrelevant to this algorithm.

Given a pointer PO, this algorithm sets the MARK field to 1 in NODE (PO) and in every other node which can be reached from NODE (PO) by a chain of ALINK and BLINK pointers in nodes with ATOM = 0. The algorithm uses three pointer variables, T, Q, and P, and modifies the links during its execution in such a way that all ATOM, ALINK, and BLINK fields are restored to their original settings after completion, although they may be changed temporarily.

```
1. Set $T \leftarrow \Lambda$, $P \leftarrow PO$.
2. Set MARK ($P$) $\leftarrow$ 1.
3. If ATOM ($P$) = 1, go to 6.
4. Set $Q \leftarrow$ ALINK ($P$). If $Q \neq \Lambda$ and MARK ($Q$) = 0, set ATOM ($P$) $\leftarrow$ 1, ALINK ($P$) $\leftarrow T$, $T \leftarrow P$, $P \leftarrow Q$ and go to 2. (Here the ATOM field and ALINK fields are temporarily being altered, so that the list structure in certain marked nodes has been rather drastically changed.
   But these changes will be restored in step 6).
5. Set $Q \leftarrow$ BLINK ($P$). If $Q \neq \Lambda$ and MARK ($Q$) = 0, set BLINK ($P$) $\leftarrow T$, $T \leftarrow P$, $P \leftarrow Q$, and go to 2.
6. (This step undoes the link switching made in step 4 or 5; the setting of ATOM ($T$) tells whether ALINK ($T$) or BLINK ($T$) is to be restored).
   If $T = \Lambda$, the algorithm terminates. Otherwise set $Q \leftarrow T$. If ATOM ($Q$) = 1, set ATOM ($Q$) $\leftarrow$ 0, $T \leftarrow$ ALINK ($Q$), ALINK ($Q$) $\leftarrow P$, $P \leftarrow Q$, and return to 5. If ATOM ($Q$) = 0, set $T \leftarrow$ BLINK ($Q$), BLINK ($Q$) $\leftarrow P$, $P \leftarrow Q$, and return to 6.
```

A proof that this algorithm is valid can be formulated by induction on the number of nodes that are to be marked. One proves at the same time that $P = PO$ at the conclusion of the algorithm.

The interesting idea used in this algoritm can be applied to problems other than garbage collection, for example for tree traversal.

The fastest garbage collection method known combines allgorithms **B** and **E**, like algorithm **C** combines algorithms **A** and **B** [cf. D. Knuth].

Algorithm F. In the second phase of garbage collection this algorithm compacts storage in the following sense:

Let NODE (1), ..., NODE *(M)* be nodes with fields MARK, ATOM, ALINK, and BLINK, as described in algorithm **E**. Assume

MARK = 1 in all nodes that are not garbage. The algoritm relocates the marked nodes, if necessary, so that they all appear in consecutive locations NODE (1), ..., NODE (k), and at the same time the ALINK and BLINK fields of nonatomic nodes are altered if necessary so that the list structure is preserved.

1. Set $L \leftarrow 0$, $K \leftarrow M+1$, MARK (0) $\leftarrow$ 1, MARK (M +1) $\leftarrow$ 0.

2. $L \leftarrow L + 1$, and if MARK (L) = 1 repeat this step.

3. $K \leftarrow K - 1$, and if MARK (K) = 0 repeat this step.

4. If $L > K$, go to step **5**; otherwise set NODE (L) $\leftarrow$ NODE (K), ALINK (K) $\leftarrow L$, MARK (K) $\leftarrow$ 0, and return to **2**.

5. For L = 1, 2, ..., K do the following: Set MARK (L) $\leftarrow$ 0. If ATOM (L) = 0 and ALINK (L) > K, set ALINK (L) $\leftarrow$ ALINK (ALINK(L)).

If ATOM (L) = 0 and BLINK (L) > K, set BLINK (L) $\leftarrow$ ALINK (BLINK(L)).

§ 10. Multilinked structures

A multilinked structure involves nodes with several link fields in each node, not just one or two as in our previous examples.

The problem we will consider arises in connection with writing a compiler program for translating COBOL and related languages.

A programmer who used COBOL may give alphabetic names to the quantities in his program on several levels; for example, he may have two files of data for sales and purchases which have the following structure

```
1 SALES                    1 PURCHASES
  2 DATE                     2 DATE
    3 MONTH                    3 DAY
    3 DAY                      3 MONTH
    3 YEAR                     3 YEAR
  2 TRANSACTION              2 TRANSACTION
    3 ITEM                     3 ITEM
    3 QUANTITY                 3 QUANTITY
    3 PRICE                    3 PRICE
    3 TAX                      3 TAX
    3 BUYER                    3 SHIPPER
      4 NAME                     4 NAME
      4 ADDRESS                  4 ADDRESS
```

This configuration indicates that each item in SALES consists of two parts, the DATE and the TRANSACTION; the DATE is further divided into three parts, and likewise TRANSACTION has five subdivisions. Similar remarks apply to PURCHASES. The relative order of these name indicates the order in which the quantities appear in external representations of the file; note that in this example DAY and MONTH appear in opposite order in the two files. To refer to an individual variable in the example above, it would not be sufficient merely to give the name DAY; it could also write, more completely, DAY OF DATE OF SALES, but in general there is no need to give more qualification than necessary to avoid ambiguity. Thus, NAME OF SHIPPER OF TRANSACTION OF PURCHASES, may be abbreviated to NAME OF SHIPPER, since only one part of the data has been called SHIPPER.

The rules may be stated more precisely as follows:

a) Each name is immediately preceded by an associated positive integer called its level number. A name either refers to an elementary item or else it is the name of a group of one or more items whose names follow. In the latter case, each item of the group must have the same level number, which must be greater than the level number of the group name.

b) To refer to an elementary item or group of items named A_0, the general form is

$$A_0 \text{ OF } A_1 \text{ OF} \dots \text{OF } A_n,$$

where $n \geq 0$ and where, for $0 \leq j < n$, A_j is the name of some item contained directly or indirectly within a group named A_{j+1}. There must be exactly one item A_0 satisfying this condition.

c) If the same name A_0 appears in several places, there must be a way to refer to each use of the name by using qualification.

As an example of rule (c), the data configuration

```
1 A
 2 B
  3 C
  3 D
 2 C
```

would not be allowed, since there is no unambiguous way to refer to the second appearance of C. A programmer may write

MOVE CORRESPONDING α TO β

which moves all items with corresponding names from data area α to data area β. For example, the statement

MOVE CORRESPONDING DATE OF SALES TO DATE OF PURCHASES

would mean that the values of MONTH, DAY and YEAR from the SALES file are to be moved to the variables DAY, MONTH, YEAR in the PURCHASES file.

Hence the problem is to design three algorithms which are to do the following things:

1. To process a description of names and level numbers, putting the relevant information into tables for use in operations **2** and **3**.

2. To determine if a given qualified reference, as in rule (b), is valid, and when it is valid to locate the corresponding data item.

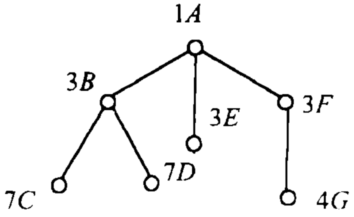
3. To find all corresponding pairs of items indicated by a CORRESPONDING statement. We will assume that a "symbol table subroutine" exists within our compiler, which will convert an alphabetic name into a pointer to a memory location that contains a table entry for that name. In addition to the Symbol Table, there is a larger table which contains one entry for each item of data; we will call this the Data Table.

In each Data Table entry we need five link fields:

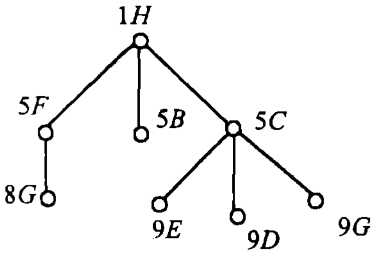
PREV (a link to the previous entry with the same name, if any);
FATHER (a link to the smallest group, if any, containing this item);
NAME (a link to the Symbol Table entry for this item);
SON (a link to the first subitem of a group);
BROTHER (a link to the next subitem in the group containing this item).

It is clear that these data structures are essentially trees. As an example of the multiple linking used, consider the two COBOL data structures

```
1 A
  3 B
    7 C
    7 D
  3 E
  3 F
    4 G
```



1 *H*
  5 *F*
    8 *G*
  5 *B*
  5 *C*
    9 *E*
    9 *D*
    9 *G*



They would be represented as shown below. Note that the LINK field of the Symbol Table entries points to the most recently encountered Data Table entry for the symbolic name in question.

SYMBOL TABLE

		LINK
A		*A* 1
B		*B* 5
C		*C* 5
D		*D* 9
E		*E* 9
F		*F* 5
G		*G* 9
H		*H* 1

DATA TABLE

	PREV	FATHER	NAME	SON	BROTHER
A 1	Λ	Λ	*A*	*B* 3	*H* 1
B 3	Λ	*A* 1	*B*	*C* 7	*E* 3
C 7	Λ	*B* 3	*C*	Λ	*D* 7
D 7	Λ	*B* 3	*D*	Λ	Λ
E 3	Λ	*A*1	*E*	Λ	*F* 3
F 3	Λ	*A*1	*F*	*G* 4	Λ
G 4	Λ	*F* 3	*G*	Λ	Λ
H 1	Λ	Λ	*H*	*F* 5	Λ
F 5	*F* 3	*H*1	*F*	*G* 8	*B* 5
G 8	*G* 4	*F* 5	*G*	Λ	Λ
B 5	*B* 3	*H*1	*B*	Λ	*C* 5
C 5	*C* 7	*H*1	*C*	*E* 9	Λ
E 9	*E* 3	*C*5	*E*	Λ	*D* 9
D 9	*D* 7	*C*5	*D*	Λ	*G* 9
G 9	*G* 8	*C*5	*G*	Λ	Λ

Algorithm A (Build Data Table). This algorithm is given a sequence of pairs (*L*, *P*), where *L* is a positive integer "level number" and *P* points to a Symbol Table entry, corresponding to data structures. This ordered sequence of pairs is in fact produced by traversing the trees (non binary in general) in the order: Root, *L*, *R*. The algorithm builds a Data Table. When *P* points to a Symbol Table entry that has not appeared before, LINK(P) will equal Λ. This algorithm uses an auxiliary stack which is treated as usual (using either sequential or linked allocation).

1. Set the stack contents to the single entry $(0, \Lambda)$. (The stack entries throughout this algorithm are pairs (L, P), where L is an integer and P a pointer; as this algorithm proceeds, the stack contains the level number and pointers to the last data entries on all levels higher in the tree than the current level).

2. Let (L, P) be the next data item from the input. If the input is exhausted the algorithm terminates. Set $Q \Leftarrow$ AVAIL (Q is a location of a new node in which we can put the next Data Table entry).

3. Set PREV(Q) $\leftarrow$ LINK (P), LINK (P) $\leftarrow Q$, NAME(Q) $\leftarrow P$.

4. Let the top entry of the stack be ($L1$, $P1$). If $L1 < L$, set SON($P1$) $\leftarrow Q$ (or, if $P1 = \Lambda$, set FIRST $\leftarrow Q$, where FIRST is a variable which is to point to the first Data Table entry) and go to **6**.

5. If $L1 > L$, remove the top stack entry, let ($L1$, $P1$) be the new entry which has just come to the top of the stack, and repeat step **5**.

If $L1 < L$, signal an error (different numbers have occurred on the same level).

Otherwise, i.e. when $L1 = L$, set BROTHER ($P1$) $\leftarrow Q$, remove the top stack entry, and let ($L1$, $P1$) be the pair which has just come to the top of the stack.

6. Set FATHER(Q) $\leftarrow P1$, SON(Q) $\leftarrow \Lambda$, BROTHER(Q) $\leftarrow \Lambda$.

7. Place (L, Q) on the top of the stack, and return to step **2**.

The next problem is to locate the data table entry corresponding to a reference A_0 OF A_1 OF ... OF A_n, $n \geq 0$.

Algorithm B (Check a qualified reference). Corresponding to reference A_0 OF A_1 OF ... OF A_n, a Symbol Table subroutine will find pointers P_0, P_1, ..., P_n to the Symbol Table entries for A_0, A_1, ..., A_n, respectively.

The purpose of this algorithm is to examine P_0, P_1, ..., P_n and either to determine that this reference is ambiguous, or to set variable Q to the address of the Data Table entry for the item refered to by it.

1. Set $Q \leftarrow \Lambda$, $P \leftarrow$ LINK(P_0).

2. If $P = \Lambda$, the algorithm terminates; at this point Q will equal Λ if the reference does not correspond to any Data Table entry.

Otherwise set $S \leftarrow P$ and $k \leftarrow 0$.

(S is a pointer variable which will run from P up the tree through FATHER links; k is an integer variable which goes from 0 to n. The pointers $P_0, \ldots, P_n$ may be kept in a linked list, and instead of k, we can substitute a pointer variable which traverses this list).

3. If $k < n$ go on to **4**. Otherwise we have found a matching data table entry; if $Q \neq \Lambda$, this is the second entry found, so an error condition is signaled. Set $Q \leftarrow P$, $P \leftarrow$ PREV(P), and go to **2**.

4. Set $k \leftarrow k + 1$.

5. Set $S \leftarrow$ FATHER (S). If $S = \Lambda$, we have failed to find a match; set $P \leftarrow$ PREV(P) and go to **2**.

6. If NAME(S) = P_k, go to **3**, otherwise go to **5**.

Note that the SON and BROTHER links are not needed by this algorithm.

The statement MOVE CORRESPONDING α TO β where α and β are references to data items, is an abbreviation for the set of all statements MOVE α' TO β' where there exists an integer $n \geq 0$ and n names $A_0, A_1, \ldots, A_{n-1}$ such that

$$\begin{aligned} \alpha' &= A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_{n-1} \text{ OF } \alpha \\ \beta' &= A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_{n-1} \text{ OF } \beta \end{aligned} \tag{1}$$

and either α' or β' is an elementary item (not a group item). Furthermore we require that (1) show complete qualifications, i.e., that A_{j+1} is the father of A_j for $0 \leq j \leq n-1$; α' and β' must be exactly n levels farther down in the tree than α and β are.

In our example, MOVE CORRESPONDING A TO H is an abbreviation for the statements MOVE B OF A TO B OF H; MOVE G OF F OF A TO G OF F OF H. The algorithm to recognize all corresponding pairs α', β' proceeds as follows: we move through the tree, whose root is α, in preorder (root, L, R), simultaneously looking in the β tree for matching names, and skipping over subtrees in which no corresponding elements can possibly occur. The names $A_0, \ldots, A_{n-1}$ of (1) are discovered in the opposite order $A_{n-1}, \ldots, A_0$.

Algorithm C (Find CORRESPONDING pairs). Given PO and QO, which point to Data Table entries for α and β, respectively, this algorithm successively finds all pairs (P, Q) of pointers to items (α', β') satisfying the constraints mentioned above.

1. Set $P \leftarrow PO$, $Q \leftarrow QO$. (In the remainder of this algorithm, the pointer variables P and Q will walk through trees having the respective roots α and β).

2. If SON(P) = Λ or SON(Q) = Λ, output (P, Q) as one of the desired pairs and go to **5**. Otherwise set $P \leftarrow$ SON(P), $Q \leftarrow$ SON(Q).

3. (Now P and Q point to data items which have respective qualifications of the forms A_0 OF A_1 OF ... OF A_{n-1} OF α and B_0 OF A_1 OF ... OF A_{n-1} OF β. The object is to see if we can make $B_0 = A_0$ by examining all the names of the group A_1 OF ... OF A_{n-1} OF β).

If NAME(P) = NAME(Q), go to **2** (a match has been found).

Otherwise, if BROTHER(Q) $\neq \Lambda$, set $Q \leftarrow$ BROTHER(Q) and repeat step **3**. (If BROTHER(Q) = Λ, no matching name is present in the group, and we continue on to step **4**).

4. If BROTHER(P) $\neq \Lambda$, set

$P \leftarrow$ BROTHER(P) and $Q \leftarrow$ SON(FATHER(Q)),

and go back to **3**. If BROTHER(P) = Λ, set

$P \leftarrow$ FATHER(P) and $Q \leftarrow$ FATHER(Q).

5. If $P = PO$, the algorithm terminates; otherwise go to **4**.

A proof that this algorithm is valid can readily be constructed by induction on the size of the trees involved.

The five link fields are not all essential, although they are helpful from the standpoint of speed in algorithms **B** and **C**.

This situation is fairly typical of most multilinked structures.

It is interesting to note that we can achieve the effects of algorithms **B** and **C**, using only two link fields and sequential storage of the Data Table, without a very great decrease in speed: PREV (as in the text); SCOPE (link to the last elementary item in this group). We have SCOPE(P) = P if and only if NODE(P) represents an elementary item and NODE(P) is part of the tree below NODE(Q) if and only if $Q < P \leq$ SCOPE(Q).

§ 11. Dynamic storage allocation

We have seen how the use of links implies that tables need not be sequentially located in memory; a number of tables may independently grow and shrink in a common „pooled" memory area. For a great number of applications a single node size is not reasonable; we often wish to have nodes of varying sizes sharing a common memory area. Putting this another way, we want algoritms for reserving and freeing variable-size blocks of memory from a larger storage area, where these blocks are to consist of consecutive memory locations. Such techniques are generally called „dynamic storage allocation" algorithms. Sometimes, often in simulation programs, we want dynamic storage allocation for nodes of rather small sizes (say one to ten words); and at other times, often in „executive" control programs, we are dealing primarily with rather large blocks of information.

For uniformity in terminology between these two approaches, we will generally use the terms block and area rather than „node" in this section, to denote a set of contiguous memory locations.

A. **Reservation**. The problems we want to solve are the following:

a) How is this partitioning of available space to be represented inside the computer?

b) Given such a representation of the available spaces, what is a good algorithm for finding a block of *n* consecutive free spaces and reserving them?

The answer to question (a) is, of course, to keep a list of the available space somewhere; this is almost always done best by using the available space itself to contain such a list.

An exception is the case when we are allocating storage for a disk file or other memory in which nonuniform access time makes it better to maintain a separate directory of available space.

Thus, we can link together the available segments: the first word of each free storage area may contain the size of that block and the address of the next free area. The free blocks can be linked together in increasing or decreasing order of size, or in order of memory address, or in essentially random order. As for question (b), if we want *n* consecutive words, clearly we must locate some block of $m \geq n$ available words and reduce its size to $m - n$.

Furthermore, when $m = n$, we must also delete this block from the list. There may be several blocks with *n* or more cells, and so the question becomes which area should be chosen?

Two principal answers to this question suggest themselves: We can use the „best-fit" method or the „first-fit" method. In the former case, we decide to choose an area with *m* cells, where *m* is the smallest value present which is n or more. This usually requires searching the entire list of available space before a decision can be made. The „first-fit" method, on the other hand, simply chooses the first area encountered that has $\geq n$ words. Historically, the best-fit method was widely used for several years; this naturally appears to be a good policy since it saves the larger available areas for a later time when they might be needed. But several objections to the best-fit technique can be raised:

It is rather slow, since it requires a fairly long search; more important, the best-fit method tends to increase the number of very small blocks, and proliferation of small blocks is usually undesirable.

Algorithm A. (F i r s t – f i t m e t h o d). Let AVAIL point to the first available block of storage, and suppose that each available block with address P has two fields: SIZE(P) , the number of words in the block, and LINK(P), a pointer to the next available block. The last pointer is Λ. This algorithm searches for and reserves a block of N words, or reports failure.

1. Set $Q \leftarrow$ LOC (AVAIL) (Throughout the algorithm we use two pointers, Q and P which are generally related by the condition $P =$ LINK(Q). We assume that

$$\text{LINK(LOC(AVAIL))} = \text{AVAIL}).$$

2. Set $P \leftarrow$ LINK(Q). If $P = \Lambda$, the algorithm terminates unsuccessfully, there is no space for a block of N consecutive words.

3. If SIZE(P) $\geq N$, go to **4**; otherwise set $Q \leftarrow P$ and return to **2**.

4. Set $K \leftarrow$ SIZE(P) $- N$. If $K = 0$, set LINK(Q) $\leftarrow$ LINK(P). Otherwise set SIZE(P) $\leftarrow K$.

The algorithm terminates successfully, having reserved an area of length N beginning with location $P + K$.

Let us temporarily assume, however, that we are primarily interested in large values of N.

We would have been better off if we had reserved the whole block of $N + K$ words when K is very small, instead of saving K extra words.

If we allow the possibility of reserving slightly more than N words, it will be necessary to remember how many words have been reserved, so that later when this block becomes available again the entire set of $N + K$ words is freed. Hence it is usually to expect the SIZE field to be present in the first word of every block whether it is available or not.

In accordance with these conventions, we would modify step 4 above to read as follows:

4′. Set K $\leftarrow$ SIZE $(P) - N$. If $K < c$ (where c is a small positive constant chosen to reflect an amount of storage we are willing to sacrifice in the interests of saving time), set

LINK(Q) $\leftarrow$ LINK(P) and $L \leftarrow P$.

Otherwise set SIZE(P) $\leftarrow K$, $L \leftarrow P + K$ and SIZE(L) $\leftarrow N$.

The algorithm terminates successfully, having reserved an area of length N or more beginning with location L.

When the best-fit method is being used, the test $K < c$ is even more important than it is to the first-fit method, because smaller values of K are much more likely to occur, and the number of available blocks should be kept as small as possible for that algorithm.

B. ***Liberation.*** Now let us consider the inverse problem: How should we return blocks to the available space list when they are no longer needed?

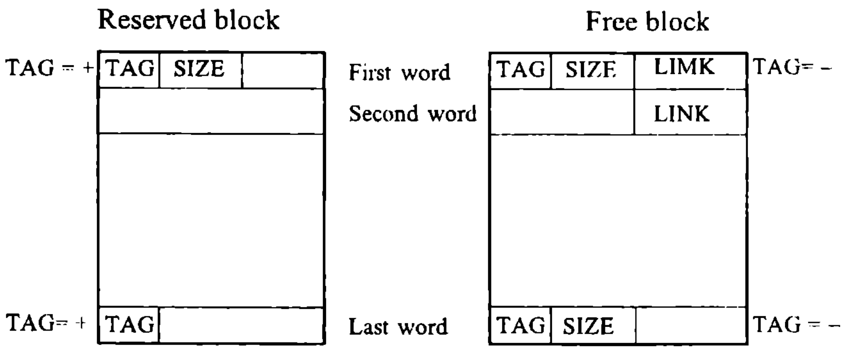
The only difficulty in liberation methods is the collapsing problem: two adjacent free areas should be merged into one. In fact, when an area bounded by two available blocks becomes free, all three areas should be merged together into one. In this way a good balance is obtained in memory even though storage areas are continually reserved and freed over a long period of time (see the „fifty-percent rule" below).

The problem is to determine whether the areas at either side of the returned block are currently available; and if they are, we want to update the AVAIL list properly.

We will consider a method which eliminates all searching when storage is returned to the AVAIL list.

This technique makes use of a TAG field at both ends of each block, and a SIZE field in the first word of each block, this „overhead" is negligible when reasonable large size blocks are being used.

The method we will describe assumes each block has the following form:



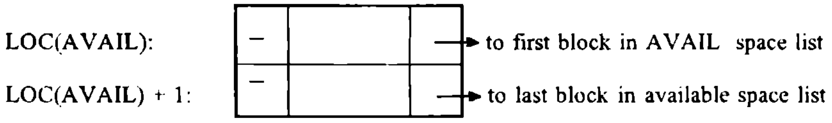
The idea in the following algorithm is to maintain a doubly linked AVAIL list, so that entries may conveniently be deleted from random parts of the list.

The TAG field at either end of a block can be used to control the collapsing process, since we can tell easily whether or not both adjacent blocks are available.

Double linking is achieved by letting the LINK in the first word point to the next free block in the list, and letting the LINK in the second word point back to the previous block; thus, if P is the address of an available block, we always have

LINK(LINK(P)+1) = LINK(LINK(P+1)) = P

To ensure proper „boundary conditions", AVAIL and the following location are set up as follows:


LOC(AVAIL):
LOC(AVAIL) + 1:
−
−
to first block in AVAIL space list
to last block in available space list

A „first-fit" reservation algorithm for this technique may be designed very much like Algorithm A. The principal new feature of this method is the way a block can be freed in essentially a fixed amount of time:

Algorithm C (Liberation with boundary tags). Assume that blocks of locations have the forms shown above, and assume that the AVAIL list is doubly linked. This algorithm puts the block of locations starting with address PO into the AVAIL list. If the pool of available storage runs from locations m_0 through m_1, inclusive, the algorithm assumes for convenience that

$$\mathrm{TAG}\ (m_0 - 1) = \mathrm{TAG}\ (m_1 + 1) = +.$$

1. If TAG(PO – 1) = +, go to **3**.

2. Set $P \leftarrow PO$ – SIZE (PO – 1), and then set $P1 \leftarrow$ LINK(P), $P2 \leftarrow$ LINK(P + 1), LINK($P1$ + 1) $\leftarrow P2$, LINK($P2$) $\leftarrow P1$, SIZE(P) $\leftarrow$ SIZE(P) + SIZE(PO), $PO \leftarrow P$.

3. Set $P \leftarrow PO$ + SIZE(PO). If TAG (P) = +, go to **5**.

4. Set $P1 \leftarrow$ LINK(P), $P2 \leftarrow$ LINK(P+1), LINK($P1$ + 1) $\leftarrow P2$, LINK($P2$) $\leftarrow P1$, SIZE(PO) $\leftarrow$ SIZE(PO) + SIZE(P), $P \leftarrow P$ + SIZE(P).

5. Set SIZE(P – 1) $\leftarrow$ SIZE(PO), LINK(PO) $\leftarrow$ LINK(AVAIL), LINK (PO + 1) $\leftarrow$ LOC(AVAIL), LINK(LINK(AVAIL) + 1) $\leftarrow PO$, LINK(AVAIL) $\leftarrow PO$, TAG(PO) $\leftarrow$ TAG(P – 1) $\leftarrow$ –.

The „buddy system”. This method takes one bit of „overhead” in each block, and it requires all blocks to be of length 2^k($k \geq 0$, k integer). If a block is not 2^k words long for some integer k, the next higher power of 2 is chosen and extra unused space is allocated accordingly. When this method is applicable it has an advantage of speed, especially in „real-time" situations.

The idea of this method is to keep separate lists of available blocks of each size 2^k, $0 \leq k \leq n$. The entire pool of memory space under allocation consist of 2^m words, which we will assume for convenience have the addresses 0 through $2^m - 1$. Originally, the entire block of 2^m words is available.

Later, when a block of 2^k words is desired, and if none of this size are available, a larger available block is split into two equal parts; ultimately, a block of the right size 2^k will appear.

When one block splits into two (each of which is half as large as the original), these two blocks are called buddies.

Later when both buddies are available again, they coalesce back into a single block; thus the process can be maintained indefinitely (unless we run out of space at some point).

If we know the address of a block (i.e., the memory location of its first word), and if we also know the size of that block, we know the address of its buddy.

The address of a block of size 2^k is a multiple of 2^k and this property is easily justified by induction. In general, let $\text{buddy}_k(x)$ = address of the buddy of the block of size 2^k whose address is x; we can prove by induction on k that:

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, \text{if } x \equiv 0 \pmod{2^{k+1}} \\ x - 2^k, \text{if } x \equiv 2^k \pmod{2^{k+1}} \end{cases}$$

The buddy system makes use of a one-bit TAG field in each block:

TAG(P) = 0, if the block with address P is reserved;

TAG(P) = 1, if the block with address P is available.

Besides this TAG field, which is present in all blocks, available blocks also have two link fields, LINKF and LINKB, which are the usual forward and backward links of a doubly linked list. They also have a KVAL field to specify k when their size is 2^k. The algorithms below make use of the table locations AVAIL[0], AVAIL[1] ,..., AVAIL[n], which serve respectively as the heads of the lists of available storage of sizes 1, 2, 4,..., 2^m.

These lists are doubly linked, so as usual the list heads contain two pointers:

AVAILF[k] = LINKF(AVAIL[k])) = link to rear of AVAIL[k] list;

AVAILB[k] = LINKB(LOC(AVAIL[k])) = link to front of AVAIL[k] list.

Initially, before any storage has been allocated, we have

AVAILF[m] = AVAILB[m] = 0,

LINKF[0] = LINKB[0] = LOC(AVAIL[m]),

TAG(0) = 1, KVAL(0) = m

(indicating a single available block of length 2^m, beginning in location 0), and also AVAILF[k] = AVAILB[k] = LOC(AVAIL[k]), for $0 \le k < m$, indicating empty lists for available blocks of lengths 2^k for all $k < m$.

Algorithm R (B u d d y s y s t e m r e s e r v a t i o n). This algorithm finds and reserves a block of 2^k locations, or reports failure, using the organization of the buddy system.

1. Set j be the smallest integer in the range $k \le j \le m$ for which AVAILF[j] ≠ LOC (AVAIL[j]), that is, for which the list of available blocks of size 2^j is not empty. If no such j exists, the algorithm terminates unsuccessfully, since there are no known available blocks of sufficient size to meet the request.

2. Set $L \leftarrow$ AVAILF[j], AVAILF[j] $\leftarrow$ LINKF(L),
LINKB(LINKF(L)) $\leftarrow$ LOC(AVAIL[j]), and TAG(L) $\leftarrow$ 0.

3. If $k = j$, the algorithm terminates (we have found and reserved an available block starting at address L).

4. $j \leftarrow j - 1$. Then set
$P \leftarrow L + 2^j$, TAG(P) $\leftarrow$ 1, KVAL(P) $\leftarrow j$,
LINKF(P) $\leftarrow$ LOC(AVAIL[j]), LINKB(P) $\leftarrow$ LOC(AVAIL[j]),
AVAILF[j] $\leftarrow$ AVAILB[j] $\leftarrow P$.

(This splits a large block and enters the unused half in the AVAIL[j] list which was empty). Go back to step **3.**

Algorithm S (B u d d y s y s t e m l i b e r a t i o n)

This algorithm returns a block of 2^k locations starting in address L to free storage, using the organization of the buddy system.

1. Set $P \leftarrow \text{buddy}_k(L)$. If $k = m$ or if TAG(P) = 0, or if TAG (P) = 1 and KVAL(P) $\neq k$, go to **3**.

2. Set LINKF(LINKB(P)) $\leftarrow$ LINKF(P), LINKB(LINKF(P)) $\leftarrow$ LINKB(P).

(This removes block P from the AVAIL[k] list).

Then set $k \leftarrow k + 1$, and if $P < L$ set $L \leftarrow P$.

Return to **1**.

3. Set TAG(L) $\leftarrow$ 1, LINKF(L) $\leftarrow$ AVAILF[k], LINKB(AVAILF[k]) $\leftarrow L$, KVAL(L) $\leftarrow k$, LINKB(L) $\leftarrow$ LOC(AVAIL[k]), AVAILF[k] $\leftarrow L$.

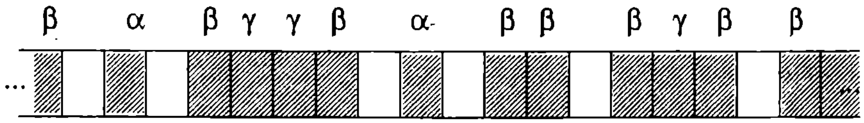
(This puts block L in the top of the AVAIL[k] list).

We can prove an interesting phenomenon, the so-called „fifty-percent rule":

„If algorithms **A** and **C** are used continually in such a way that the system tends to an equilibrium condition, where there are N reserved blocks in the system, on the average, each with an independent lifetime, and where the quantity K in algorithm **A** takes on nonzero values (or values $\geq c$ as in step **4**′) with probability p, then the average number of available blocks tends to approximately $\frac{1}{2}\ pN$".

When the quantity p is near 1 – this will happen if c is very small and if the block sizes are not frequently equal to each other – we have about half as many available blocks as unavailable ones.

In order to deduce this rule consider the following memory map:



The reserved blocks are divided into three types:

α : when freed, the number of available blocks will decrease by one;

β : when freed, the number of available blocks will not change;

γ : when freed, the number of available blocks will increase by one.

Now let N be the number of reserved blocks, and let M be the number of available ones; let A, B and C be the number of blocks of the types α, β and γ, respectively.

We have

$$N = A + B + C$$

$$M = \frac{1}{2}\,(2A + B + \varepsilon),$$

where $\varepsilon = 0$, 1 or 2 depending on conditions at the lower and upper boundaries.

To derive the fifty-percent rule, we set probability that M increases by one = probability that M decreases by one (or the average change in M is set to zero during equilibrium). This leads to

$$\frac{C}{N} = \frac{A}{N} + 1 - p$$

If ε is assumed to be zero (when M and N are assumed to be reasonably large), we get $N - 2M + A = A + (1 - p)N$, or $M = \frac{1}{2}\,pN$ and the rule follows.

Chapter 2

Sorting techniques

Sorting a set with respect to some ordering is a very frequently occuring problem. IBM estimates that about 25% of total computing time is spent on sorting in commercial computing centers. The most important applications of sorting are (according to Knuth):

a) Collecting related things: In an airline reservation system one has to manipulate a set of pairs consisting of passenger name and flight number. Suppose that we keep that set in sorted order of passenger names. Sorting according to flight number then gives us a list of passengers for each flight.

b) Finding duplicates: Suppose that we are given a list of N persons and are asked to find out which of them are present in a group of M persons ($M \leq N$). An efficient solution is to create a list of the persons in the group, sort both lists and then compare them in a sequential scan through both lists.

c) Sorting makes searching simpler as we will see later.

We give a formal definition of the sorting problem. Given is a set of N objects (records) $R_1, R_2, \ldots, R_N$. Each object R_i consists of a key (name) K_i and some information associated with that name. An important fact is that the keys are drawn from some linearly ordered universe U; we use $\leq$ to denote the linear order (e.g., the lexicographic order on words or the usual order relation between real numbers). We want to find a rearrangement of the objects, or a permutation $p(1)p(2)\ldots p(N)$ such that

$$K_{p(1)} \leq K_{p(2)} \leq \ldots \leq K_{p(N)}$$

In some cases we will want the records to be physically rearranged in memory so that their keys are in order, while in other cases it may be sufficient merely to have an auxiliary table of some sort which specifies the permutation.

If the records and/or the keys each take up quite a few words of computer memory, it is often better to make up a new table of link addresses which point to the records, and to manipulate these link addresses instead of moving the bulky records around. This method is called address table sorting (see Fig. 1).

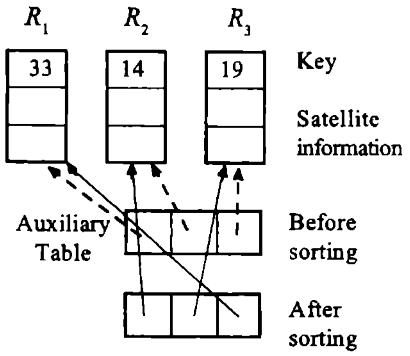


Fig. 1. Address table sorting.

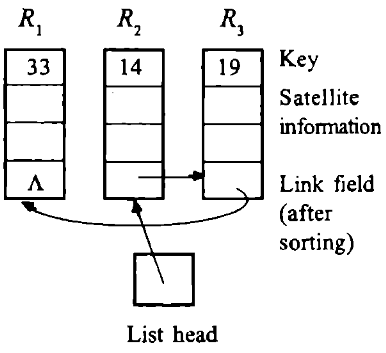


Fig. 2. List sorting.

If the key is short but the satellite information of the records is long, the key may be placed with the link addresses for greater speed; this is called key sorting. Other sorting schemes utilize an auxiliary link field which is included in each record; these links are manipulated in such a way that, in the final result, the records are linked together to form a straight linear list, with each link pointing to the following record. This is called list sorting (see Fig. 2).

After sorting with an address table or list method, the records can be rearranged into increasing order as desired. There are several ways to do this, requiring only enough additional memory space to hold one record; alternatively, we can simply move the records into a new area capable of holding all records. The latter method is usually about twice as fast as the former, but it demands nearly twice as much storage space. It is unnecessary to move the records at all, in many applications, since the link fields are often adequate for subsequent addressing operations.

As an example of the first method we shall describe an efficient algorithm which replaces the N quantities $(R_1,\dots,R_N)$ by $(R_{p(1)},\dots,R_{p(N)})$, respectively, given the values of $R_1,...,R_N$ and the permutation $p(1)...p(N)$ of $\{1,...,N\}$, obtained after an address table sort, without requiring space for storing $2N$ records.

Algorithm P

```
1. Do step 2 for $1 \leq i \leq N$, then terminate the algorithm.
2. Do steps 3 through 5, if $p(i) \neq i$.
3. Set $t \leftarrow R_i, j \leftarrow i$.
4. Set $k \leftarrow p(j), R_j \leftarrow R_k, p(j) \leftarrow j, j \leftarrow k$.
       If $p(j) \neq i$, repeat this step.
5. Set $R_j \leftarrow t, p(j) \leftarrow j$.
```

This algorithm is based on the cycle structure of the permutation p; it changes $p(i)$, since the sorting application lets us assume that $p(i)$ is stored in memory. On the other hand, there are applications such as matrix transposition where $p(i)$ is a function of i which is to be computed (not tabulated) in order to save memory space. In such a case we can use the following method, performing steps $B1$ through $B3$ for $1 \leq i \leq N$:

```
B1. Set $k \leftarrow p(i)$
B2. If $k > i$, set $k \leftarrow p(k)$ and repeat this step.
B3. If $k < i$, do nothing; but if $k=i$ (this means that $i$ is smallest
    in its cycle), we permute the cycle containing $i$ as follows:
```

Set $t \leftarrow R_i$; then while $p(k) \neq i$ repeatedly set $R_k \leftarrow R_{p(k)}$ and $k \leftarrow p(k)$; finally set $R_k \leftarrow t$.

We will discuss sorting algorithms that are generally comparison-based, i.e. they make only use of the fact that the universe is linearly ordered. They belong to the following classes: sorting by counting, sorting by insertion, sorting by exchanging, sorting by selection, sorting by merging; finally we shall discuss minimum - comparison sorting (asymptotic behaviour).

§ 1. Sorting by counting

This simple method is based on the idea that the j-th key in the final sorted sequence is greater than exactly $j-1$ of the other keys. So the idea is to compare each pair of keys, counting how many are less than each particular one. We need merely to ((compare K_j with K_i) for $1 \leq j < i$) for $1 < i \leq N$. Hence we are led to the following algorithm.

Algorithm C (Comparison counting). This algorithm sorts $R_1,\ldots,R_N$ on the keys $K_1,\ldots,K_N$ by maintaining an auxiliary table COUNT [1],...,COUNT [N] to count the number of keys less than a given key. After the conclusion of the algorithm, COUNT [j]+1 specifies the final position of record R_j.

```
1. Set COUNT [1] through COUNT [$N$] to zero.
2. Perform step 3, for $i = N, N-1,\ldots,2$; then terminate the algorithm.
3. Perform step 4, for $j = i-1, i-2,\ldots,1$.
4. If $K_i < K_j$, increase COUNT [$j$] by 1; otherwise increase COUNT
      [$i$] by 1.
```

This algorithm involves no movement of records. It is similar to an address table sort, since the COUNT table specifies the final arrangement of records. But it is somewhat different because COUNT [j] tells us where to move R_j instead of indicating which record should be moved into the place of R_j; thus the „inverse" of the permutation $p(1)\ldots p(n)$ is specified in the COUNT table. Note that algorithm **C** gives the correct result no matter how many equal keys are present.

Since the number of pair comparisons made in step **4** is equal to

$\binom{N}{2} = \frac{N^2 - N}{2}$, the factor N^2 dominates the running time of this algorithm. Hence this is not an efficient way to sort when N is large; we will see later that the average running time can be reduced to order $N \log N$ using the „partition exchange" technique.

There is another way to sort by counting which is quite important from the standpoint of efficiency; it is primarily applicable in the case that many equal keys are present, and when all keys fall into the range $u \le K_j \le v$, where $v - u$ is small.

Algorithm D (Distribution counting). Assuming that all keys are integers in the range $u \le K_j \le v$ for $1 \le j \le N$, this algorithm sorts the records $R_1,\ldots,R_N$ by making use of an auxiliary table COUNT [u],...,COUNT [v].

At the conclusion of the algorithm the records are moved to an output area $S_1,\ldots,S_N$ in the desired order.

```
1. Set COUNT [$u$] through COUNT [$v$] all to zero.
```

2. Perform step 3 for $1 \le j \le N$; then go to 4.
3. Increase the value of COUNT[K_j] by 1.
4. (At this point COUNT[i] is the number of keys which are equal to i). Set
$$\text{COUNT}[i] \leftarrow \text{COUNT}[i] + \text{COUNT}[i-1]$$
for $i = u+1, u+2, \ldots, v$.
5. (At this point COUNT [i] is the number of keys which are less than or equal to i; in particular COUNT [v] = N).

Perform step 6 for $j = N, N-1, \ldots, 1$; then terminate the algorithm.

6. Set $i \leftarrow$ COUNT [K_j]; $S_i \leftarrow R_j$, and
$$\text{COUNT } [K_j] \leftarrow i - 1.$$

Under the conditions stated above, this sorting procedure is very fast.

§ 2. Sorting by insertion

Assume that $1 < j \le N$ and that records $R_1, \ldots, R_{j-1}$ have been rearranged so that
$$K_1 \le K_2 \le \ldots \le K_{j-1}$$
We compare the new key K_j with $K_{j-1}, K_{j-2}, \ldots$, in turn, until discovering that R_j should be inserted between records R_i and R_{i+1}; then we move records $R_{i+1}, \ldots, R_{j-1}$ up one space and put the new record into position $i+1$.

Algorithm S (Straight insertion sort). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \ldots \le K_N$

1. Perform steps 2 through 5 for $j = 2, 3, \ldots, N$; then terminate the algorithm.
2. Set $i \leftarrow j-1$, $K \leftarrow K_j$; $R \leftarrow R_j$.
3. If $K \ge K_i$, go to step 5. (We have found the desired position for record R).
4. Set $R_{i+1} \leftarrow R_i$, then $i \leftarrow i-1$. If $i > 0$, go back to step 3. (If i=0, K is the smallest key found so far, so record R belongs in position 1).
5. Set $R_{i+1} \leftarrow R$.

If we want to make improvements over straight insertion, we need some mechanism by which the records can take long leaps instead of short steps.

Such a method was proposed by Donald Shell; we shall call it the diminishing increment sort. Any sequence $h_t, h_{t-1}, \ldots, h_1$ of increments can be used, so long as the last increment h_1 equals 1.

Algorithm D (Diminishing increment sort). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \ldots \leq K_N$. An auxiliary sequence of increments $h_t, h_{t-1}, \ldots, h_1$ is used to control the sorting process, where $h_1 = 1$; proper choice of these increments can significantly decrease the sorting time. This algorithm reduces to Algorithm **S** when $t=1$.

```
1. Perform step 2 for $s=t,t-1,\ldots,1$; then terminate the algorithm.
2. Set $h \leftarrow h_s$, and perform steps 3 through 6 for $h < j \leq N$. (We
   will use a straight insertion method to sort elements that are $h$
   positions apart, so that $K_i \leq K_{i+h}$ for $1 \leq i \leq N-h$).
3. Set $i \leftarrow j-h$, $K \leftarrow K_j$, $R \leftarrow R_j$.
4. If $K \geq K_i$, go to step 6.
5. Set $R_{i+h} \leftarrow R_i$, then $i \leftarrow i-h$. If $i > 0$, go back to step 4.
6. Set $R_{i+h} \leftarrow R$.
```

§ 3. Sorting by exchanging

We come now to the second family of sorting algorithms: „exchange" or „transposition" methods which systematically interchange pairs of elements that are out of order until no more such pairs exist.

The bubble sort. We compare K_1 with K_2, interchanging R_1 and R_2 if the keys are out of order; then we do the same to R_2 and R_3, R_3 and R_4 etc. During this sequence of operations, records with large keys will move up. Repetitions of the process will get the appropriate records into positions R_N, R_{N-1}, R_{N-2} etc., so that all records will ultimately be sorted.

The method is called „bubble sorting" because large elements „bubble up" to their proper position. After each pass through the file, it is not hard to see that all records above and including the last one to be exchanged must be in their final position, so they need not be examined on subsequent passes.

Algorithm B (Bubble sort). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \leq \ldots \leq K_N$.

1. Set BOUND ← N. (BOUND is the highest index for which the record is not known to be in its final position).
2. Set t ← 0. Perform step 3 for j=1,2,..., BOUND −1, and then go to step 4.
3. If $K_j > K_{j+1}$, interchange $R_j \leftrightarrow R_{j+1}$ and set $t \leftarrow j$.
4. If t=0, the algorithm terminates. Otherwise set BOUND ← t and return to step 2.

Quicksort. Consider the following comparison/exchange scheme: Keep two pointers, i and j, with i=1 and j=N initially. Compare K_i: K_j, and if no exchange is necessary decrease j by 1 and repeat the process. After an exchange first occurs, increase i by 1, and continue comparing and increasing i until another exchange occurs. Then decrease j again, and so on, „burning the candle at both ends", until i=j.

Each comparison will involve the original value of K_1, because it keeps getting exchanged every time we switch directions. By the time that i=j, the original record R_1 will have moved into its final position, since there will be no greater keys to its left and no smaller keys to its right. The original file will have been partitioned in such a way that the original problem is reduced to two simpler problems, sorting $R_1,...,R_{i-1}$ and independently sorting $R_{i+1},...,R_N$. We can apply the same technique to each of these subfiles.

Inside a computer, these subfiles can be represented by two variables l and r (the boundaries of the subfile currently under examination) and a stack of additional pairs (l_k, r_k). Each time the file is subdivided, we put the largest subfile on the stack and commence work on the other one, until we reach trivially short files; this procedure assures that the stack will never contain more than about $\log_2 N$ entries. This sorting procedure is due to C. Hoare and is called partition-exchange sorting.

The partition-exchange (or quicksort partitioning procedure) is suitable for large N; therefore it is desirable to sort short subfiles in a special manner as in the following algorithm.

Algorithm *Q* (Partition-exchange sort). Records $R_1,...,R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \leq .... \leq K_N$.

An auxiliary stack with at most $\log_2 N$ entries is needed for temporary storage. This algorithm follows the quicksort partitioning procedure described in the text above, with slight modifications for extra efficiency:

a) We assume the presence of artificial keys $K_0 = -\infty$ and $K_{N+1} = \infty$ such that $K_0 \le K_i \le K_{N+1}$ for every $1 \le i \le N$.

b) Subfiles of M or fewer elements are sorted by straight insertion, where $M \ge 1$ is a parameter which must be initially chosen.

c) One or two extra comparisons are made during particular stages (allowing the pointers i, j to cross), so that the main comparison loops can be as fast as possible.

d) Records with equal keys are exchanged although it is not strictly necessary to do so (This idea helps to split subfiles nearly in half when equal elements are present).

1. Set the stack empty, and set $l \leftarrow 1$, $r \leftarrow N$.

2. (We now wish to sort the subfile $R_l,\ldots,R_r$; we have $r \ge l-1$, and $K_{l-1} \le K_i \le K_{r+1}$ for $l \le i \le r$). If $r-l < M$, go to step **8**. Otherwise set $i \leftarrow l$, $j \leftarrow r$, $K \leftarrow K_l$, $R \leftarrow R_l$.

3. If $K < K_j$, decrease j by 1 and repeat this step.

4. If $j \le i$, set $R_i \leftarrow R$ and go to 7. Otherwise set $R_i \leftarrow R_j$ and increase i by 1.

5. If $K_i < K$, increase i by 1 and repeat this step.

6. If $j \le i$, set $R_j \leftarrow R$ and $i \leftarrow j$. Otherwise set $R_j \leftarrow R_i$, decrease j by 1, and go to **3**.

7. (Now the subfile $R_l \ldots R_i \ldots R_r$ has been partitioned so that $K_k \le K_i$ for $l \le k \le i$ and $K_i \le K_k$ for $i \le k \le r$).

If $r-i \ge i-1$, insert $(i+1,r)$ on top of the stack and set $r \leftarrow i-1$. Otherwise insert $(l,i-1)$ on top of the stack and set $l \leftarrow i+1$. (Each entry (a,b) on the stack is a request to sort the subfile $R_a \ldots R_b$ at some future time).

Now go back to step **2**.

8. For $j=l+1$, $l+2,\ldots$, until $j > r$ do the following operations: Set $K \leftarrow K_j$, $R \leftarrow R_j$, $i \leftarrow j-1$; then set $R_{i+1} \leftarrow R_i$, $i \leftarrow i-1$ zero or more times until $K_i \le K$; then set $R_{i+1} \leftarrow R$. (This is algorithm of sorting by insertion, applied to a subfile of M or fewer elements).

9. If the stack is empty, we are done sorting; otherwise remove its top entry (l', r'), set $l \leftarrow l'$, $r \leftarrow r'$, and return to step **2**.

Quicksort is an example for a very powerful problem solving method: divide and conquer. A problem is split into several smaller parts (divide) which are then solved using the same algorithm recursively (conquer). Finally the answer is put together from the answers to the subproblems.

We have seen that a good solution encloses an array $S[1...n]$ with two addition elements $S[0]$ and $S[n+1]$ such that $S[0] \leq S[i] \leq S[n+1]$ for all i. Quicksort can be also written as a recursive procedure in a more compact form as follows:

```
            *procedure* Quicksort ($l,r$);
*co* Quicksort ($l,r$) sorts the subarray $S[l],...,S[r]$ into increasing order;
     (1) *begin*  $i \leftarrow l$, $k \leftarrow r+1$, $S \leftarrow S[l]$;
     (2) *while*  $i < k$ *do*
     (3) *begin*  *repeat* $i \leftarrow i+1$ until $S[i] \geq S$;
     (4)          *repeat* $k \leftarrow k-1$ until $S[k] \leq S$;
     (5)          *if* $k > i$ *then* exchange $S[k]$ and $S[i]$
*end*;
     (6) exchange $S[l]$ and $S[k]$;
     (7) *if* $l < k-1$ *then* Quicksort ($l,k-1$);
     (8) *if* $k+1 < r$ *then* Quicksort ($k+1,r$)
*end*
```

Although the maximal number $QS(n)$ of key comparisons which are needed on an array of n elements is quadratic ($QS(n) = O(n^2)$ and this is achieved for example for the array $1,2,...,n$), the average case behaviour is much better.

We analyse it under the assumption that keys are pairwise distinct and that all permutations of the keys are equally likely.

We may then assume w.l.o.g. that the keys are the integers $1,...,n$. Key S_1 is equal to k with probability $1/n$, $1 \leq k \leq n$.

Then subproblems of size $k-1$ and $n-k$ have to be solved recursively and these subproblems are again random sequences, i.e. they satisfy the probability assumption set forth above. This can be seen as follows.

If $S_1=k$, then array S looks as follows just prior to execution of line (6):

$$k\ i_1 i_2 \ldots i_{k-1} j_{k+1} j_{k+2} \cdots j_n$$

Here $i_1...i_{k-1}$ is a permutation of integers $1,...,k-1$ and $j_{k+1},...,j_n$ is a permutation of integers $k+1,...,n$. How did the array look like before the partitioning step? If s interchanges occured in line (5) then there are s

positions in the left subproblem, i.e. among array indices $2,\dots,k$, and s positions in the right subproblem, i.e. among $k+1,\dots,n$, such that the entries in these positions were interchanged pairwise, namely the leftmost selected entry in the left subproblem with the rightmost selected entry in the right subproblem, and so on. Thus there are exactly

$$\sum_{s\geq 0} \binom{k-1}{s}\binom{n-k}{s}$$

arrays before partitioning which produce the array above by the partitioning process.

The important fact to observe is that this expression only depends on k but not on the particular permutations $i_1,\dots,i_{k-1}$ and $j_{k+1},\dots,j_n$. Thus all permutations are equally likely, and hence the subproblems of size $k-1$ and $n-k$ are again random.

Let $QS_{av}(n)$ be the expected number of comparisons on an input of size n. Then

$QS_{av}(0) = QS_{av}(1) = 0$

and

$$QS_{av}(n) = \frac{1}{n}\sum_{k=1}^{n} (n+1+QS_{av}(k-1)+QS_{av}(n-k)) =$$

$$= n+1+\frac{2}{n}\sum_{k=0}^{n-1} QS_{av}(k)$$

for $n \geq 2$.

We solve this recurrence as follows: Multiplication by n gives us:

$$nQS_{av}(n) = n(n+1) + 2\sum_{k=0}^{n-1} QS_{av}(k)$$

Subtracting from this the equality for $n-1$ instead of n, yields

$$\frac{QS_{av}(n)}{n+1} = \frac{2}{n+1} + \frac{QS_{av}(n-1)}{n}.$$

By denoting $P(n) = \dfrac{QS_{av}(n)}{n+1}$ we get

$$P(n) = \frac{2}{n+1} + P(n-1) = \frac{2}{n+1} + \frac{2}{n} + P(n-2) = \ldots$$

$$\ldots = 2\left(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{3}\right) \text{ since } P(1) = 0.$$

But $\frac{1}{n+1} + \ldots + \frac{1}{3} = H_{n+1} - \frac{3}{2}$, where $H_{n+1} = 1 + \frac{1}{2} + \ldots + \frac{1}{n+1}$ is the (n+1)-th harmonic number ($H_n - \ln n \to \gamma \approx 0.57$, Euler's constant, as $n \to \infty$). Hence

$$QS_{av}(n) = 2(n+1)\left(H_{n+1} - \frac{3}{2}\right) \le 2(n+1)\ln(n+1).$$

The run time of the partitioning phase is proportional to the number of comparisons and the total cost of all other operations is $O(n)$. Thus quicksort sorts n elements with run time $O(n^2)$ in the worst case and it uses at most $2(n+1)\ln(n+1)$ comparisons and time $O(n \log n)$ on the average.

Quicksort has quadratic worst case behaviour; the worst case behaviour occurs on the completely sorted sequence. Also almost sorted sequences, which occur frequently in practice, are unfavourable to Quicksort. There is an interesting way out of this dilemma: randomized Quicksort. We change the algorithm by replacing line (1) by:

```
(1a) begin   $i \leftarrow l$; $k \leftarrow r+1$;
(1b)         $j \leftarrow$ a random element of $\{0,\ldots,r-l\}$;
(1c)         interchange $S[l]$ and $S[l+j]$;
(1d)         $S \leftarrow S[l]$;
```

§ 4. Sorting by selection

Another important family of sorting techniques is based on the idea of repeated selection: find the smallest key and move it into its proper position by exchanging it with the record currently occupying that position. Then we need not consider that position again in future selections. This idea yields our first selection sorting algorithm.

Algorithm S. (Straight selection sort). Records $R_1 \ldots R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \ldots \le K_N$.

Sorting is based on the method indicated above, except that it proves to be more convenient to select the largest element first, then the second largest etc.

1. Perform steps **2** and **3** for $j = N, N-1, \dots, 2$.
2. Search through keys $K_j, K_{j-1}, \dots, K_1$ to find a maximal one; let it be K_i.
3. Interchange records $R_i \leftrightarrow R_j$. (Now records $R_j, \dots, R_N$ are in their final position).

The number of comparisons needed for this algorithm is equal to $\binom{N}{2} = \frac{N(N-1)}{2}$, regardless of the values of the input keys, but it involves very little data movement.

Can this algorithm be improved upon the method for finding the maximum ? The answer to this question is no, at least, if we restrict ourselves to comparison-based algorithms. In comparison-based algorithms there is no operation other than the comparison of two elements which is applicable to elements of the universe from which the keys are drawn.

Lemma. *Any comparison-based algorithm needs at least n–1 comparisons to find the maximum of n elements.*

Proof: Interpret a comparison $S_i < S_j$? as a match between S_i and S_j. If $S_i < S_j$ then S_j is the winner and if $S_i \geq S_j$ we can consider that S_i is the winner. If an algorithm uses less than $n-1$ matches (comparisons) then there are at least two players (keys) which are unbeaten at the end of the tournament. Both players could still be best (the maximum), a contradiction. □

This lemma implies that we have to look for a different sorting method if we want to inprove upon the quadratic running time of the naive algorithm. A selection process which finds the largest element must take at least $n-1$ steps; perhaps all sorting methods based on n repeated selections require order n^2 steps? Fortunately this lemma applies only on the first selection step; subsequent selections can make use of previously-gained information. Suppose that we want to sort the sequence 4, 2, 3, 1.

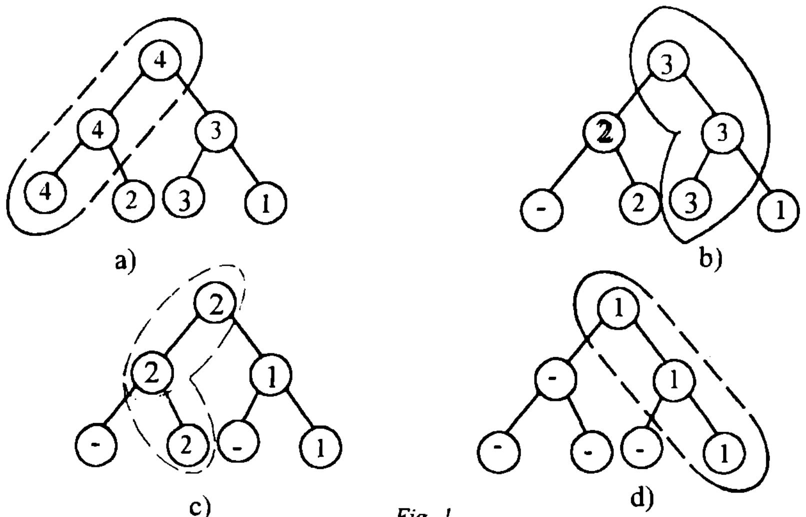


Fig. 1

If we use the complete binary tree 1a), each key being associated to a terminal vertex of this tree, we shall associate to internal vertices keys from this set such that the key of a vertex is equal to maximum of the keys associated with its sons.

For this $n-1$ comparisons are necessary , since every binary tree having n terminal vertices contains $n-1$ internal vertices. New key 4 is the greatest and will be deleted from the tree; to obtain the second largest key we need to compute only the keys associated to the vertices belonging to the path from the root to the terminal vertex which was associated to the current largest key. In our example, this path is encircled by a dotted line in fig. 1a). 2 is moved up and will be compared at the level 1 with 3, giving the result 3 wich will be associated to the root of the tree in fig. 1 b); thus 3 is the second largest element (key) and so on.

In this way the number of key comparisons is equal to

$$3\ (1a) + 1\ (1b) + 1\ (1c) + 0\ (1d) = 5.$$

This tree structure has the following property: Whenever one considers a path through this tree from the root to any terminal vertex then the labels along that path are monotonically decreasing. This is also called the heap property. Figures 1a) – 1d) are complete binary trees with 4 terminal vertices, and it is convenient to represent such a tree in consecutive locations as shown in fig. 2. Note that the father of node

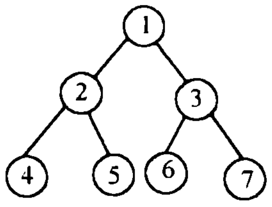


Fig. 2

number k is vertex $\lfloor k/2 \rfloor$, and its sons are vertices $2k$ and $2k+1$. This representation can be easily extended to complete binary trees having a number of terminal vertices which is not of the form $2^p (p \in \mathbf{N})$.

Thus we can store the tree in a single array and save the space for pointers.

Definition. An array $S[1 \ldots n]$ satisfies the heap property if $S[\lfloor k/2 \rfloor] \geq S[k]$ for every $2 \leq k \leq n$. The array is a heap starting at l, $1 \leq l \leq n$, if $S[\lfloor k/2 \rfloor] \geq S[k]$ for $l \leq \lfloor k/2 \rfloor < k \leq n$. ($\lfloor x \rfloor$ is the largest integer $\leq x$).

Notice that every array $S[1 \ldots n]$ is a heap starting at $\lfloor n/2 \rfloor + 1$.

The array 2, 10, 9, 5, 8, 7, 3, 6, 4, 1 is a heap starting at the second position.

If an array $S[1 \ldots n]$ is a heap starting at the first position (or equivalently, satisfies the heap property) then the largest element appears „on top of the heap"

$$S[1] = \max (S[1], S[2], \ldots, S[n]).$$

Indeed, let $m \geq 2$; then $S[m] \leq S[\lfloor m/2 \rfloor] \leq S[\lfloor \lfloor m/2 \rfloor /2 \rfloor] \leq \ldots \leq S[1]$ by the heap property.

If we can somehow transform an arbitrary input file into a heap, we can use a „top-down" selection procedure like that described above to obtain an efficient sorting algorithm. An efficient approach to heap creation has been suggested by R. W. Floyd.

Let us assume that we have been able to arrange the file so that

$$S[\lfloor j/2 \rfloor] \geq S[j] \text{ for } l < \lfloor j/2 \rfloor < j \leq n, \qquad (1)$$

where l is some number ≥ 1 (in the original file this condition holds for $l = \lfloor n/2 \rfloor$, since no subscript j satisfies the condition $\lfloor n/2 \rfloor < \lfloor j/2 \rfloor < j \leq n$). It is not difficult to see how to transform the file so that the inequalities in (1) are extended to the case $\lfloor j/2 \rfloor = l$, working entirely in the subtree whose root is vertex l. Therefore we can decrease l by 1, until heap property is finally achieved. These ideas lead to the following elegant algorithm:

Algorithm H. (Heapsort). Assume that $N \geq 2$.

Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \ldots \leq K_N$. First we rearrange the file so that it forms a heap, then we repeatedly remove the top of the heap and transfer it to its proper final position.

```
1. Set $l \leftarrow \lfloor N/2 \rfloor + 1$, $r \leftarrow N$
2. If $l > 1$, set $l \leftarrow l - 1$, $R \leftarrow R_l$, $K \leftarrow K_l$.
   Otherwise set $R \leftarrow R_r$, $K \leftarrow K_r$, $R_r \leftarrow R_1$, and $r \leftarrow r - 1$; if this makes $r = 1$, set $R_l \leftarrow R$ and terminate the algorithm.
3. Set $j \leftarrow l$.
4. Set $i \leftarrow j$ and $j \leftarrow 2j$. If $j < r$, go to step 5; if $j = r$, go to step 6;
     and if $j > r$, go to 8.
5. If $K_j < K_{j+1}$, then set $j \leftarrow j + 1$.
6. If $K \geq K_j$, then go to step 8.
7. Set $R_i \leftarrow R_j$, and go back to step 4.
8. Set $R_i \leftarrow R$. Return to step 2.
```

At step **2** if $l > 1$ then $K_1, \ldots, K_N$ is a heap starting at l (build-up phase) (when $r = N$); otherwise $l = 1$ and the $N - r$ largest elements are stored in increasing order in $K_{r+1}, \ldots, K_N$ (selection phase). If $l > 1$ we are building the heap and add key K_l; otherwise we are in the selection phase, K_1 is the maximum of $K_1 \ldots K_r$, we exchange K_l and K_r and have to restore the heap property on $K_l, \ldots, K_{r-1}$, by interchanging K_i repeatedly with the larger of its sons.

In the build–up phase $r = N$ and l decreases to 1; in the selection phase, $l = 1$ and r decreases from N to 1.

We will evaluate the comparisons number in the case of sorting by selection algorithm using a tree as in fig. 1.

If $2^r \leq N < 2^{r+1}$ two cases may occur:

(i) $N = 2^r$, and the tree has all terminal vertices on level r. In the first phase we make $N - 1$ comparisons and after this we make $N - 1$ times comparisons at levels $r-2, r-3, \ldots, 1, 0$ (at most), hence the number of key comparisons is bounded above by

$$N - 1 + (N - 1)(r - 1) = (N - 1)r = (N - 1)\log_2 N.$$

(ii) $2^r < N < 2^{r+1}$. In this case we like to build a tree having all terminal vertices on two consecutive levels, r and $r + 1$.

If we denote by x the number of terminal vertices on level r and by y this number for level $r + 1$, we obtain

$$x + y = N$$
$$2x + y = 2^{r+1}$$

It follows that $x = 2^{r+1} - N$ and $y = 2N - 2^{r+1}$. In this case the total number of key comparisons is at most

$N - 1 + (N - 1)r = (N - 1)(r + 1) < (N - 1)(\log_2 N + 1)$ since $r = \lfloor \log_2 N \rfloor$.

Hence the number of key comparisons is bounded above by $N\log_2 N + O(N)$ in both cases.

It is not difficult to show that the number of key comparisons for heapsort algorithm is also of the form $O(N\log N)$.

§ 5. Sorting by merging

Merging means the combination of two or more ordered files into a single ordered file. A simple way to accomplish this is to compare the two smallest items, output the smallest, and then repeat the same process. Some care is necessary when one of the two files becomes exhausted; a detailed description of the process appears in the following algorithm:

Algorithm M. (Two – way merge). This algorithm merges the ordered files $x_1 \leq x_2 \leq \ldots \leq x_m$ and $y_1 \leq y_2 \leq \ldots \leq y_n$ into a single file $z_1 \leq z_2 \leq \ldots \leq z_{m+n}$.

```
1. Set $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$.
2. If $x_i \leq y_j$, go to step 3, otherwise go to 5.
3. Set $z_k \leftarrow x_i$, $k \leftarrow k+1$, $i \leftarrow i+1$. If $i \leq m$, return to 2.
4. Set $(z_k ,\ldots, z_{m+n}) \leftarrow (y_j ,\ldots, y_n)$ and terminate the algorithm.
5. Set $z_k \leftarrow y_j$, $k \leftarrow k+1, j \leftarrow j+1$. If $j \leq n$, return to 2.
6. Set $(z_k ,\ldots, z_{m+n}) \leftarrow (x_i ,\ldots, x_m)$ and terminate the algorithm.
```

From a historical point of view, merge sorting is one of the very first methods proposed for computer sorting; it was suggested by John von Neumann as early as 1945.

In the worst case, each element comes in the sorted sequence $z_1 \leq \ldots \leq z_{m+n}$ as a result of a key comparison (except the last). Hence the number of key comparisons it at most equal to $m+n-1$.

This number of comparisons is reached for example when $x_1 \le x_2 \le \ldots \le x_{m-1} < y_1 \le y_2 \le \ldots \le y_n < x_m$.

If $x_1, \ldots, x_m, y_1, \ldots, y_n$ are pairwise distinct keys and assuming that each of the $\binom{m+n}{n}$ possible arrangements of m x's among n y's is equally likely, we shall find the mean of the number of key comparisons (i.e., of the number of times step **2** is performed during algorithm *M*).

Let C be the number of comparisons; we have $C = m + n - S$, where S is the number of elements transmitted in step **4** or **6**. The probability that $S \ge s$ is easily seen to be

$q_s = \left[\binom{m+n-s}{m} + \binom{m+n-s}{n}\right] \Big/ \binom{m+n}{n}$ for $1 \le s \le m+n$. Hence the mean of S is

$$S_{ave} = q_1 - q_2 + 2(q_2 - q_3) + 3(q_3 - q_4) + \ldots = q_1 + q_2 + q_3 + \ldots =$$

$$= \frac{1}{(m+n)}\left[\sum_{s\ge1}\binom{m+n-s}{m} + \sum_{s\ge1}\binom{m+n-s}{n}\right] = \frac{1}{\binom{m+n}{n}}\left[\binom{m+n}{m+1} + \binom{m+n}{n+1}\right]$$

$= \dfrac{n}{m+1} + \dfrac{m}{n+1}$, by writing each binomial number as a difference of two binomial numbers. Hence $C_{ave} = m + n - \dfrac{n}{m+1} - \dfrac{m}{n+1}$. For $m = n$ this average is asymptotically $2n - 2 + O(n^{-1})$; thus C is close to its maximum value.

Quicksort's bad worst case behaviour stems from the fact that the size of the subproblems created by partitioning is badly controlled. How can we achieve the splitting into two subproblems of size $n/2$ each? There is a simple answer: Take the first half of the input sequence and sort it, take the second half of the input sequence and sort it. Then merge the two sorted subsequences to a single sorted sequence. These considerations lead to the following algorithm.

```
procedure Mergesort ($S$);
begin let $n = |S|$; split $S$ into two subsequences $S_1$ and $S_2$ of
        length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively;
        Mergesort ($S_1$);
        Mergesort ($S_2$);
suppose that the first recursive call produces sequence $x_1 \le x_2 \le \ldots \le x_{\lceil n/2 \rceil}$,
and the second call produces $y_1 \le y_2 \le \ldots \le y_{\lfloor n/2 \rfloor}$; merge the two sequences
into a single sorted sequence $z_1 \le z_2 \le \ \ldots \le \ z_n$
end
```

Note that the merge of the two sequences is performed with algorithm M.

Wė will next compute the number of comparisons which Mergesort uses on an input of length n in the worst case. We use $M(n)$ to denote that number. We have

$M(1) = 0$ and

$M(n) \doteq n - 1 + M(\lceil n/2 \rceil) + M(\lfloor n/2 \rfloor)$, if $n > 1$. We use induction on n to show

$M(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$

This is correct for $n = 1$. So let $n > 1$.

Case 1: $n \neq 2^k + 1$. Then $\lceil \log\lceil n/2 \rceil \rceil = \lceil \log\lfloor n/2 \rfloor \rceil = \lceil \log n \rceil - 1$ and therefore

$$M(n) = n - 1 + \lceil n/2 \rceil \lceil \log\lceil n/2 \rceil \rceil - 2^{\lceil \log\lceil n/2 \rceil \rceil} + 1 +$$
$$+ \lfloor n/2 \rfloor \lceil \log\lfloor n/2 \rfloor \rceil - 2^{\lceil \log\lfloor n/2 \rfloor \rceil} + 1 = n + n(\lceil \log n \rceil - 1) - 2^{\lceil \log n \rceil} + 1 =$$
$$= n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

Case 2: $n = 2^k + 1$. Then $\lceil \log\lfloor n/2 \rfloor \rceil = k - 1 = \lceil \log\lceil n/2 \rceil \rceil - 1 =$ $= \ \lceil \log n \rceil - 2$ and therefore

$$M(n) = n - 1 + \lceil n/2 \rceil(\lceil \log n \rceil - 1) - 2^{\lceil \log n \rceil - 1} + 1$$
$$+ \lfloor n/2 \rfloor(\lceil \log n \rceil - 2) - 2^{\lceil \log n \rceil - 2} + 1 = n\lceil \log n \rceil - \lfloor n/2 \rfloor - 2^{\lceil \log n \rceil} +$$
$$+ 2^{\lceil \log n \rceil - 2} + 1 = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$ since for

$n = 2^k + 1$ we have $\lfloor n/2 \rfloor = 2^{\lceil \log n \rceil - 2}$ (all logarithms are in the base 2).

We introduced Mergesort by way of a recursive program. It is easy to replace recursion by iteration in the case of Mergesort. Split the input sequence into n sequences of length 1. A sequence of lenght 1 is sorted. Then we pair the sequences of lenght 1 and merge them. This gives us $\lfloor n/2 \rfloor$sorted sequences of lenght 2 and maybe one sequence of length 1.

Then we merge the sequences of length 2 into sequences of length 4, and so on.

It is clear that the run time of this algorithm is proportional to the number of comparisons and this is less than or equal to

$$n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 = O(n\log n).$$

§ 6. Optimum sorting

In this section we prove a lower bound on the number of comparisons required for sorting problems. We have seen several sorting methods which are based essentially on comparisons of keys, yet their running time in practice is dominated by other considerations such as data movement. However we shall restrict our discussion to sorting techniques which are based solely on an abstract linear ordering relation $<$ between keys. For simplicity, we shall also confine our discussion to the case of distinct keys, so that there are only two possible outcomes of any comparison between K_i and K_j: either $K_i < K_j$ or $K_i > K_j$.

The problem of sorting by comparisons can also be expressed in other equivalent ways, e.g. : given a set of n players in a tournament, we can ask for the smallest number of games which suffice to rank all contestants, assuming that the strengths of the players can be linearly ordered (with no ties).

All n-element sorting methods which satisfy the above constraints can be represented in terms of an extended binary tree structure such as that shown in fig.1.

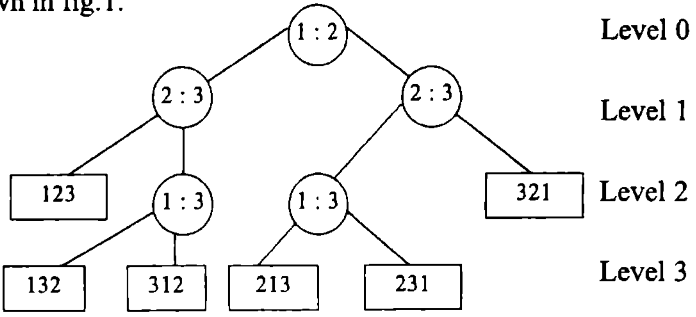


Fig. 1

Each internal vertex (drawn as a circle) contains two indices $i{:}j$ denoting a comparison of K_i versus K_j. The left subtree of this node represents the subsequent comparisons to be made if $K_i < K_j$, and the right subtree represents the actions to be taken when $K_i > K_j$. Each external vertex of the tree (drawn as a box) contains a permutation $a_1a_2...a_n$ of $\{1,2,...,n\}$, denoting the fact that the ordering

$$K_{a_1} < K_{a_2} <...< K_{a_n}$$

has been established.

Thus fig.1 represents a sorting method which first compares K_1 with K_2; if $K_1 > K_2$, it goes on (via the right subtree) to compare K_2 with K_3, and then if $K_2 < K_3$ it compares K_1 with K_3; finally if $K_1 > K_3$ it knows that $K_2 < K_3 < K_1$ and so on.

An actual sorting algorithm will usually also move the keys around in the file, but we are interested here only in the comparisons, so we ignore all data movement. A comparison of K_i with K_j in this tree always means the original keys K_i and K_j, not the keys which might currently occupy the i-th position and j-th position of the file after the records have been shuffled around. It is possible to make redundant comparisons; for example, in fig.1 on the left branch there is no reason to compare 3:1, since $K_1 < K_2$ and $K_2 < K_3$ implies that $K_1 < K_3$. Since we are interested in minimizing the number of comparisons, we may assume that no redundant comparisons are made; hence we have an extended binary tree structure in which every external vertex corresponds to a permutation.

All permutations of the input keys are possible, and every permutation defines a unique path from the root to an external vertex; it follows that there are exactly $n!$ external vertices in a comparison tree which sorts n elements with no redundant comparisons.

Definition. A *decision tree* is a binary tree whose vertices have labels of the form $K_i{:}K_j$. The two outgoing edges are labelled $\leq$ and $>$. Let $K_1,...,K_n$ be elements of universe U. The computation of decision tree T on input $K_1,...,K_n$ is defined in a natural way. We start in the root of the tree. Suppose now that we reached vertex v which is labelled $K_i{:}K_j$. We then compare K_i with K_j and proceed to one of the sons of v depending on whether $K_i \leq K_j$ (or $K_i < K_j$ in the case of pairwise different keys) or $K_i > K_j$.

The leaves of a decision tree represent the different outcomes of the algorithm.

Definition. Let T be a decision tree. *T solves the sorting problem of size n* if there is a labelling of the leaves of T by permutations of $\{1,\ldots,n\}$ such that for every input $K_1,\ldots,K_n$: if the leaf reached on $K_1,\ldots,K_n$ is labelled by π then $K_{\pi(1)} \le K_{\pi(2)} \le \ldots \le K_{\pi(n)}$. We can now define the worst case and average case complexity of the sorting problem. For a decision tree T and permutation π let l_π^T be the depth of the leaf which is reached on input $K_1,\ldots,K_n$ with $K_{\pi(1)} \le K_{\pi(2)} \le \ldots \le K_{\pi(n)}$ (the number of comparisons needed to sort file $K_1,\ldots,K_n$). Define

$$S(n) = \min_T \max_\pi \; l_\pi^T$$

$$A(n) = \min_T \; \tfrac{1}{n!} \sum_\pi l_\pi^T$$

where T ranges over all decision trees for sorting n elements and π ranges over all permutations of n elements. Since l_π^T is the number of comparisons used on input π by decision tree T, or the depth of the terminal vertex with label π, it follows that $\max_\pi \; l_\pi^T = h(T)$, the height of T. Thus $S(n)$ is the minimum worst case complexity of any algorithm and $A(n)$ is the minimum average case complexity of any algorithm. We prove lower bounds on $S(n)$ and $A(n)$.

Suppose $S(n) \le k$. A tree of depth $\le k$ has at most 2^k leaves. A decision tree for sorting n elements has at least $n!$ leaves. Thus

$$2^{S(n)} \ge n! \text{ or } S(n) \ge \lceil \log n! \rceil \qquad (1)$$

Note that by Stirling 's approximation,

$$n! \sim \sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n \text{or } \log n! = \left(n+\tfrac{1}{2}\right)\log n - \tfrac{n}{\ln 2} + O(1) =$$

$$= n\log n - 1.43n + O(\log n).$$

Relation (1) is often called the „information theoretic lower bound". An upper bound for $S(n)$ comes from the analysis of sorting algorithms. In particular, we infer from the analysis of sorting by selection that $S(n) \leq n\log n + O(n)$, and hence $\lim_{n\to\infty} \frac{S(n)}{n\log n} = 1$. (2)

Although we know that $S(n) \sim n\log n$ only few values of $S(n)$ were found (e.g. S(3)=3, S(4)=5, S(5)=6, etc.). In order to find a lower bound on $A(n)$ consider once again the decision tree T representing a sorting procedure, as shown in fig.1. The average number of comparisons in that tree is $\frac{2+3+3+3+3+2}{6} \cong 2.66$, averanging over all permutations.In general, the average number of comparisons in a sorting method is $E(T)/n!$, where $E(T)$ denotes the external path length of the tree, defined as the sum of the distances from the root to each of the external nodes (leaves) of the tree.

Lemma. *If T is a binary tree having N external vertices then E(T) is minimum if and only if all external vertices of T belong to at most two consecutive levels* (*when there are* $2^q - N$ *external vertices at level* $q-1$ *and* $2N-2^q$ *at level q, where* $q = \lceil \log_2 N \rceil$, *the level of the root being* 0).

Proof. Suppose that T contains terminal vertices u and v on level L, y and z on level l, such that $L - l \geq 2$.

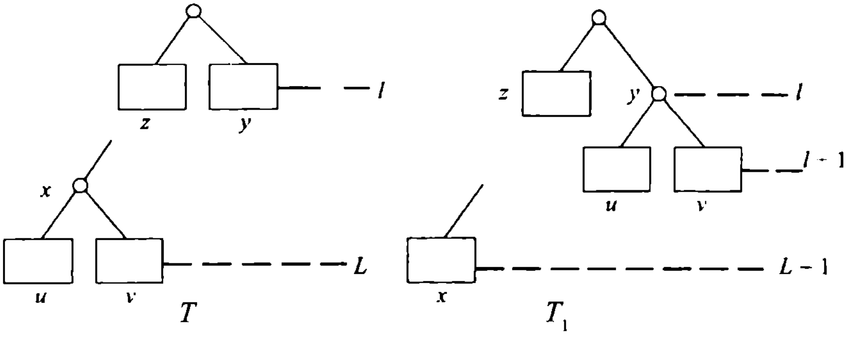


Fig. 2.

We shall define another binary tree T_1 (see fig.2) by transfering u and v on level $l + 1$ as sons of y; x becomes an external vertex and y an internal one. It follows that $E(T) - E(T_1) = 2L-(L-1) + l - 2(l+1) =$

$= L-l-1 \geq 1$ and T cannot have a minimum external path length. Hence T has all external vertices on a single level if N is a power of 2 or on two consecutive levels: $q-1$ and q otherwise, where $q = \lceil \log_2 N \rceil$.

The number of vertices on each level can be obtained in the same way as on pp. 73–74.

It follows that

$$A(n) \geq \tfrac{1}{N}\left[(q-1)(2^q - N) + q(2N - 2^q)\right] = \tfrac{1}{N}\left[(q+1)N - 2^q\right],$$ where $N = n!$.

If we set $q = \log N + \theta$, where $0 \leq \theta < 1$, the formula for minimum external path length becomes

$$N(\log N + 1 + \theta - 2^\theta)$$

The function $1 + \theta - 2^\theta$ for $0 < \theta < 1$ is positive but very small, never exceeding $1 - \frac{1 + \ln\ln 2}{\ln 2} \cong 0.0861$ and vanishes for $\theta = 0$ and $\theta = 1$. In this way we get a lower bound for the average number of comparisons in any sorting scheme, of the form

$$\log n! + 1 + \theta - 2^\theta = n \log n - \tfrac{n}{\ln 2} + O(\log n)$$

In general, the problem of minimizing the average number of comparisons turns out to be substantially more difficult than the problem of determining $S(n)$.

For example, when $n = 7$ it has been shown that no sorting method can attain this lower bound on external path length, but it is possible to construct procedures which do achieve the lower bound $(q+1)N - 2^q$ when $n = 9$ or 10.

§ 7. Sorting by distribution

1. *Sorting words.* Let Σ be a finite alphabet of size m and $\leq$ be a linear order on Σ. We may assume w.l.o.g. that $\Sigma = \{1,2,\ldots,m\}$ with the usual ordering relation. The ordering on Σ is extented to an ordering on the words over Σ as usual.

Definition. Let $x = x_1 \ldots x_k$ and $y = y_1 \ldots y_l$ be words over Σ, i.e. $x_i, y_j \in \Sigma$. Then x is smaller than y in the alphabetic ordering (denoted

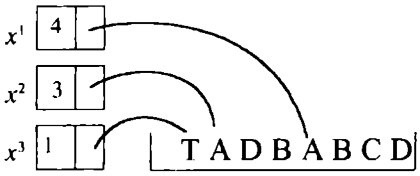


Fig. 1

$x < y$) if there is an i, $0 \leq i \leq k$ such that $x_j = y_j$ for $1 \leq j \leq i$ and either $i = k < l$ or $i < k$, $i < l$ and $x_{i+1} < y_{i+1}$. For example, we have $x = ABCE < AD = y$ because $x_1 = y_1$ and $x_2 < y_2$.

We treat the following problem in this section: Given n words $x^1, x^2, \ldots, x^n$ over alphabet $\Sigma = \{1, \ldots, m\}$ sort them according to alphabetic order. There are many different ways of storing words. A popular method stores all words in a common storage area called string space. The cha-racters of any word are stored in consecutive storage locations. Each string variable then consists of a pointer into the string space and a location containing the current length of the word. The figure 1 shows an example for $x^1 = ABCD$, $x^2 = ADB$ and $x^3 = T$. The basic idea of bucketsort is most easily explained when we consider a special case first: all words $x^1 \ldots x^n$ have length 1, i.e. the input consists of n numbers between 1 and m. Bucketsort starts with m empty buckets. Then we process word by word, throwing x^i into the x^i-th bucket. Finally, we step through the buckets in order and collect the words in sorted order. Buckets are linear lists, the heads of the lists are stored in array $K[1 \ldots m]$. Initially, we have to create m empty lists. This is easily done in time $O(m)$ by initializing array K. In order to process x^i we have to access $K[x^i]$ and to insert x^i into the list with head $K[x^i]$. This can be done in time $O(1)$ if we insert x^i at the front end of the list or at the back end of the list. In the latter case $K[x^i]$ must also contain a pointer to the end of the list. Thus we can distribute n words in time $O(n)$. Finally we have to collect the buckets. We step through array $K[1 \ldots m]$ and concatenate the front of the $(j+1)$-st list with the end of the j-th list. This takes time $O(m)$, if array K also contains pointers to the back ends of the lists, and time $O(n + m)$ otherwise. Note that the total length of all m lists is n. In either case total running time is $O(n + m)$. Should we add x^i to the front or to the rear end of the x^i-th list? If we always add to the rear, then the order of elements x^i, x^j with $x^i = x^j$ is unchanged, i.e. bucketsort is stable. This will be important for what follows. We proceed to a slightly more difficult case next. The x^i's, $1 \le i \le n$, are proper words and suppose that all of them have equal length, say k. Then $x^i = x_1^i x_2^i \dots x_k^i$ with $x_j^i \in \Sigma$. Now one sorts according to the last letter first. After having done so we sort the entire list of n words, which is sorted according to the last letter, according to the next to last letter. The words are sorted according to the last two letters now, because bucketsort is stable. Next we sort according to the (k --2)-th letter, and so on.

This approach requires k passes over the set of n words, each pass having a cost of $O(n + m)$ time units. Thus total running time is $O(k(n + m))$. Let us consider an example with $m = 4$, $n = 5$ and $k = 3$. The words are: 124, 123, 324, 223, 321. The first pass yields:

Bucket	1	321	The imput sequence for the second pass is 321, 123, 223, 324, 124
	2	∅	
	3	123, 223	
	4	324, 124	
Bucket	1	∅	The input sequence for the third pass is: 321, 123, 223, 324, 124 The third pass yields:
	2	321, 123, 223. 324, 124	
	3	∅	
	4	∅	
Bucket	1	123, 124	and hence the final result sequence is 123, 124, 223, 321, 324
	2	223	
	3	321, 324	
	4	∅	

Notice that we collected a total of 3·4 = 12 buckets, but only 7 buckets where non-empty altogether. It would improve running time if we knew ahead of time which buckets to collect in each pass. Let us assume that s_j buckets are non-empty in the j-th pass, $1 \le j \le k$. If we could avoid looking at empty buckets then the cost of the j-th pass were only $O(s_j + n)$. Since $s_j \le n$, the cost of a pass would be only $O(n)$ instead of $O(n + m)$.

There is a very simple method for determining which buckets are non-empty in the j-th pass, i.e. which letters occur in the j-th position. Create set $\{(j, x_j^i), 1 \le j \le k, 1 \le i \le n\}$ of size $n{\cdot}k$ and sort it by bucketsort according to the second component and then according to the first. Then the j-th bucket contains all characters which appear in the j-th position in sorted order. The cost of the first pass is $O(n \cdot k + m)$, the cost of the second pass is $O(n \cdot k + k)$. Total cost is thus $O(nk + m)$.

In order to extend to words of arbitrary length let $x^i = x_1^i x_2^i \dots x_{l_i}^i$, $1 \le i \le n$; l_i is the length of x_i. We basically proceed as above, however we have to make sure that x^i has to be considered for the first time when we sort according to the l_i-th letter. This leads to the following algorithm.

Let $L = \sum_{i=1}^{n} l_i \ge n$.

1. Determine the length of x^i, $1 \le i \le n$, and create pairs (l_i, pointer to x^i).
2. Sort the pairs (l_i, pointer to x^i) by bucketsort according to the first component. Then the k-th bucket contains all words x^i with $l_i = k$, i.e. all these strings are contained in a linear list. Call this list length $[k]$ $(1 \le k \le \max(l_1, \dots l_n))$.
3. Create L pairs $(j,\ x_j^i)$, $1 \le i \le n$, $1 \le j \le l_i$ and sort them according to the second and then according to the first component. Let Nonempty $[j]$, $1 \le j \le l_{\max} = \max(l_1, \dots l_n)$ be the j-th bucket after the second pass.

Nonempty $[j]$ contains all letters which appear in the j-th position in sorted order. Delete all duplicates from Nonempty $[j]$.

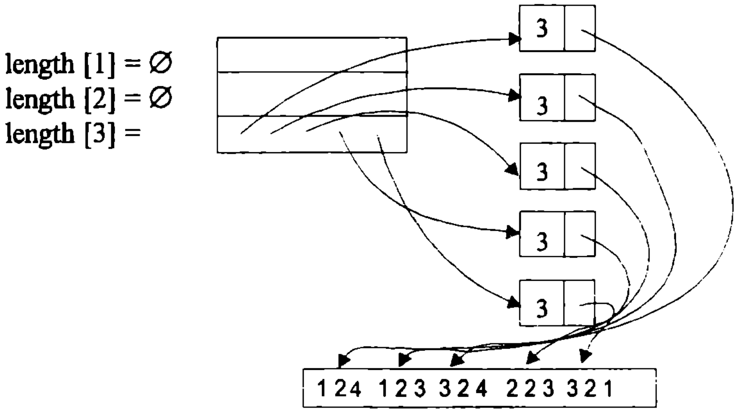
4. We finally sort words x^i by distribution. All lists in the following program are used as queues; the head of each list contains a pointer to the last element. Also x is a string variable and x_j is the j-th letter of string x.

```
(1) $W \leftarrow$ empty queue;
(2) for k from 1 to m do $S[k] \leftarrow$ empty queue od;
(3) for j from $l_{\max}$ to 1
(4) do add length $[j]$ to the front of $W$ and call the new queue $W$;
(5) while $W \neq \varnothing$
(6) do let $x$ be the first element of $W$; delete $x$ from $W$;
(7) add $x$ to the end of $S[x_j]$;
```

```
        od
    (8) while Nonempty [$j$] $\neq \varnothing$
    (9) do let $k$ be the first element of Nonempty [$j$];
    (10) delete $k$ from Nonempty [$j$];
    (11) add $S[k]$ to the end of $W$;
    (12) set $S[k]$ to the empty queue;
        od
    od
```

By a careful analysis of the cost of each line of this algorithm we deduce that bucketsort sorts n words of total length L over an alphabet of size m in time $O(m + L)$.

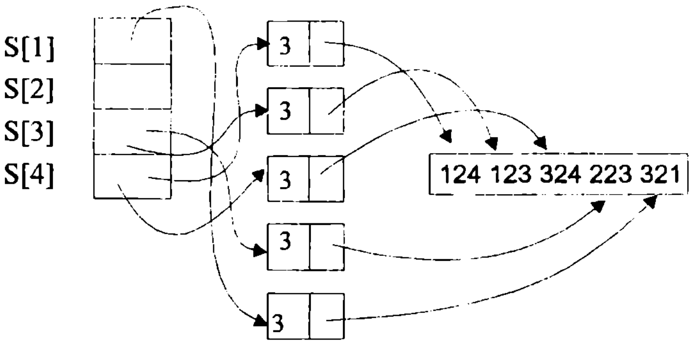
For our example, before step 4) we have:



and Nonempty is:

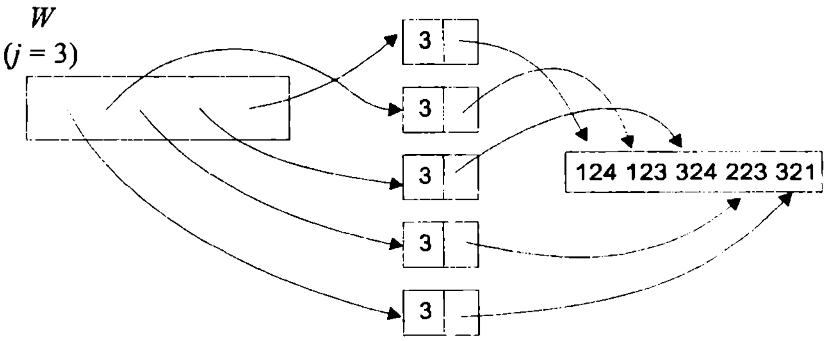
1	1, 2, 3	NONEMPTY (1)
2	2	NONEMPTY (2)
3	1, 3, 4	NONEMPTY (3)

After the first pass we have (now j = 3):

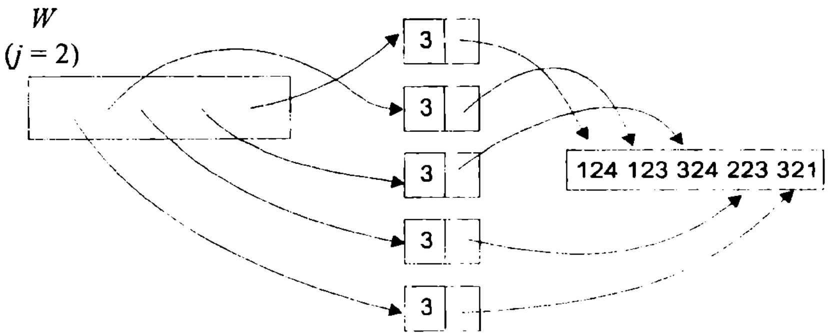


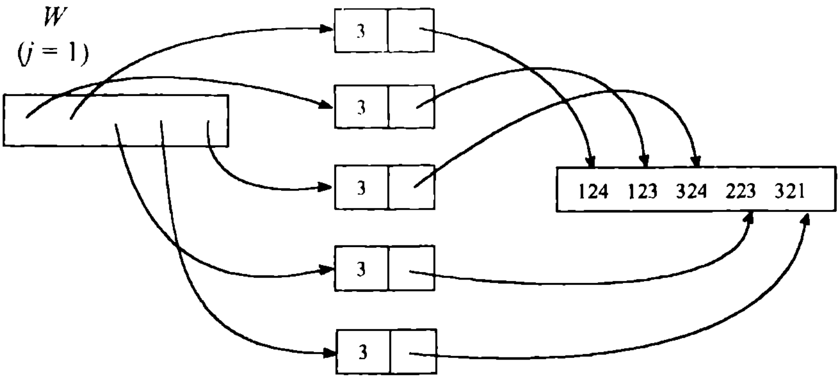
(by performing steps(1) - (6)). Now the content of *W* coincides to that of length [3].

After steps (8) - (12) the content of *W* will be:



and so on (for *j* = 2 and *j* = 1, respectively):





2. *Sorting reals by distribution.* We briefly describe distribution sort applied to real numbers. We assume that we are given a sequence x_i, $1 \le i \le n$, of reals from the interval $(0,1]$. We use the following simple algorithm, called Hybridsort.

α is some fixed real and k is equal to $\lceil \alpha n \rceil$.

1. Create k empty buckets. Put x_i into bucket $\lceil kx_i \rceil$ for $1 \le i \le n$.

2. Apply Heapsort to every bucket and concatenate the buckets.

The correctness of this algorithm is obvious.

Theorem. a) *Worst case running time of Hybridsort is* $O(n \log n)$.

b) *If the* x_i*' s are drawn independently from a uniform distribution over the interval* $(0, 1]$, *then Hybridsort has expected running time* $O(n)$.

Proof: a) Running time of the first phase is clearly $O(n)$. Let us assume that t_i elements end up in the i-th bucket, $1 \le i \le k$. Then the cost of the second phase is $O\left(\sum_i t_i \log t_i\right)$, where by definition $0 \log 0 = 0$ and $\sum_i t_i = n$.

But $\sum_i t_i \log t_i \le \sum_i t_i \log n = n \log n$.

b) Let B_i be the random variable representing the number of elements in the i-th bucket after pass 1. Then $P(B_i = h) = \dfrac{\binom{n}{h}(k-1)^{n-h}}{k^n} =$

$= \binom{n}{h}\left(\frac{1}{k}\right)^h\left(1-\frac{1}{k}\right)^{n-h}$ since any single x_j is in the i-th bucket with probability $1/k$.

Expected running time of phase 2 is

$$E\left(\sum_{i=1}^{k} B_i \log B_i\right) = k\sum_{h=2}^{n} h\log h\binom{n}{h}\left(\frac{1}{k}\right)^h\left(1-\frac{1}{k}\right)^{n-h} \le k\sum_{h=2}^{n} h^2\binom{n}{h}\left(\frac{1}{k}\right)^h\left(1-\frac{1}{k}\right)^{n-h}.$$

But $h^2\binom{n}{h} = (h(h-1)+h)\binom{n}{h} = n(n-1)\binom{n-2}{h-2} + n\binom{n-1}{h-1}$ and

$$\sum_{h=2}^{n} \frac{n(n-1)}{k^2}\binom{n-2}{h-2}\left(\frac{1}{k}\right)^{h-2}\left(1-\frac{1}{k}\right)^{n-h} = \frac{n(n-1)}{k^2}\sum_{h=2}^{n}\binom{n-2}{h-2}\left(\frac{1}{k}\right)^{h-2}\left(1-\frac{1}{k}\right)^{n-h} =$$

$$= \frac{n(n-1)}{k^2};\sum_{h=1}^{n}\frac{n}{k}\binom{n-1}{h-1}\left(\frac{1}{k}\right)^{h-1}\left(1-\frac{1}{k}\right)^{n-h} = \frac{n}{k}.$$

It follows that $E\left(\sum_{i=1}^{k} B_i \log B_i\right) \le k\left(\frac{n(n-1)}{k^2} + \frac{n}{k}\right) = O(n)$ since $k = \lceil \alpha n \rceil$. □

§ 8. The linear median algorithm

Selection is a problem which is related to but simpler than sorting. We are given a sequence $S_1, \ldots, S_n$ of pairwise distinct elements and an integer i, $1 \le i \le n$, and want to find the i-th smallest element of the sequence, i.e. an S_j such that there are $i-1$ keys S_l with $S_l < S_j$ and $n-i$ keys S_l with $S_l > S_j$. For $i = \lfloor n/2 \rfloor$ such a key is called median. Of course, selection can be reduced to sorting. We might first sort sequence $S_1, \ldots, S_n$ and then find the i-th smallest element by a single scan of the sorted sequence. This results in an $O(n \log n)$ algorithm.

However, there is a linear time solution. We describe a simple, linear expected time solution (procedure Find) first and then extend it to a linear worst case time solution (procedure Select).

Procedure Find is based on the partitioning algorithm used in Quicksort. We choose some element of the sequence, say S_1, as partitioning element and then divide the sequence into the elements smaller than S_1 and the elements larger than S_1. We then call Find recursively on the appropriate subsequence.

```
(1) procedure Find ($M$, $i$);
        co finds the $i$-th smallest element of set $M$;
      begin
(2) $S \leftarrow$ some element of $M$;
(3) $M_1 \leftarrow \{m \in M;\ m < S\}$;
(4) $M_2 \leftarrow \{m \in M;\ m > S\}$;
(5) case      $|M_1|$ of
(6)           $< i - 1$: Find ($M_2$, $i - |M_1| - 1$);
(7)           $= i - 1$: return $S$;
(8)           $> i - 1$: Find ($M_1$, $i$)
(9) esac
      end
```

When set M is stored in an array then lines (2) - (4) of Find are best implemented by lines (2) - (6) of Quicksort. Then a call of Find has cost $O(|M|$ + the time spent in the recursive call). The worst case running time of Find is clearly $O(n^2)$ (consider the case $i = 1$ and $|M_1| = |M| - 1$ always). Expected running time of algorithm Find is linear as we show next. We use the same randomness assumption as for the analysis of Quicksort, namely the elements of M are pairwise distinct and each permutation of the elements of M is equally likely. In particular, this implies that element S chosen in line (2) is the k-th largest element of M with probability $1/|M|$. It also implies that both subproblems M_1 and M_2 again satisfy the randomness assumption (cf. the analysis of Quicksort). Let $T(n, i)$ be the expected runnig time of Find (M, i) where $|M| = n$ and let $T(n) = \max\limits_i T(n, i)$. We have $T(1) = 0$ and

$$T(n,i) \le cn + \frac{1}{n}\left[\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k-1, i)\right]$$

for some constant c, since the partitioning process takes time cn and the recursive call takes expected time $T(n - k, i - k)$ if $k = |M_1| + 1 < i$ and time $T(k - 1, i)$ if $k = |M_1| + 1 > i$.

Thus

$$T(n) \le cn + \frac{1}{n} \max_i \left[\sum_{k=1}^{i-1} T(n-k) + \sum_{k=i}^{n-1} T(k) \right]$$

We show $T(n) \le 4\,cn$ by induction on n. This is true for $n = 1$. For $n > 1$ we have

$$T(n) \le cn + \frac{1}{n} \max_i \left[\sum_{k=n-i+1}^{n-1} 4ck + \sum_{k=i}^{n-1} 4ck \right] \le cn +$$
$$+ \frac{4c}{n} \max_i [n(n-1) - (n-i)(n-i+1)/2 - i(i-1)/2] \le 4cn,$$

since the expression in square brackets is maximal for $i = \dfrac{n+1}{2}$ (notice that the expression is symmetric in i and $n - i + 1$) and then has value $n(n-1) - n(n-1)/4 \le \dfrac{3n^2}{4}$. We have thus shown:

Theorem 1. *Algorithm Find has linear expected running time.*

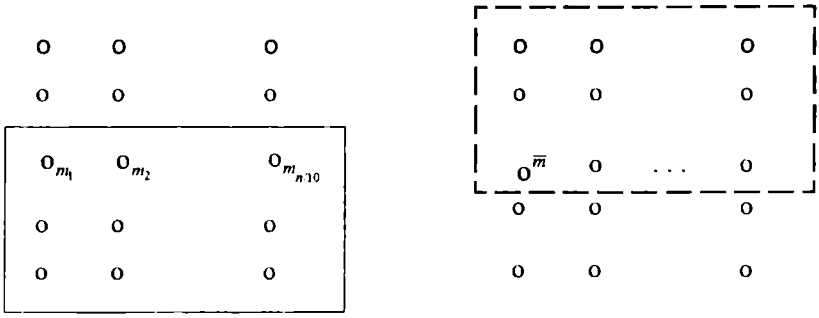
The expected linearity of Find stems from the fact that the expected size of the subproblem to be solved is only a fraction of the size of the original problem. However, the worst case running time of Find is quadratic because the size of the subproblem might be only one smaller than the size of the original problem. If one wants a linear worst case algorithm one has to choose the partitioning element more carefully. A first approach is to take a reasonable size sample of M, say of size $|M|/5$, and to take the median of the sample as partitioning element.

However, this idea is not good enough yet because the sample might consist of small elements only. A better way of choosing the sample is to divide M into small groups of say 5 elements each and to make the sample the set of medians of the groups. Then it is guaranteed that a fair fraction of elements are smaller (larger) than the partitioning element.

This leads to the following algorithm.

```
(1) procedure Select (M, i);
    co finds the i-th smallest element of set M;
    begin
(2) $n \leftarrow |M|$;
(3) if $n \le 100$ then sort M and find the i-th smallest element directly
    else
(4) divide M in $\lceil n/5 \rceil$ subsets $M_1, \ldots, M_{\lceil n/5 \rceil}$ of 5 elements each (the
    last subset may contain less that 5 elements);
(5) sort $M_j$; $1 \le j \le \lceil n/5 \rceil$;
(6) let $m_j$ be the median of $M_j$;
(7) call Select ($\{m_1, \ldots m_{\lceil n/5 \rceil}\}$, $\lceil n/10 \rceil$) and determine $\overline{m}$, the
    median of the medians;
(8) let $M_1 = \{m \in M; m < \overline{m}\}$ and $M_2 = \{m \in M; \overline{m} \le m\}$;
(9) if $i \le |M_1|$
(10) then Select ($M_1$, i)
(11) else Select ($M_2$, $i - |M_1|$)
    fi
    fi
    end
```

It is very helpful to illustrate algorithm Select pictorially. In line (4) M is divided into groups of 5 elements each and the median of each group is determined in lines (5) and (6).



At this point we have $n/5$ linear orders of 5 elements each. Next we find $\overline{m}$, the median of the medians. Assume w.l.o.g. that $m_1, \ldots, m_{n/10} < \overline{m}$ and $\overline{m} \le m_{n/10+1}, \ldots, m_{n/5}$. In the diagram representing this situation each of the groups is represented by a vertical line of 5 elements, largest element at the top. Note that all elements in the solid rectangle are smaller than $\overline{m}$ and hence belong to M_1 and that all points in the dashed rectangle are at least as large as $\overline{m}$ and hence belong to M_2. Each rectangle contains $3n/10$ points, provided that 10 divides n.

Lemma 1. *We have* $|M_1|, |M_2| \leq 8n/11$.

Proof: Note that from the discussion above we have $|M_1| + |M_2| = n$ and $|M_1|, |M_2| \geq \frac{3n}{10}$ if 10 divides n. If 10 does not divide n then $|M_1|, |M_2| \geq \frac{3n}{11}$ for $n \geq 100$. Indeed, if 10 does not divide n then there is a number m, $1 \leq m \leq 9$, such that $10|n-m$ and in this case

$$|M_1|,|M_2| \geq \frac{3(n-m)}{10} \geq \frac{3(n-9)}{10} \geq \frac{3n}{11}$$

since the last inequality is equivalent to $n \geq 99$. It follows that $|M_1| \leq n - \frac{3n}{11} = \frac{8n}{11}$ and a similar inequality holds for $|M_2|$. □

Let $T(n)$ be the maximal running time of algorithm Select on any set M of n elements and any i.

Lemma 2. *There are constants a, b such that*

$T(n) \leq an$, *for* $n \leq 100$

$T(n) \leq T(21n/100) + T(8n/11) + bn$, *for* $n \geq 100$.

Proof: The claim is obvious for $n \leq 100$. So let us assume $n \geq 100$. Select is called twice within the body of Select, once for a set of $\lceil n/5 \rceil \leq \frac{n+4}{5} \leq \frac{21n}{100}$ elements and once for a set of size at most $8n/11$. Furthermore, the total cost of Select outside recursive calls is clearly $O(n)$. □

Theorem 2. *Algorithm Select works in linear time.*

Proof: We show $T(n) \leq cn$ where $c = \max(a, 1100\, b/69)$ by induction on n. For $n \leq 100$ there is nothing to show. For $n > 100$ we have

$$T(n) \leq T(21n/100) + T(8n/11) + bn$$
$$\leq c\, 21\, n/100 + c\, 8\, n/11 + bn \leq cn,$$

by definition of c. □

Chapter 3

SEARCHING TECHNIQUES

We are concerned with the process of collecting information in a computer's memory, and with the subsequent recovery of that information as quickly as possible. Sometimes we are confronted with more data than we can really use, and it may be wisest to forget and to destroy most of it; but at other times it is important to retain and organize the given facts in such a way that fast retrieval is possible.

Most of this paragraph is devoted to the study of a very simple search problem: how to find the data that has been stored with a given identification. For example, in a numerical application we might want to find $f(x)$, given x and a table of the values of f; in a nonnumerical application, we might want to find the English translation of a given Romanian word.

In general, we shall suppose that a set of N records has been stored, and the problem is to locate the appropriate one. As in the case of sorting, we assume that each record includes a special field called its key.

We generally require the N keys to be distinct, so that each key uniquely identifies its record. The collection of all records is called a table or a file. A large group of files is frequently called a data base.

Algorithms for searching are presented with a so-called argument, K, and the problem is to find which record has K as its key. After the search is complete, two possibilities can arise: Either the search was successful, having located the unique record containing K, or it was unsuccessful, having determined that is nowhere to be found. After an unsuccessful search it is sometimes desirable to enter a new record, containing K, into the table; a method which does this is called a "search and insertion" algorithm.

Although the goal of searching is to find the information stored in the record associated with K. the algorithms in this section generally ignore everything but the keys themselves. Searching is the most time - consuming part of many programs, and the substitution of a good search method for a bad one often leads to a substantial increase in speed.

Sometimes it is possible to arrange the data structure so that searching is eliminated entirely, i.e., so that we always know just where to find the information we need. For example, if we are allowed to choose the keys freely, we might as well let them be the numbers $\{1, 2, \ldots, N\}$; then the record containing K can simply be placed in location TABLE + K.

However, there are many cases when a search is necessary, so it is important to have efficient algorithms for searching.

Search methods might be classified in several ways; for example we might divide search methods into static vs. dynamic searching, where ''static'' means that the contents of the table are essentially unchanging (so that it is important to minimize the search time without regard for the time required to set up the table), and ''dynamic'' means that the table is subject to frequent insertions (and perhaps also deletions). We might also divide searching into those methods which use the actual keys and those which work with transformed keys (by some hashing methods).

There is a certain amount of interaction between searching and sorting, as we shall see later. A number of interesting new search procedures based on tree structures were introduced, and research about searching is still actively continuing at the present time.

§ 1. Sequential searching

Algorithm S (Sequential search). Given a table of records R_1, R_2, ..., R_N, whose respective keys are $K_1, K_2, \ldots, K_N$, this algorithm searches for a given K. We assume that $N \geq 1$.

```
1. Set $i \leftarrow 1$
2. If $K = K_i$, the algorithm terminates successfully.
3. $i \leftarrow i + 1$
4. If $i \leq N$ go to 2. Otherwise the algorithm terminates unsuccessfully.
```

Note that this algorithm can terminate in two different ways, successfully (having located the desired key) or unsuccessfully (having established that the given argument is not present in the table). The same will be true of most other algorithms in this chapter.

If every input key occurs with equal probability, the average value of C, the number of key comparisons, in a successful search will be $\frac{1+2+\ldots+N}{N}=\frac{N+1}{2}$. A straightforward change makes the algorithm faster, unless the list of records is quite short:

Algorithm Q (Quick sequential search). This algorithm is the same as Algorithm S, except that it assumes the presence of a "dummy" record R_{N+1} at the end of the file.

```
1. Set $i \leftarrow 1$, and set $K_{N+1} \leftarrow K$.
2. If $K = K_i$, go to 4.
3. $i \leftarrow i + 1$ and return to 2.
4. If $i \leq N$, the algorithm terminates successfully; otherwise it terminates unsuccessfully ($i = N + 1$).
```

A slight variation of the algorithm is appropriate if we know that the keys are in increasing order:

Algorithm T (Sequential search in ordered table). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \ldots < K_N$, this algorithm searches for a given argument K. For convenience and speed, the algorithm assumes that there is a dummy record K_{N+1} whose key value is $K_{N+1} = \infty > K$.

```
1. Set $i \leftarrow 1$.
2. If $K \leq K_i$, go to 4.
3. $i \leftarrow i + 1$ and return to 2.
4. If $K = K_i$, the algorithm terminates successfully. Otherwise it terminates unsuccessfully.
```

If all input keys are equally likely, this algorithm takes essentially the same average time as Algorithm **Q**, for a successful search. But unsuccessful searches are performed about twice as fast. The same search procedures can be used for tables which have a linked representation since the data are traversed sequentially. Now suppose that key K_i will occur with probability p_i, where $p_1 + p_2 + \ldots + p_N = 1$. The time required to do a successful search is essentially proportional to the number of comparisons, C, which now has the average value

$$\overline{C}_N = p_1 + 2p_2 + \ldots + Np_N$$

If we have the option of putting the records into the table in any desired order this quantity $\overline{C}_N$ is smallest when $p_1 \geq p_2 \geq \ldots \geq p_N$, i.e., when the most frequently used records appear near the beginning. If $p_1 = p_2 = \ldots = p_N = \frac{1}{N}$, this formula reduces to $\overline{C}_N = \frac{N+1}{2}$. A more typical distribution is ‘’Zipf’s law”, $p_1 = \frac{c}{1}, p_2 = \frac{c}{2}, \ldots, p_N = \frac{c}{N}$, where $c = \frac{1}{H_N}$. This distribution was formulated by G. Zipf, who observed that the n-th most common word in natural language text seems to occur with a frequency inversely proportional to n. If Zipf’s law governs the frequency of the keys in a table, we have immediately $\overline{C}_N = \frac{N}{H_N}$; searching such a file is about $\frac{1}{2}$ ln N times as fast as searching file with randomly-ordered records.

The above calculations with probabilities are nice, but in most cases we do not know the probabilities are. In the next section we discuss self-organizing linear lists.

§ 2. Self-organizing linear lists

The idea of self–organization is quite simple. Suppose that we have a data-structure for set $S = \{x_1,\ldots,x_n\}$. Whenever we access an element of S, say x_i, then we move x_i closer to the entry point of the data structure. For linear lists this means that whenever a record has been successfully located, it is moved to the beginning of the table. This procedure is readily implemented if the table is a linked linear list, especially because the record being moved to the beginning often has to be substantially updated anyway. This procedure will make subsequent accesses to x_i cheaper. In this way the elements of S compete for the "good" places in the data structure and high-frequency elements are more likely to be there. Notice however, that we do not maintain any explicite frequency counts or weights; we hope that the data structure self-organizes to a good data structure. For an average case analysis we need to have probabilities for the various operations.

The data structure then leads to a Markov chain whose states are the different configurations of the data structure. We can then use probability theory to compute the stationary probabilities of the various states and use these probabilities to derive bounds on the expected behaviour of the data structure.

Let $S = \{x_1,...,x_n\}$. We consider operations Access (x), where $x \in S$ is assumed, Insert (x), where $x \notin S$ is assumed, and Delete (x), where $x \in S$ is assumed. We always organize S as a linear list which may be realized as either an array or a linked list. We use pos(i) to denote the position of element x_i in the list. We assume that the cost of operations Access (x_i) and Delete (x_i) is pos(i) and that the cost of operation Insert (x) is $|S| + 1$.

A popular strategy is the Move-to-Front rule.

Move-to-Front Rule (MFR): Operations Access (x) and Insert (x) make x the first element of the list and leave the order of the remaining elements unchanged; operation Delete(x) removes x from the list.

Example: We give an example for the *MF* rule:

$$134 \xrightarrow{\text{Insert(2)}} 2134 \xrightarrow{\text{Access(4)}} 4213 \xrightarrow{\text{Delete(1)}} 423.$$

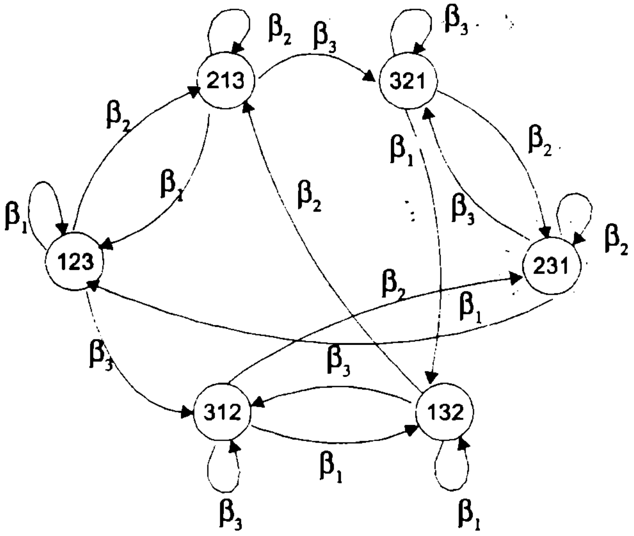
The cost of this sequence is $4 + 4 + 3 = 11$.

For the expected case analysis we consider only sequences of Access operations. If $S = \{x_1,...,x_n\}$ let β_i be the probability of an Access to element x_i, $1 \le i \le n$. We assume w.l.o.g. that $\beta_1 \ge \beta_2 \ge ... \ge \beta_n$. The frequency decreasing rule (**FDR**) arranges S as list $x_1x_2 \ldots x_n$ and has minimum expected access time $P_{\text{FDR}} = \sum_{i=1}^{n} i\beta_i$. The expected access time of the frequency decreasing rule is easily seen to be optimal. Consider a sequence ... x_jx_i ... in which x_j is directly in front of x_i and $j > i$. Exchanging this pair of elements changes the expected access time by $\beta_j - \beta_i$, a non-positive amount. This shows that the expected access time of the **FD**– rule is optimal.

For the analysis of the **MF** rule we use Markov chains. The following diagram illustrates the move-to-front rule for a set of three elements, $\{1, 2, 3\}$.

The diagram has 3! = 6 states.

In this diagram 123 denotes linear list $x_1x_2x_3$. If x_3 is accessed (probability β_3) then list $x_1x_2x_3$ is changed into list $x_3x_1x_2$.



The move-to-front rule for n elements induce a Markov chain with $n!$ states. The states correspond to the $n!$ linear arrangements of set S, i.e. permutation π represents a linear list where x_i is at position $\pi(i)$. Furthermore, there is a transition from state π to state ρ if there is an i such that $\rho(i) = 1$, $\rho(j) = \pi(j) + 1$ for j such that $\pi(j) < \pi(i)$ and $\rho(j) = \pi(j)$ otherwise. This transition has probability β_i. This Markov chain is irreducible (the corresponding digraph is strongly connected) and aperiodic and in these conditions stationary probabilities γ_π exist. γ_π is the asymptotic (limiting) probability that the chain is in state π, i.e. that x_i is at position $\pi(i)$ for $1 \le i \le n$.

Then

$$P_{MFR} = \sum_{\pi} \gamma_\pi \sum_{i} \beta_i \pi(i)$$

is the asymptotic expected search time under the move-to-front rule.

Lemma 1. Let $\delta(j, i)$ *be the asymptotic probability that* x_j *is in front of* x_i*. Then*

a) $P_{\text{MFR}} = \sum_i \beta_i \left(1 + \sum_{j \neq i} \delta(j,i)\right)$

b) $\delta(j,i) = \dfrac{\beta_j}{\beta_i + \beta_j}$.

Proof: a) Let $p_i = \sum_\pi \gamma_\pi \pi(i)$. Then $P_{\text{MFR}} = \sum_i \beta_i p_i$ and p_i is the expected position of element x_i.

Furthermore,

$$p_i = \sum_\pi \gamma_\pi \pi(i) = \sum_\pi \gamma_\pi \left(1 + |\{j: \pi(j) < \pi(i)\}|\right) = 1 + \sum_{j \neq i} \sum \{\gamma_\pi : \pi(j) < \pi(i)\} =$$

$$= 1 + \sum_{j \neq i} \delta(j,i)$$

since $\Sigma\{\gamma_\pi : \pi(j) < \pi(i)\}$ is the asymptotic probability that x_j is in front of x_i.

b) x_j is in front of x_i asymptotically if there is a k such that the last k+1 accesses are: an access to x_j followed by k accesses to elements different from x_j and x_i. Hence

$$\delta(j,i) = \beta_j \sum_{k \geq 0} \left(1 - (\beta_i + \beta_j)\right)^k = \beta_j \frac{1}{1 - (1 - (\beta_i + \beta_j))} = \frac{\beta_j}{\beta_i + \beta_j}.$$

Theorem 1. *Let* $\beta_1 \geq \beta_2 \geq \ldots \geq \beta_n$. *Then*

a) $P_{\text{MFR}} = 1 + 2 \sum_{1 \leq j < i \leq n} \dfrac{\beta_i \beta_j}{\beta_i + \beta_j}$

b) $P_{\text{MFR}} \leq 2P_{\text{FDR}} - 1$

Proof: a) We have by lemma 1,

$$P_{\mathbf{MFR}} = \sum_i \beta_i \left(1 + \sum_{j \neq i} \delta(j,i)\right) = \sum_i \beta_i \left(1 + \sum_{j \neq i} \frac{\beta_j}{\beta_i + \beta_j}\right) =$$

$$= 1 + \sum_i \sum_{j \neq i} \frac{\beta_i \beta_j}{\beta_i + \beta_j} = 1 + 2\sum_{j<i} \frac{\beta_i \beta_j}{\beta_i + \beta_j}.$$

b) Since $\frac{\beta_j}{\beta_i + \beta_j} \leq 1$ we deduce from part a)

$$P_{\mathbf{MFR}} \leq 1 + 2\sum_i \beta_i (i-1) = 2P_{\mathbf{FDR}} - 1 \quad \square$$

This theorem gives a closed form expression for P_{MFR} which is readily evaluated for particular distributions β and usually shows that the expected cost of the **MF** – rule is only a few percent above the optimum. It also shows that P_{MFR} is never more than twice the optimum.

If $p_i = \frac{1}{N}$ for $1 \leq i \leq N$, the self – organizing table is always in completely random order, and this formula reduces to $P_{\mathrm{MFR}} = \frac{N+1}{2}$ derived above. When the key probabilities obey Zipf's law one can prove easy, using harmonic numbers, that $P_{\mathrm{MFR}} \sim 2N/\log_2 N$. This is substantially better than $\frac{1}{2}N$, when N is reasonably large, and it is only about $\ln 4 \approx 1.386$ times as many comparisons as would be obtained in the optimum arrangement (P_{FDR}).

Computational experiments involving actual compiler symbol tables indicate that the self-organizing method works even better than the above formulas predict, because successive searches are not independent (small groups of keys tend to occur in bunches) (cf. D. Knuth).

Tape searching with unequal-length records. Suppose the table we are searching is stored on tape, and the individual records have varying lengths. Let L_i be the length of record R_i, and let p_i be the probability that this record will be sought. The running time of the search method will now be approximately proportional to

$$p_1L_1 + p_2(L_1 + L_2) + \ldots + p_N(L_1 + L_2 + \ldots + L_N).$$

The optimum arrangement of programs on a library tape may be determined as follows.

Theorem 2. *Let L_i and p_i be as defined above. The arrangement of records in the table is optimal if and only if*

$$\frac{p_1}{L_1} \geq \frac{p_2}{L_2} \geq \ldots \geq \frac{p_N}{L_N} \qquad (1)$$

Proof: Suppose that R_i and R_{i+1} are interchanged on the tape; the cost changes from

$$\ldots + p_i(L_1 + \ldots + L_{i-1} + L_i) + p_{i+1}(L_1 + \ldots + L_{i+1}) + \ldots$$

to $\ldots + p_{i+1}(L_1 + \ldots + L_{i-1} + L_{i+1}) + p_i(L_1 + \ldots + L_{i+1}) + \ldots$ with a difference of

$p_iL_{i+1} - p_{i+1}L_i$. Therefore if $\frac{p_i}{L_i} < \frac{p_{i+1}}{L_{i+1}}$ the given arrangement is not optimal. It follows that if the arrangement is optimal then (1) holds.

Conversely, assume that (1) holds; we need to prove that the arrangement is optimal. We know that any permutation of the records can be sorted into the order $R_1R_2 \ldots R_N$ by using a sequence of interchanges of adjacent records. Each of these interchanges replaces $\ldots R_jR_i \ldots$ by $\ldots R_iR_j \ldots$ for some $i < j$, so it decreases the search time by the non-negative amount $p_iL_j - p_jL_i$.

If the initial arrangement was optimum, it follows that all arrangements obtained in this way must be optimum also; therefore the order $R_1R_2 \ldots R_N$ must have minimum search time.

§ 3. Searching by comparison of keys

In this section we shall discuss search methods which are based on a lincar ordering of the keys. After comparing the given argument K to a key K_i in the table, the search continues in three different ways, depending on whether $K < K_i$, $K = K_i$, or $K > K_i$.

With so many sorting methods at our disposal, we will have little difficulty rearranging a file into order so that it may be searched conveniently. Of coursc, if wc only need to search the table once, it is faster to do a sequential search than to do a complete sort of the file; but if we need to make repeated searches in the same file, we are better off having it in order. Therefore in this section we shall concentrate on methods which are appropriate for searching a table whose keys are in order, $K_1 < K_2 < \dots < K_N$, making random accesses to the table entries. After comparing K to K_i in such a table, we either have: * $K < K_i$ [R_i, $R_{i+1}, \dots, R_N$ are eliminated from consideration]; or * $K = K_i$ [the search is done]; or * $K > K_i$ [$R_1, R_2, \dots, R_i$ are eliminated from consideration].

In each of these three cases, substantial progress has been made, unless is near one of the ends of the table; this is why the ordering leads to an efficient algorithm.

Let $S = \{K_1 < K_2 < \dots < K_n\}$ be stored in array $K[1 \dots n]$, i.e. $K[i] = K_i$, and let $a \in U$. In order to decide $a \in S$, we compare a with some table element and then proceed with either the lower or the upper part of the table.

This idea leads to the following general algorithm from which various algorithms can be obtained by replacing lines (2) and (7) by specific strategies for choosing next:

```
(1) low ← 1; high ← n;
(2) next ← an integer in [low, high];
(3) while a ≠ K [next] and high > low
(4) do if a < K [next]
(5)     then high ← next −1
(6)     else low ← next + 1 fi;
(7)     next ← an integer in [low, high]
(8) od;
(9) if  a = K [next] then „successful" else „unsuccessful".
```

Linear search is obtained by next ← low; binary search by next ← $\lfloor (\text{low} + \text{high})/2 \rfloor$ (or next ← $\lceil (\text{low} + \text{high})/2) \rceil$) and interpolation search by:

$$\text{next} \leftarrow \text{low} - 1 + \left\lceil \frac{a - K[low-1]}{K[high+1] - K[low-1]}(high - low + 1) \right\rceil.$$

We discuss binary search in greater detail below.

The correctness of the program is independent of the particular choice made in line (2) and (7). If $a = K$ [next] then the search is successful. Suppose now that $a \neq K$ [next]. We know that $a \in S$ implies $a \in \{K[\text{low}], \dots, K[\text{high}]\}$. If high < low then certainly $a \notin S$. If high = low then next = high and hence $a \notin S$. In either case the search is unsuccessful.

Finally, the program terminates because high – low is decreased by at least one in each execution of the loop body.

Note that for interpolation search it is assumed that positions $K[0]$ and $K[n+1]$ are added and filled with artificial elements. The worst case complexity of interpolation search is clearly $O(n)$; to see this consider the case that $K[0] = 0$, $K[n+1] = 1$, $a = \frac{1}{n+1}$ and $S \subseteq (0,a)$. Then next = low always and interpolation search deteriorates to linear search. Average case behaviour is much better. We shall rewrite the binary search algorithm as follows:

Algorithm B (Binary search). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \ldots < K_N$, this algorithm searches for a given argument K.

1. Set $l \leftarrow 1$, $u \leftarrow N$.
2. If $u < l$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow \lfloor (l + u)/2 \rfloor$, the approximate midpoint of the relevant table area.
3. If $K < K_i$, go to **4**; if $K > K_i$, go to **5**; and if $K = K_i$, the algorithm terminates successfully.
4. Set $u \leftarrow i - 1$ and return to **2**.
5. Set $l \leftarrow i + 1$ and return to **2**.

Note that if $u < l$ at step **2**, then $u = l - 1$ in all cases.

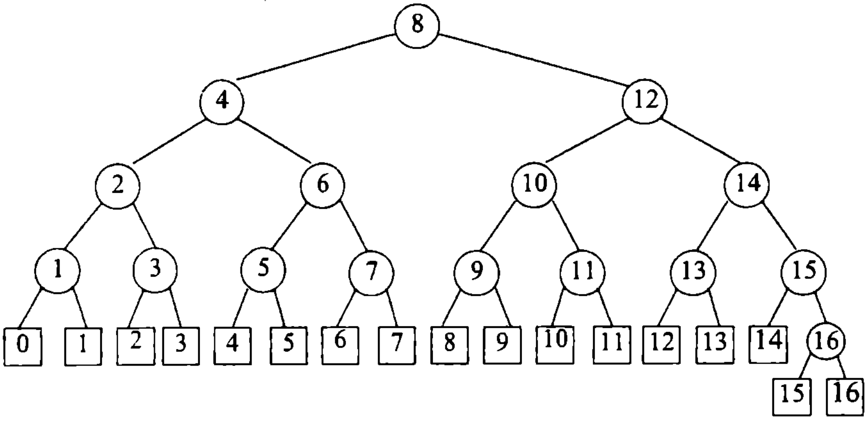


Fig. 1 A binary tree which corresponds to binary search when N = 16.

In order to understand what is happening in Algorithm **B**, it is best to think of it as a binary decision tree, as shown in fig. 1 for the case $N = 16$.

When $N = 16$, the first comparison made by the algorithm is $K : K_8$; this is represented by the root node $\textcircled{8}$ in the figure. Then if $K < K_8$, the algorithm follows the left subtree, comparing K to K_4; similarly if $K > K_8$, the right subtree is used. An unsuccessful search will lead to one of the „external" square nodes numbered 0 through N; for example, we reach node $\boxed{6}$ if and only if $K_6 < K < K_7$.

The binary tree corresponding to a binary search on N records can be constructed as follows:

If $N = 0$, the tree is simply $\boxed{0}$. Otherwise the root node is $\textcircled{\lceil N/2 \rceil}$, the left subtree is the corresponding binary tree with $\lceil N/2 \rceil - 1$ vertices, and the right subtree is the corresponding binary tree with $\lfloor N/2 \rfloor$ vertices and with all vertex numbers increased by $\lceil N/2 \rceil$. Here we have counted only internal vertices corresponding to a successful search.

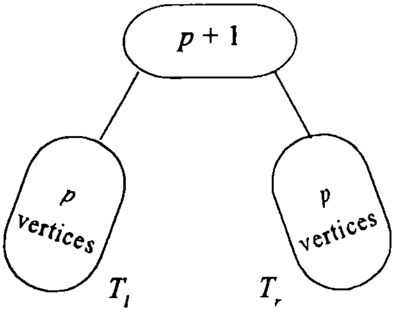
In an analogous fashion, any algorithm for searching an ordered table of length N by means of nonredundant comparisons can be represented as a binary tree in which the internal vertices are labelled with the numbers 1 to N and external vertices with 0 to N.

Conversely, any binary tree corresponds to a valid method for searching an ordered table; we simply label the vertices $\boxed{0}$ $\textcircled{1}$ $\boxed{1}$ $\textcircled{2}$ $\boxed{2}$ ... $\boxed{N-1}$ $\textcircled{N}$ $\boxed{N}$ in symmetric order, from left to right, if the tree has N internal vertices and consequently, $N + 1$ external ones.

If the search argument input to Algorithm **B** is K_{10}, the algorithm makes comparisons $K > K_8$, $K < K_{12}$, $K = K_{10}$. This corresponds to the path from the root $\textcircled{8}$ to vertex $\textcircled{10}$ in fig. 1. Similarly, the behaviour of Algorithm **B** on other keys corresponds to the other paths leading from the root of the tree. The method of constructing the binary trees corresponding to Algorithm **B** therefore makes it easy to prove the following result by induction on N:

Theorem 1. *If $2^{k-1} \leq N < 2^k$, a successful search using Algorithm* **B** *requires at most k comparisons. If $N = 2^k - 1$, an unsuccessful search requires k comparisons; and if $2^{k-1} \leq N < 2^k - 1$, an unsuccessful search requires either $k - 1$ or k comparisons. This means that for $N = 2^k - 1$ all terminal vertices of the binary tree associated to Algorithm* **B** *are on level k and for $2^{k-1} \leq N < 2^k - 1$ terminal vertices are on levels $k - 1$ and k.*

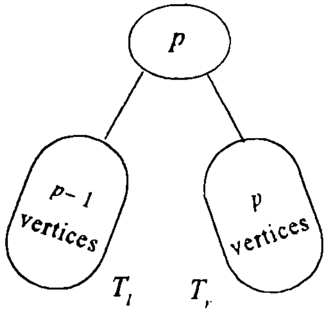
Proof: The property is true for $k = 1$. Let $k \geq 2$ and suppose that the property is true for all $2^{k-1} \leq N < 2^k$. If $2^k \leq N < 2^{k+1}$, we shall distinguish two cases: A. N is odd, $N = 2p + 1$; *B*. N is even, $N = 2p$, where $p \in \mathbf{N}, p \geq 1$.



A. In this case the root of the binary tree T corresponding to Algorithm **B** has the label $p + 1 = \left\lfloor \frac{N+1}{2} \right\rfloor$; the left subtree T_l and the right subtree T_r contain each exactly p vertices. Since

$$2^k \leq 2p + 1 < 2^{k+1}$$

it follows that $p \geq 2^{k-1} - \frac{1}{2}$, hence $p \geq 2^{k-1}$, since p is an integer and also $p < 2^k - \frac{1}{2}$, , hence $p < 2^k$. Because $2^{k-1} \leq p < 2^k$ we can apply the induction hypothesis for both T_l and T_r: $h(T_l) = h(T_r) = k$, hence $h(T) = k + 1$, the height of T being equal to the maximum number of key comparisons performed by Algorithm **B** for a successful search. If $p = 2^k - 1$ all terminal vertices of T_l and T_r are on level k, hence if $N = 2p + 1 = 2^{k+1} - 1$ all terminal vertices of T are on level $k + 1$. If $2^{k-1} \leq p < 2^k - 1$, which implies that $2^k \leq N < 2^{k+1} - 1$ since $N = 2p + 1$ is odd, both T_l and T_r have terminal vertices on levels $k - 1$ and k, hence T has all terminal vertices on two consecutive levels, k and $k + 1$.



B. In this case the root of the binary tree has label $p = \left\lfloor \frac{N+1}{2} \right\rfloor$ and T_l and T_r have not equal sizes: T_l has $p - 1$ vertices and T_r p vertices. Since $2^k \leq 2p < 2^{k+1}$, it follows that $2^{k-1} \leq p < 2^k$. We have also $2^{k-1} \le p - 1 < 2^k$ unless $p = 2^{k-1}$, or $N = 2^k$. But in this case the tree T is similar to the tree in fig. 1, p. 103 for every $k \ge 1$: it has $h(T) = k + 1$, $N - 1$ terminal vertices on level k and two terminal vertices on level $k + 1$, and the theorem is proved directly in this case ($N = 2^k$).

It remains to consider the case $2^k < N < 2^{k+1}$ only. By the induction hypothesis we deduce that $h(T_l) = h(T_r) = k$, hence $h(T) = k + 1$ and Algorithm **B** requires at most $k + 1$ comparisons for a successful search.

Since N is even, $N \neq 2^s - 1$ for every $s \ge 1$, hence we must prove that the terminal vertices of T are on two consecutive levels k and $k + 1$. This follows also by the induction hypothesis for trees T_l and T_r which correspond to the same algorithm applied for a set of $p - 1$ and p keys, respectively. For T_l we have $p - 1 \neq 2^k - 1$ since N is not a power of 2. It follows that terminal vertices of T_l belong to two consecutive levels $k - 1$ and k. For T_r all terminal vertices belong to level k (for $p = 2^k - 1$) or to levels $k - 1$ and k. It follows that T has its terminal vertices on two consecutive levels, k and $k + 1$ and the proof is complete. □

It follows that the number of key comparisons in a successful (or an unsuccessful) search by binary search algorithm is at most $\lfloor \log_2 N \rfloor + 1$. We have defined on p. 80 $E(T)$, the external path length of a binary tree T, as the sum of the lengths (number of edges) of the paths from the root to all external vertices of T. In a similar manner, $I(T)$, the internal path length of T is defined as the sum of the lengths of the paths from the root to all internal vertices of T.

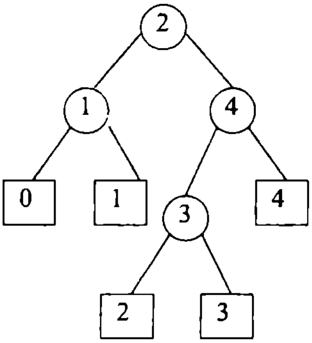


Fig. 2

For example, for tree T in fig. 2 we have $E(T) = 2 + 2 + 3 + 3 + 2 = 12$ and $I(T) = 1 + 2 + 1 = 4$; $E(T) - I(T) = 2 \cdot 4$, twice the number of internal vertices of T. This property holds for any binary tree:

Lemma 2. *For any binary tree T which is complete we have*

$$E(T) = I(T) + 2N,$$

where N denotes the number of internal vertices of T.

Proof: Suppose that a complete binary tree has a_k internal vertices and b_k external vertices at level k, for $k = 0,1...$ (the root is at level zero). Thus in fig. 2 we have $(a_0, a_1, a_2,...) = (1, 2, 1, 0, 0,...)$ and $(b_0, b_1,...) = (0, 0, 3, 2, 0, 0,...)$.

Consider the generating functions associated with these sequences:

$$A(z) = \sum_{k=0}^{\infty} a_k z^k \quad \text{and}$$

$$B(z) = \sum_{k=0}^{\infty} b_k z^k$$

where only a finite number of terms are non–vanishing. We have

$$2a_{k-1} = a_k + b_k$$

for every $k \geq 0$ since all a_{k-1} internal vertices on level $k - 1$ have exactly two sons on level k and the number of vertices on level k is equal to $a_k + b_k$. We deduce

$$A(z) + B(z) = \sum_{k=0}^{\infty}(a_k + b_k)z^k = a_0 + b_0 + \sum_{k=1}^{\infty}(a_k + b_k)z^k =$$

$$= 1 + 2\sum_{k=1}^{\infty} a_{k-1}z^k = 1 + 2z\sum_{k=1}^{\infty} a_{k-1}z^{k-1} = 1 + 2z\sum_{k=0}^{\infty} a_k z^k = 1 + 2zA(z).$$

We have

$$A(z) + B(z) = 1 + 2zA(z) \tag{1}$$

For $z = 1$ one obtains $B(1) = 1 + A(1)$; but $B(1) = \sum_{k=0}^{\infty} b_k$ is the number of external vertices of T; $A(1) = \sum_{k=1}^{\infty} a_k$ is the number of internal vertices of T, hence the number of external vertices is one more than the number of internal vertices.

By differentiating (1) one gets
$A'(z) + B'(z) = 2\,A(z) + 2\,zA'(z)$. If $A(1) = N$, then for $z = 1$ this equality yields: $B'(1) = A'(1) + 2\,N$. But $A'(1) = \sum_{k=1}^{\infty} ka_k = I(\mathrm{T})$ and $B'(1) = \sum_{k=1}^{\infty} kb_k = E(T)$, hence $E(T) = I(T) + 2\,N$. □

The tree representation of algorithm **B** shows us also how to compute the average number of comparisons in a simple way.

Let C_N be the average number of comparisons in a successful search, assuming that each of the N kays is an equally likely argument; and let C'_N be the average number of comparisons in an unsuccessful search, assuming that each of the $N + 1$ intervals between keys is equally likely. Then we have

$$C_N = 1 + \frac{I(T)}{N}; \qquad C'_N = \frac{E(T)}{N+1}$$

From $E(T) = I(T) + 2\,N$ we deduce

$$C_N = \left(1 + \frac{1}{N}\right)C'_N - 1 \tag{2}$$

This formula holds for search methods which correspond to binary trees, i.e., for all methods which are based on nonredundant comparisons. It follows that C_N is minimum if and only if C'_N is minimum, or by lemma on p. 80 if and only if all external vertices of T belong to at most two consecutive levels. By theorem 1 this is the case for binary search which minimizes the average number of comparisons in both cases of a successful or an unsuccessful search.

§ 4. Binary tree searching

We proved in the preceding section that for a given value of n, the tree corresponding to binary search achieves the theoretical minimum number of comparisons that are necessary to search a table by means of key comparisons. But the methods of the preceding section are appropriate mainly for fixed – size tables, since the sequential allocation of records makes insertions and deletions rather expensive. If the table is dynamically changing, we might spend more time maintaining it than we save in binary searching it. The use of an explicit binary tree structure makes it possible to insert and delete records quickly, as well as to search the table efficiently.

As a result, we essentially have a method which is useful both for searching and for sorting. This gain in flexibility is achieved by adding two link fields to each record of the table.

Techniques for searching a growing table are often called symbol table algorithms, because assemblers and compilers and other system routines generally use such methods to keep track of user – defined symbols.

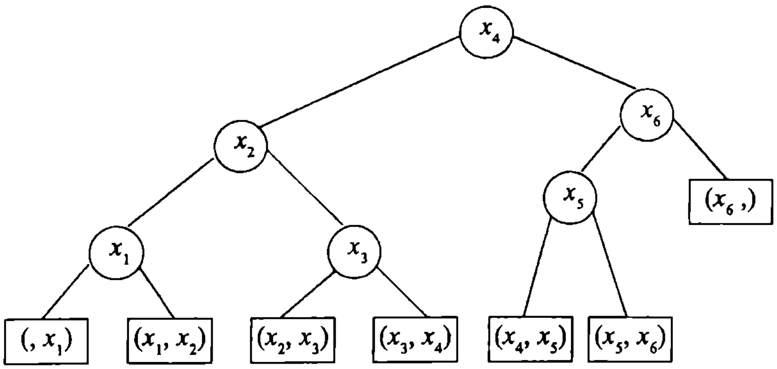
By representing the tree explicitely by pointers and not implicitely by an array the operations Insert and Delete can also be executed fast as we will see shortly.

This leads to the following definition:

A binary search tree for set $S = \{x_1 < x_2 <...< x_n\}$ is a binary tree with n internal vertices $\{v_1,...,v_n\}$. These vertices are labelled with the elements of S, i.e. there is an injective mapping CONTENT: $\{v_1,...,v_n\} \rightarrow S$.

The labelling preserves the order, i.e. if $v_i(v_j)$ is a vertex in the left (right) subtree of the tree with root v_k then CONTENT $[v_i]$ < CONTENT $[v_k]$ < CONTENT $[v_j]$.

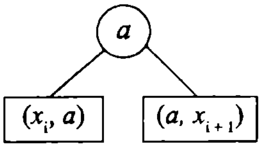
An equivalent definition is as follows: a traversal of a search tree for S in symmetric order reproduces the order on S. We will mostly identify internal vertices with their labellings, i.e. instead of speaking of vertex v with label x we speak of vertex x and write x. Vertex x corresponds to the test: *if $a < x$ then* go to the left son *else if $a = x$ then* terminate search *else* go to the right son *fi fi*. The $n + 1$ leaves represent unsuccessful access operations. It is not necessary to store leaves explicitely. Each leaf represents one of the $n + 1$ open intervals of the universe U generated by the elements of S. We draw the leaf corresponding to interval $x_i < a < x_{i+1}$ as $\boxed{(x_i, x_{i+1})}$. An example of a binary search tree for $n = 6$ is the following:



and it corresponds to a binary search algorithm when at step 2 $i \leftarrow \lceil (l+u)/2 \rceil$ instead of $i \leftarrow \lfloor (l+u)/2 \rfloor$ (see p. 103).

Leaf $(, x_1)$ represents all $a \in U$ with $a < x_1$. If a searching algorithm terminates in vertex v then a = CONTENT $[v]$, if we perform operation Access (a, S) in T, a binary search uree for set S.

Otherwise it terminates in a leaf, say (x_i, x_{i+1}). Then $x_i < a < x_{i+1}$. Operation Insert (a, S) is now very easy to implement. We only have to replace leaf (x_i, x_{i+1}) by the tree



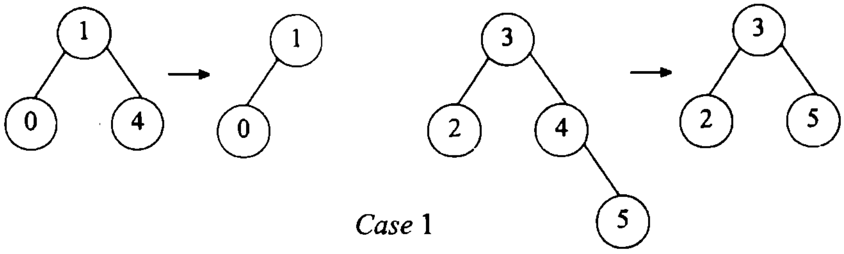
Deletion is slightly more difficult. A search for a yields internal vertex v with content a. We have to distinguish two cases.

Case 1: At least one son of v is a leaf, say the left. Then we replace v by its right son and delete v and its left son from the tree.

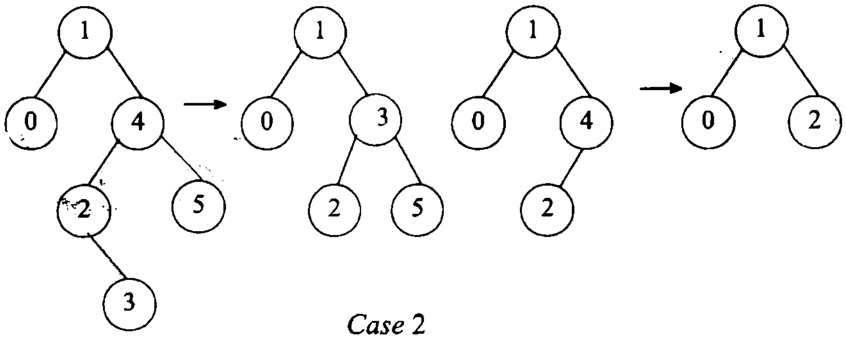
Case 2: No son of v is a leaf. Let w be the rightmost vertex in the left subtree of v (the last internal vertex produced by traversing the left subtree of v in symmetric order). Vertex w can be found by following the left pointer out of v and then always the right pointer until a leaf is hit.

We replace CONTENT [v] by CONTENT [w] and delete w as described in case 1. Note that w's right son is a leaf.

The following figure illustrates both cases. The vertex with content 4 is deleted, leaves are not drawn.



Case 1



Case 2

Of course, the height of the search tree plays a crucial role for the efficiency of the basic operations.

The following algorithm spells out the searching and insertion processes in detail.

Algorithm T. (Tree search and insertion). Given a table of records which form a binary tree as described above, this algorithm searches for a given argument K.

If K is not in the table, a new vertex containing K is inserted into the tree in the appropriate place.

The vertices of the tree are assumed to contain at least the following fields:

KEY (P) = key stored in NODE (P)
LLINK (P) = pointer to left subtree of NODE (P)
RLINK (P) = pointer to right subtree of NODE (P).

Null subtrees are represented by the null pointer Λ. The variable ROOT points to the root of the tree. We assume that the tree is not empty (i.e., ROOT $\neq \Lambda$).

1. Set $P \leftarrow$ ROOT.

2. If $K <$ KEY (P), go to **3**; if $K >$ KEY (P), go to **4**; and if $K =$ KEY (P), the search terminates successfully.

3. If LLINK (P) $\neq \Lambda$, set $P \leftarrow$ LLINK (P) and go back to **2**. Otherwise go to **5**.

4. If RLINK (P) $\neq \Lambda$, set $P \leftarrow$ RLINK (P) and go back to **2**.

5. (The search is unsuccessful; we will now put K into the tree). Set $Q \Leftarrow$ AVAIL, the address of a new node. Set KEY (Q) $\leftarrow K$, LLINK (Q) $\leftarrow$ RLINK (Q) $\leftarrow \Lambda$. If K was less than KEY (P), set LLINK (P) $\leftarrow Q$, otherwise set RLINK (P) $\leftarrow Q$ and terminate the algorithm. It can be proved that the average height of a randomly grown tree is O (log n) and that tree search will require only about 2 ln $N \approx 1.386 \log_2 N$ comparisons, if the keys are inserted into the tree in random order. Hence well–balanced trees are common, and degenerate trees are very rare.

There is a simple proof of this fact.

Let us assume that each of the $N!$ possible orderings of the N keys is an equally likely sequence of insertions for building the tree. The number of comparisons needed to find a key is exactly one more than the number of comparisons that were needed when that key was entered into the tree. Therefore if C_N is the average number of comparisons involved in a successful search and C'_N is the average number in an unsuccessful search, we have

$$C_N = 1 + \frac{C'_o + C'_1 + \ldots + C'_{N-1}}{N} \qquad (1)$$

But the relation between internal and external path length implies

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1 \qquad (2)$$

This is Eq. (2) p. 108. Putting this together with (1) yields

$$(N+1)C'_N = 2N + C'_o + C'_1 + \ldots + C'_{N-1}$$

Subtracting the equation

$$NC'_{N-1} = 2(N-1) + C'_o + C'_1 + \ldots + C'_{N-2}$$

we obtain

$$(N+1)C'_N - NC'_{N-1} = 2 + C'_{N-1}, \text{ or } C'_N = C'_{N-1} + \frac{2}{N+1}.$$

Since $C'_0 = 0$, this means that

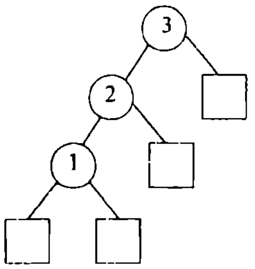
$$C'_N = 2H_{N+1} - 2$$

Applying (2) and simplifying yields the desired result $C_N = 2\left(1+\frac{1}{N}\right)H_N - 3 \sim 2\ln N$.

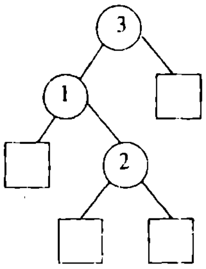
§ 5. Weighted trees

In this section we consider operation Access applied to weighted sets S. We associate a weight (access probability) with each element of S. Large weight indicates that the element is important and accessed frequently; it is desirable that these elements are high in the tree and can therefore be accessed fast.

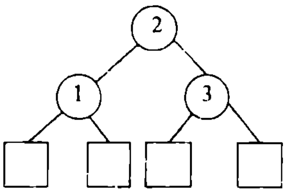
Let us now explore the problem of finding the optimum tree. When $N = 3$, for example, let us assume that the keys $K_1 < K_2 < K_3$ have respective probabilities p, q, r. There are five possible binary trees with three internal vertices:



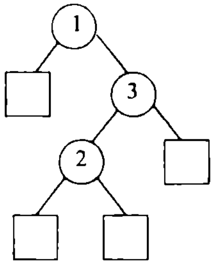
$3p + 2q + r$



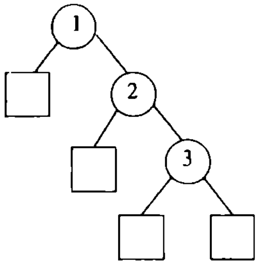
$2p + 3q + r$



$2p + q + 2r$



$p + 3q + 2r$



$p + 2q + 3r$

We obtain in this way five algebraic expressions for the average number of comparisons in a search.

When N is large, the number of binary trees having N internal vertices (the N–th Catalan number $C_N = \frac{1}{N+1}\binom{2N}{N}$) is asymptotically equal to $4^N/(N\sqrt{\pi N})$ by Stirling´s formula, so we cannot try them all and see which is best.

So far we have considered only the probabilities for a successful search; in practice, the unsuccessful case must usually be considered as well. Therefore let us set the problem up in the following way:

Let $S = \{K_1 < K_2 < ... < K_n\}$ and let p_i (q_j) be the probability of operation Access (a, S) where $a = K_i$ ($K_j < a < K_{j-1}$) for $1 \le i \le n$ ($0 \le j \le n$). By convention, q_0 is the probability that the search argument is less than K_1 and q_n is the probability that the search argument is greater than K_n.

Then p_i, $q_j \ge 0$ and $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$.

The $(2n + 1)$ – tuple $(q_0, p_1, q_1, ..., p_n, q_n)$ is called access (probability) distribution.

Let T be a search tree for set S, let α_i^T be the depth (level) of internal vertex i (the i–th internal vertex in symmetric order) and let β_j^T be the depth of leaf j (the $(j + 1)$ – st external vertex or leaf (K_j, K_{j+1})).

Consider a search for element a of the universe. If $a = K_i$ then we compare a with $\alpha_i^T + 1$ elements in the tree; if $K_j < a < K_{j+1}$ then we compare a with β_j^T elements in the tree. Hence

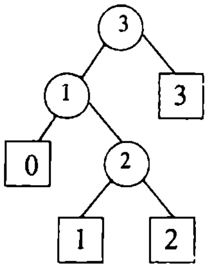
$$P^T = \sum_{i=1}^{n} p_i\left(1+\alpha_i^T\right) + \sum_{j=0}^{n} q_j\beta_j^T$$

is the average number of comparisons in a search. P^T is called weighted path length of tree T (or the cost of T relative to the given access probability distribution).

We take P^T as our basic measure for the efficiency of operation Access; the expected number of comparisons in the search is proportional to P^T.

We will suppress index T if tree T can be inferred from the context.

For example the expected number of comparisons (or the weighted path length) for the binary tree T_1 is
$2q_0 + 2p_1 + 3q_1 + 3p_2 + 3q_2 + p_3 + q_3$,
since $\alpha_1 = 1$, $\alpha_2 = 2$, $\alpha_3 = 0$,
$\beta_0 = 2$, $\beta_1 = 3$, $\beta_2 = 3$, $\beta_3 = 1$.



T_1

We associated with every search tree for set S a real number, its weighted path length (or shortly, its cost). We can therefore ask for the tree with the minimal weighted path length (cost). This tree will then also optimize average access time.

If $(q_0, p_1, q_1, \ldots, p_n, q_n)$ is a fixed access distribution for $S = \{K_1 < K_2 < \ldots < K_n\}$, tree T is said to be an optimum binary search tree for set S if its weighted path length is minimal among all search trees for set S. In this definition there is no need to require that the p's and q's sum to unity, we can ask for a minimum – cost tree with any given sequence of "weights" $(q_0, p_1, q_1, \ldots, p_n, q_n)$.

We use dynamic programming, i.e. we will construct in a systematic way optimal solutions for increasingly larger subproblems to show that an optimum binary search tree for set S and distribution of weights $(q_0, p_1, q_1,\dots,p_n, q_n)$ can be constructed in time $O(n^3)$ and space $O(n^2)$.

A search tree for set S has internal vertices $1, 2,\dots,n$ and external vertices $0, 1,\dots, n$ (or $(, K_1), (K_1, K_2),\dots, (K_n ,))$. A subtree might have internal vertices $i + 1,\dots,j$ and leaves $i,\dots,j$ for $0 \le i, j \le n$, $i \le j - 1$. Such a subtree is a search tree for set $\{K_{i+1} < \dots < K_j\}$.

If we denote $w(i, j) = p_{i+1} +\dots+ p_j + q_i +\dots+ q_j$, the cost of such a subtree by $P(i, j)$, and if its root is $k(i < k \le j)$, then the cost of this subtree is related to the costs of its subtrees (left and right) by:

$$P(i, j) = w(i, j) + P(i, k-1) + P(k, j)$$

Indeed, the left subtree of the root k has leaves $i, i + 1,\dots, k-1$; the right subtree has leaves $k, k + 1,\dots, j$ and the level of each vertex in the left or right subtree of k is less by 1 than the level of that vertex in the tree having root k. Let $c(i, j)$ be the cost of an optimum subtree with weights $(p_{i+1},\dots,p_j; q_i,\dots,q_j)$ and suppose that $c(i, j)$ and $w(i, j)$ are defined for $0 \le i \le j \le n$. It follows that

$$\begin{aligned} &c(i, i) = 0 \\ &c(i, j) = w(i, j) + \min_{i < k \le j} (c(i, k-1) + c(k, j)) \\ &\text{for } i < j \end{aligned} \tag{1}$$

since both left and right subtrees of root k must be optimum.

When $i < j$, let $R(i, j)$ be the set of all k for which the minimum is achieved in (1); this set specifies the possible roots of the optimum trees.

Eq. (1) makes it possible to evaluate $c(i, j)$ for $j - i = 1, 2,\dots,n$; there are $\binom{n}{2} \sim \frac{1}{2}n^2$ such values, and the minimization operation is carried out

$$\sum_{i=1}^{n-1} i(n-i) = \frac{n^3}{3} + O(n^2)$$

times.

This means we can determine an optimum tree in $O(n^3)$ units of time, using $O(n^2)$ cells of memory. However, time complexity can be decreased to $O(n^2)$ using a monotonicity property of $R(i, j)$.

Example:

Let $n = 4$; $q_0 = 4$, $p_1 = 1$, $q_1 = 0$, $p_2 = 3$, $q_2 = 0$, $p_3 = 3$, $q_3 = 3$, $p_4 = 0$, $q_4 = 10$. We get: $c(0,1) = w(0,1) = q_0 + p_1 + q_1 = 5$; $c(1,2) = w(1, 2) = 3$, $c(2,3) = w(2,3) = 6$, $c(3,4) = 13$.

Now we compute $c(i, j)$ for $j - i = 2$, i.e.,

$$c(0,2) = w(0,2) + \min_{k=1,2} (c(0,k-1) + c(k, 2)) =$$
$$= 8 + \min(3, 5) = 11\ (k = 1)$$
$$c(1,3) = w(1,3) + \min_{k=2,3} (c(1, k-1) + c(k, 3)) =$$
$$= 9 + \min(6, 3) = 12\ (k = 3)$$
$$c(2,4) = w(2,4) + \min_{k=3,4} (c(2, k-1) + c(k, 4)) =$$
$$= 16 + \min(13, 6) = 22\ (k = 4).$$

The values of k in parentheses indicate the points where minimum was reached. Further, let $j - i = 3$:

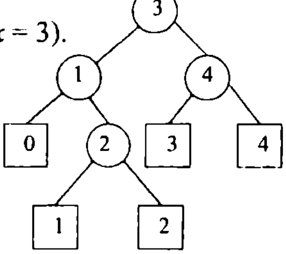
$$c(0,3) = w(0,3) + \min_{k=1,2,3} (c(0, k-1) + c(k,3)) =$$
$$= 14 + \min(12, 11, 11) = 25\ (k = 2, 3).$$

$$c(1,4) = w(1, 4) + \min_{k=2,3,4}(c(1, k-1) + c(k, 4)) =$$
$$= 19 + \min(22, 16, 12) = 31\ (k = 4).$$

Numbers $w(i, j)$ can be computed by recurrence since $w(i, j + 1) = w(i, j) + p_{j+1} + q_{j+1}$.

Finally,

$$c(0,4) = w(0,4) + \min_{k=1,2,3,4}(c(0, k-1) + c(k, 4)) =$$
$$= w(0,4) + \min(31, 27, 24, 25)\ (k = 3).$$



It follows that the root of the minimum search tree is 3, hence the root of the right subtree of 3 is 4 and the right subtree of 3 is completely determined.

The left subtree of 3 has leaves 0, 1, 2; it corresponds to $i = 0$ and $j = 2$. The cost of a minimum subtree of this type is precisely $c(0,2) = 11$ and its root is 1. It follows that optimum search tree for this distribution of weights is represented as above.

Of course, we can transform our initial access weight distribution into an access probability distribution by dividing each component by 24.

§ 6. **Balanced trees**

The tree insertion algorithm will produce good search trees, when the input data is random, but there is still the annoying possibility that a degenerate tree will occur.

A solution to the problem of maintaining a good search tree was discovered in 1962 by G.M. Adelson-Velskii and E.M. Landis. Their method requires only two extra bits per node, and it never uses more than $O(\log N)$ operations to search the tree or to insert an item.

In fact, we shall see that their approach also leads to a general technique that is good for representing arbitrary linear lists of length N, so that each of the following operations can be done in only $O(\log N)$ units of time:

i) Find an item having a given key.
ii) Find the k–th item, given k.
iii) Insert an item at a specified place.
iv) Delete a specified item.

If we use sequential allocation for linear lists, operations (i) and (ii) are efficient but operations (iii) and (iv) take order N steps; on the other hand, if we use linked allocation, operations (iii) and (iv) are efficient but operations (i) and (ii) take order N steps. A tree representation of linear lists can do all four operations in $O(\log N)$ steps. And it is also possible to do other standard operations with comparable efficiency, for example list concatenation.

The method for achieving all this involves what we shall call „balanced trees" (or AVL – trees).

However in applications which do not involve all four of the above operations, we may be able to get by with substantially less overhead and simpler programming. Furthermore, there is no advantage to use balanced trees unless N is reasonably large. Balanced trees are appropriate chiefly for internal storage of data; since internal memories seem to be getting larger and larger as time goes by, balanced tree are becoming more and more important.

A binary tree is called *balanced* if the height of the lelf subtree of every vertex never differs by more than ±1 from the height of its right subtree. The balance factor within each vertex is by definition the height of the right subtree minus the height of the left subtree. If a binary tree is balanced, then the balance factor in each vertex is 1, 0 or −1.

An important class of balanced trees is the class of Fibonacci trees, defined as follows:

Consider first the sequence $(F_n)_{n\geq 1}$ of Fibonacci numbers, defined by $F_1 = F_2 = 1$ and

$$F_{n-2} = F_{n+1} + F_n \text{ for every } n \geq 1. \tag{1}$$

In order to find an explicit formula for Fibonacci numbers, we shall find a solution for recurrence (1) of the form $F_n = r^n$. It follows that r verifies the quadratic equation:

$$r^2 - r - 1 = 0$$

having solutions $r_{1,2} = \dfrac{1 \pm \sqrt{5}}{2}$.

Hence general solution of (1) is

$$F_n = C_1 r_1^n + C_2 r_2^n$$

where constants C_1, C_2 will be determined from initial conditions $F_1 = 1$ and $F_2 = 1$. Hence by solving the linear system of equations

$$C_1 \frac{1+\sqrt{5}}{2} + C_2 \frac{1-\sqrt{5}}{2} = 1$$

$$C_1 \frac{3+\sqrt{5}}{2} + C_2 \frac{3-\sqrt{5}}{2} = 1$$

we get $C_1 = \dfrac{1}{\sqrt{5}}$ and $C_2 = -\dfrac{1}{\sqrt{5}}$, hence

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \tag{2}$$

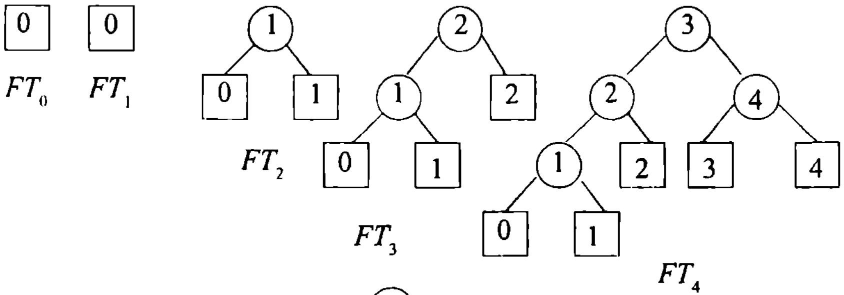
This is the so-called Binet´s formula for Fibonacci numbers.

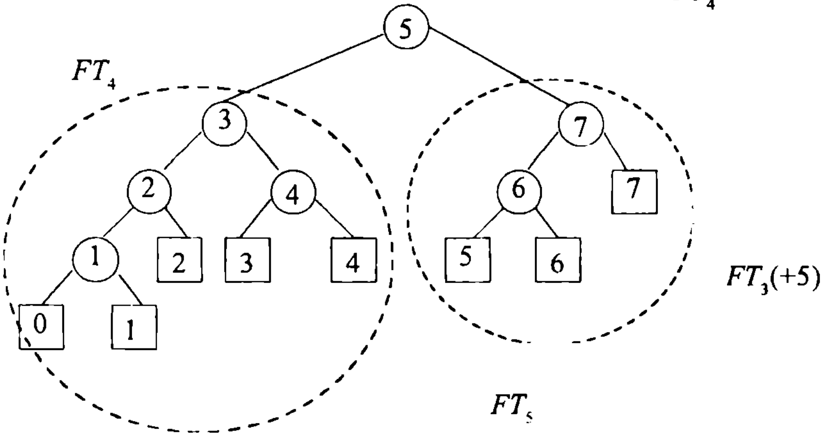
The first terms of Fibonacci sequence are:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Fibonacci trees, denoted by FT_k ; $k = 0, 1, 2, \ldots$ are labelled binary trees defined by recurrence as follows: FT_0 and FT_1 consist each of a single (external) vertex: $\boxed{0}$.

Now for every $k \geq 2$, Fibonacci tree of order k, FT_k has a root having label F_k; the left subtree of the root is FT_{k-1} and the right subtree is FT_{k-2} having all labels of the vertices (internal and external) increased by F_k, the label of the root of FT_k.





The basic properties of such kind of trees are contained in Lemma 1; an extremal property of Fibonacci trees will be proved in Theorem 2.

Lemma 1. *For every $k \geq 1$, Fibonacci tree FT_k is a balanced tree having height $h(FT_k) = k - 1$, F_{k+1} external vertices and $F_{k+1} - 1$ internal vertices.*

Proof: For $k = 1$ and $k = 2$ the property is verified. Suppose that it is true for all Fibonacci trees $FT_{k'}$ with $k' < k$ and let FT_k a Fibonacci tree of order k ($k \geq 3$). Then from the recursive definition we get $h(FT_k) = h(FT_{k-1}) + 1 = k - 1$, the number of external vertices of FT_k is equal to $F_k + F_{k-1} = F_{k+1}$, hence the number of internal vertices is equal to $F_{k+1} - 1$.

The balance property is verified by the inductioh hypothesis for all vertices different from the root of FT_k. For the root of FT_k its left subtree has height $k - 2$ and the right subtree height $k - 3$, hence FT_k is a balanced tree also . □

This definition of balance represents a compromise between optimum binary trees (with all external vertices required to be on two adjacent levels) and arbitrary binary trees (unrestricted). It is therefore natural to ask how far from optimum a balanced tree can be.

The answer is that its search paths will never be more than 45 percent longer than the optimum.

Theorem 2. *The height of a balanced tree T with N internal vertices always lies between* $\log_2(N + 1)$ and $1.4404 \log_2(N + 2) - 0.328$.

Proof: A binary tree of height h cannot have more than 2^h external vertices; so $N + 1 \leq 2^{h(T)}$ that is, $h(T) \geq \log_2 (N + 1)$.

In order to find the maximum value of h, let us turn the problem around and ask for the minimum number of internal vertices possible in a balanced tree of height h.

Let T_h be such a tree with fewest possible vertices; the one of the subtrees of the root, say the left subtree, has height $h - 1$, and the other subtree has height $h - 1$ or $h - 2$.

Since we want T_h to have the minimum number of vertices, we may assume that the left subtree has height $h - 1$, and the right subtree has height $h - 2$. So we may suppose that the left subtree of the root is T_{h-1}, and that the right subtree is T_{h-2}. This argument shows that it can be proved by induction that the Fibonacci tree of order $h + 1$, FT_{h+1}, has the fewest possible vertices among all balanced trees of height h and this minimum number of vertices is $F_{h+2} - 1$ by Lemma 1. Thus

$$N \geq F_{h+2} - 1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} - 1$$

In this expression, $\frac{1-\sqrt{5}}{2} \in (-1,0)$, hence

$$-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} \geq -1 \text{ and } N \geq \frac{1}{\sqrt{5}}\varphi^{h+2} - 2, \text{ where } \varphi = \frac{1+\sqrt{5}}{2} > 1 .$$

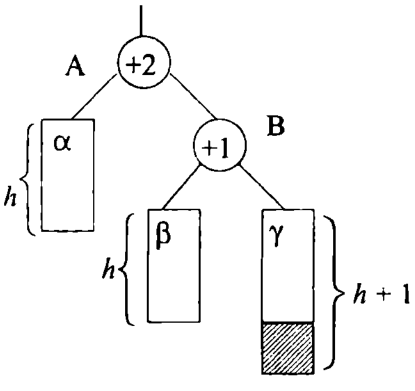
It follows that $\log_2(N+2) \geq (h+2)\log_2\varphi - \frac{1}{2}\log_2 5$, hence

$$h = h(T) \leq \frac{1}{\log_2\varphi}\log_2(N+2) + \frac{\log_2 5}{2\log_2\varphi} - 2 .$$

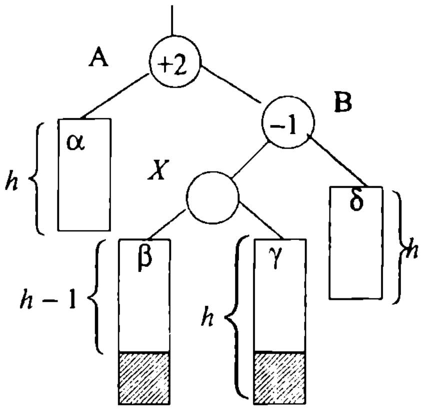
Now $\frac{1}{\log_2\varphi} < 1.4404$ and $\frac{\log_2 5}{2\log_2\varphi} - 2 < -0.328$. □

The proof of this theorem shows that a search in a balanced tree will require more than 25 comparisons only if the tree contains at least $F_{27} - 1 = 196, 417$ vertices.

Consider now what happens when a new vertex is inserted into a balanced tree using tree insertion. The problem arises when we have a vertex with a balance factor of 1 whose right subtree got higher after the insertion; or, dually, if the balance factor is – 1 and the left subtree got higher, when some adjustment will be needed. It is not difficult to see that there are essentially only two cases which cause trouble:



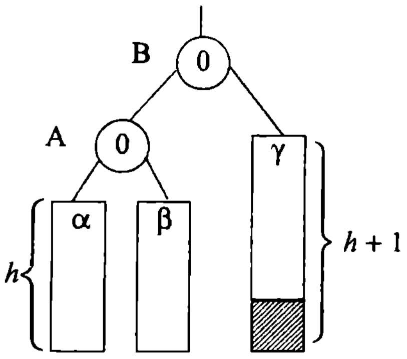
Case 1



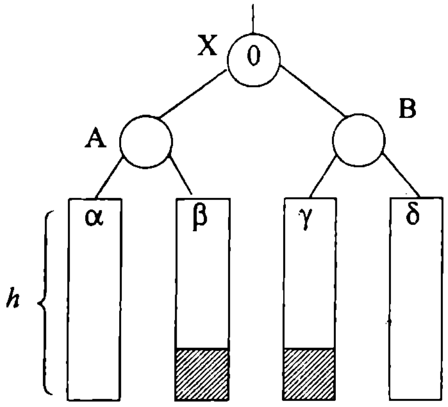
Case 2

Two other essentially identical cases occur if we reflect these diagrams, interchanging left and right. In these diagrams the rectangles α, β, γ, δ represent subtrees having the respective heights shown. Case 1 occurs when a new element has increased the height of vertex *B*'s right subtree from *h* to *h* + 1, and Case 2 occurs when the new element has increased the height of *B*'s left subtree. In the second case, we have either *h* = 0 (so that *X* itself was the new vertex), or else vertex *X* has two subtrees of respective heights (*h*–1, *h*) or (*h*, *h*–1).

Simple transformations will restore balance in both of the above cases, while preserving the symmetric order of the tree vertices and the height of the subtree which is rebalanced:



Case 1



Case 2

In Case 1 we simply rotate the tree to the left, attaching β to *A* instead of *B* and changing the root from *A* to *B*. This transformation is like applying the associative law to an algebraic formula, replacing α(βγ) by (αβ)γ.

In Case 2 we used a double rotation, first rotating (*X*, *B*) right, then (*A*, *X*) left; in this case X is the new root of the tree under consideration.

In both cases only a few links of the tree need to be changed (3 in the first case and 5 in the second one). Furthermore, the new trees have height $h + 2$, which is exactly the height that was present before the insertion; hence the rest of the tree (if any) that was originally above vertex A always remains balanced.

The following algorithm avoids the need for a stack in order to keep track of which vertices will be affected by the rebalance process.

Algoritm A. (B a l a n c e d t r e e s e a r c h a n d i n s e r t i o n). Given a table of records which form a balanced binary tree as described above, this algorithm searches for a given argument K.

If K is not in the table, a new vertex containing K is inserted into the tree in the appropriate place and the tree is rebalanced if necessary.

The vertices of the tree are assumed to contain KEY, LLINK, and RLINK fields as in Algorithm T (T r e e s e a r c h a n d i n s e r t i o n). We also have a new field:

$B(P) =$ balance factor of NODE(P), the height of the right subtree minus the height of the left subtree; this field always contain 1, 0 or -1. A special header vertex also appears at the top of the tree, in location HEAD; the value of RLINK (HEAD) is a pointer to the root of the tree, and LLINK (HEAD) is used to keep track of the overall height of the tree. We assume that the tree is nonempty, i. e., that RLINK (HEAD) $\neq \Lambda$.

For convenience in description, the algorithm uses the notation LINK(a, P) as a synonym for LLINK(P) if $a = -1$, and for RLINK(P) if $a = 1$.

1. Set $T \leftarrow$ HEAD, $S \leftarrow P \leftarrow$ RLINK (HEAD).

(The pointer variable P will move down the tree; S will point to the place where rebalancing may be necessary, and T always points to the father of S).

2. If $K <$ KEY(P), goto **3**; if $K >$ KEY(P), go to **4**; and if $K =$ KEY(P), the search terminates successfully.

3. Set $Q \leftarrow$ LLINK(P). If $Q = \Lambda$, set $Q \Leftarrow$ AVAIL and LLINK(P) $\leftarrow Q$ and go to step **5**. Otherwise if $B(Q) \neq 0$, set $T \leftarrow P$ and $S \leftarrow Q$. Finally set $P \leftarrow Q$ and return to step **2**.

4. Set $Q \leftarrow$ RLINK(P). If $Q = \Lambda$, set $Q \Leftarrow$ AVAIL and RLINK(P) $\leftarrow Q$ and go to step **5**. Otherwise if $B(Q) \neq 0$, set $T \leftarrow P$ and $S \leftarrow Q$. Finally set $P \leftarrow Q$ and return to step **2**.

5. Set KEY(Q) $\leftarrow K$, LLINK(Q) $\leftarrow$ RLINK(Q)$\leftarrow \Lambda$, $B(Q) \leftarrow 0$ (The fields of NODE(Q) are initialized).

6. If $K <$ KEY(S), set $R \leftarrow P \leftarrow$ LLINK(S), otherwise set $R \leftarrow P \leftarrow$ RLINK(S). Then repeatedly do the following operation zero or more times until $P = Q$:

If $K <$ KEY(P) set $B(P) \leftarrow -1$ and $P \leftarrow$ LLINK(P); if $K >$ KEY(P), set $B(P) \leftarrow 1$ and $P \leftarrow$ RLINK(P). (If $K =$ KEY(P), then $P = Q$ and we may go on to the next step).

(The balance factors on vertices between S and Q have been changed from zero to ± 1).

7. If $K <$ KEY(S) set $a \leftarrow -1$, otherwise set $a \leftarrow 1$. Several cases now arise:

i) If $B(S) = 0$ (the tree has grown higher), set B(S) $\leftarrow a$, LLINK(HEAD) $\leftarrow$ LLINK(HEAD) + 1, and terminate the algorithm.

ii) If $B(S) = -a$ (the tree has gotten more balanced), set $B(S) \leftarrow 0$ and terminate the algorithm.

iii) If $B(S) = a$ (the tree has gotten out of balance), go to step **8** if $B(R) = a$, or to **9** if $B(R) = -a$. (Case (iii) corresponds to Cases 1 and 2 on p. 122 when $a = 1$; S and R point, respectively, to vertices A and B, and LINK($-a,S$) points to α, LINK($-a,R$) points to β (or to X in Case 2) etc.).

8. Set $P \leftarrow R$, LINK(a,S) $\leftarrow$ LINK($-a,R$), LINK($-a,R$) $\leftarrow S$, $B(S) \leftarrow B(R) \leftarrow 0$. Go to **10**. (Single rotation).

9. Set $P \leftarrow$ LINK($-a,R$), LINK($-a,R$) $\leftarrow$ LINK(a,P), LINK(a,P) $\leftarrow R$, LINK(a,S) $\leftarrow$ LINK($-a,P$), LINK($-a,P$) $\leftarrow S$. Now set

$$(B(S), B(R)) \leftarrow \begin{cases} (-a,0) & \text{if } B(P) = a; \\ (0,0) & \text{if } B(P) = 0; \\ (0,a) & \text{if } B(P) = -a; \end{cases}$$

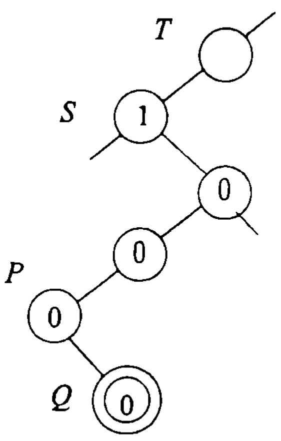
and then set $B(P) \leftarrow 0$. (Double rotation).

10. If S = RLINK(T) then set RLINK(T) $\leftarrow P$, otherwise set LLINK(T) $\leftarrow P$ (P points to the new root and T points to the father of the old root).

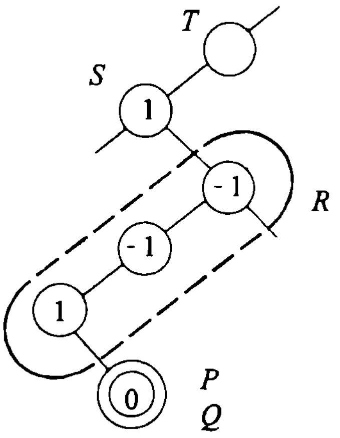
This algorithm is rather long, but it divides into three simple parts: Steps **1–4** do the search, steps **5–7** insert a new vertex, and steps **8–10** rebalance the tree if necessary.

This algorithm takes about $C \log N$ units of time, for some constant C, by theorem 2.

An example is considered below. After step **7**, $a \leftarrow 1$ since $K > \text{KEY}(S)$, $B(S) = a = 1$ and the tree has gotten out of balance;



Insertion of a new vertex Q at steps **3** or **4** plus **5**. T is the father of S; $B(S) \neq 0$ but all balance factors on vertices between S and Q are equal to zero.

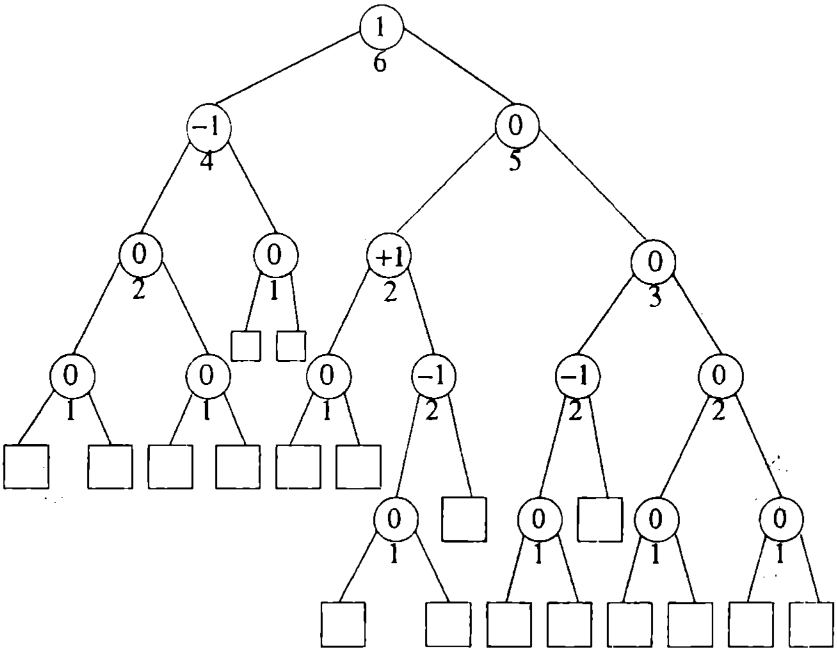


Changing the balance factors on vertices between S and Q from 0 to ± 1 at step **6** and defining the pointer R to the root of the right (left) subtree of S, depending on whether $K > \text{KEY}(S)$ or $K < \text{KEY}(S)$.

now a double rotation is necessary since $B(R) = -1 = -a$ etc.

Note that if $B(S) = 0$ at step 7i) then S points to the root of the tree, i.e. $S = \text{RLINK (HEAD)}$ and all vertices between S and Q have a balance factor equal to zero.

Linear list representation. Balanced trees can be used to represent linear lists in such a way that we can insert items rapidly (overcoming the difficulty of sequential allocation), yet can also perform random accesses to list items (overcoming the difficulty of linked allocation). The idea is to introduce a new field in each vertex, called the RANK field. This field indicates the relative position of that vertex in its subtree, i.e., one plus the number of internal vertices in its left subtree. The figure below shows the RANK values for a binary tree, as well as the balance factor in each internal vertex.



We can eliminate the KEY field entirely; or, if desired, we can have both KEY and RANK fields, so that it is possible to retrieve items either by their key value or by their relative position in the list. Using such a RANK field, retrieval by position is a straightforward modification of the search algorithms we have been studying.

Algorithm B. (Tree search by position) Given a linear list represented as a binary tree, this algorithm finds the k–th element of the list (the k–th vertex of the tree in symmetric order), given k. The binary tree is assumed to have LLINK and RLINK fields and a header as in Algorithm **A**, plus a RANK field as described above.

1. Set $M \leftarrow k$, $P \leftarrow$ RLINK(HEAD).

2. If $P=\Lambda$, the algorithm terminates unsuccessfully. (This can happen only if k was greater than the number of vertices in the tree, or $k \leq 0$). Otherwise if $M <$ RANK(P), go to **3**; if $M >$ RANK(P), go to **4**; and if $M =$ RANK(P), the algorithm terminates successfully (P points to the k–th vertex).

3. Set $P \leftarrow$ LLINK(P) and return to **2**.

4. Set $M \leftarrow M-$RANK(P) and $P \leftarrow$ RLINK(P) and return to **2**.

We can modify the insertion procedure in a similar way:

Algorithm C. (Balanced tree insertion by position). Given a linear list represented as a balanced binary tree, this algorithm inserts a new vertex just before the k–th element of the list, given k and a pointer Q to the new vertex. If $k=N+1$, the new vertex is inserted just after the last element of the list.

The binary tree is assumed to be nonempty and to have LLINK, RLINK, and B fields and a header, as in Algorithm **A**, plus a RANK field as described above. This algorithm is merely a transcription of Algorithm **A**; the difference is that it uses and updates the RANK fields instead of the KEY fields.

1. Set $T \leftarrow$ HEAD, $S \leftarrow P \leftarrow$ RLINK(HEAD), $U \leftarrow M \leftarrow k$.

2. If $M \leq$ RANK(P), go to **3**, otherwise go to **4**.

3. Set RANK(P) $\leftarrow$ RANK(P) + 1 (we will be inserting a new vertex to the left of P). Set $R \leftarrow$ LLINK(P).

If $R=\Lambda$, set LLINK(P) $\leftarrow Q$ and go to **5**. Otherwise if $B(R) \neq 0$ set $T \leftarrow P$, $S \leftarrow R$, and $U \leftarrow M$. Finally set $P \leftarrow R$ and return to **2**.

4. Set $M \leftarrow M-$RANK(P), and $R \leftarrow$ RLINK(P).

If $R=\Lambda$, set RLINK(P) $\leftarrow Q$ and go to **5**. Otherwise if $B(R) \neq 0$ set $T \leftarrow P$, $S \leftarrow R$, and $U \leftarrow M$. Finally set $P \leftarrow R$ and return to **2**.

5. Set RANK(Q) $\leftarrow$ 1, LLINK(Q) $\leftarrow$ RLINK(Q) $\leftarrow \Lambda$, $B(Q) \leftarrow 0$.

6. Set $M \leftarrow U$. (This restores the former value of M when P was S; all RANK fields are now properly set).

If $M <$ RANK(S), set $R \leftarrow P \leftarrow$ LLINK(S), otherwise set $R \leftarrow P \leftarrow$ RLINK(S) and $M \leftarrow M -$ RANK(S). Then repeatedly do the following operation until P=Q:

If $M <$ RANK(P), set $B(P) \leftarrow -1$ and $P \leftarrow$ LLINK(P); if $M >$ RANK(P), set $B(P) \leftarrow 1$ and $M \leftarrow M -$ RANK(P); $P \leftarrow$ RLINK(P). (If M=RANK(P), then P=Q and we may go on to the next step).

7. If $U <$ RANK(S), set $a \leftarrow -1$, otherwise set $a \leftarrow 1$. Several cases now arise:

i) If $B(S)$=0, set $B(S) \leftarrow a$, LLINK(HEAD) $\leftarrow$ LLINK(HEAD) + 1, and terminate the algorithm.

ii) If $B(S) = -a$, set $B(S) \leftarrow 0$ and terminate the algorithm.

iii) If $B(S) = a$, go to step **8** if $B(R) = a$, and to **9** if $B(R) = -a$.

8. Set $P \leftarrow R$, LINK(a,S) $\leftarrow$ LINK($-a$,R), LINK($-a$,R) $\leftarrow$ S, $B(S) \leftarrow B(R) \leftarrow 0$. If a=1, set RANK(R) $\leftarrow$ RANK(R) + RANK(S); if $a = -1$, set RANK(S) $\leftarrow$ RANK(S) – RANK(R).

9. Do all the operations of step **9** (Algorithm **A**).

Then if a=1, set RANK(R) $\leftarrow$ RANK(R) – RANK(P), RANK(P) $\leftarrow$ RANK(P) + RANK(S); if $a = -1$, set RANK(P) $\leftarrow$ RANK(P) + + RANK(R), then RANK(S) $\leftarrow$ RANK(S) – RANK(P).

10. If S = RLINK(T) then set RLINK(T) $\leftarrow P$, otherwise set LLINK(T) $\leftarrow P$.

All known classes of balanced trees can be divided into two groups: height – balanced and weight – balanced trees. In height – balanced trees one balances the height of the subtrees, in weight – balanced trees one balances the number of vertices in the left and right subtrees.

AVL – trees are an example in the class of height – balanced trees. We will discuss a representative of weight – balanced trees.

§ 7. Weight - balanced trees

Let α be a fixed real number, $\frac{1}{4} < \alpha \le 1 - \frac{\sqrt{2}}{2} \approx 0.2928$.

Definition: a) Let T be a binary tree with left subtree T_l and right subtree T_r. Then

$$\rho(T) = \frac{|T_l|}{|T|} = 1 - \frac{|T_r|}{|T|}$$

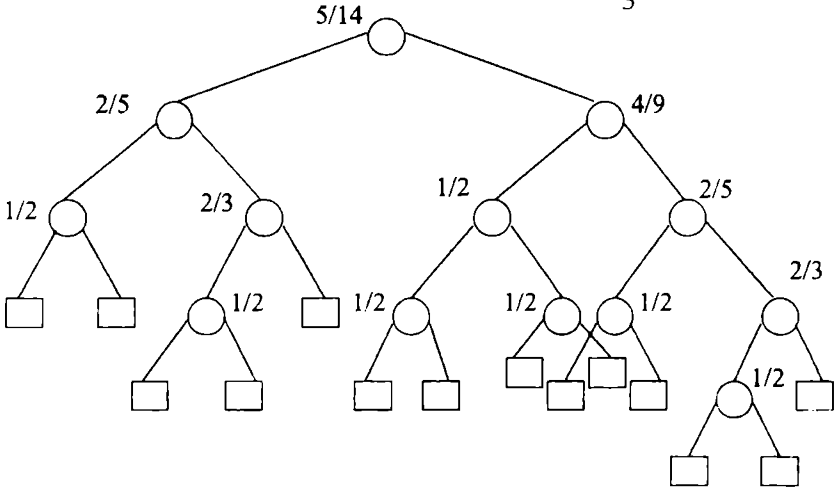
is called *the root balance* of T. Here $|T|$ denotes the number of leaves of tree T.

b) Tree T is of bounded balance α, if for every subtree T' of T one has:

$$\alpha \le \rho(T') \le 1 - \alpha$$

c) BB[α] denotes the set of all trees of bounded balance α.

In the following tree the subtrees with internal root u have root balance writen near u. The tree is in $BB[\alpha]$ for $\alpha \le \frac{1}{3}$.



Trees of bounded balance have logarithmic depth and logarithmic average internal path length.

Let $I(T) = \sum_{i=1}^{n} \alpha_i^T$ be the internal path length of T, where α_i^T denotes the depth (or level) of internal vertex i in tree T.

Theorem 1. *Let $T \in BB[\alpha]$ be a tree with n internal vertices. Then*

a) $\frac{I(T)}{n} \leq (1+\frac{1}{n})\log(n+1)\,/\,H(\alpha,1-\alpha)-2$

where entropy $H(\alpha,1-\alpha) = -\alpha\log\alpha - (1-\alpha)\log(1-\alpha)$;

b) $h(T) \leq 1 + (\log(n+1)-1)/\log(1/(1-\alpha))$.

Proof: We show $I(T) \leq (n+1)\log(n+1)/H(\alpha,1-\alpha) - 2n$ by induction on n. For $n=1$ we have $I(T)=0$. Since $0 < H(\alpha,1-\alpha) \leq 1$ this proves the claim for $n=1$. So let us assume $n > 1$.

T has a left (right) subtree with $l(r)$ vertices and internal path length $I_l(I_r)$. Then $n = l + r+1$ and $I(T) = I_l+ I_r+ n-1$. Since $T \in BB[\alpha]$ we can write

$$\alpha \leq \frac{l+1}{n+1} \leq 1-\alpha$$

Applying the induction hypothesis yields

$$I(T) = n-1+I_l+I_r \leq \frac{1}{H(\alpha,1-\alpha)}\left[(l+1)\log(l+1)+(r+1)\log(r+1)\right]-n+1=$$

$$=\frac{n+1}{H(\alpha,1-\alpha)}\left[\log(n+1)+\frac{l+1}{n+1}\log\frac{l+1}{n+1}+\frac{r+1}{n+1}\log\frac{r+1}{n+1}\right]-n+1=$$

$$=\frac{(n+1)\log(n+1)}{H(\alpha,1-\alpha)}-n+1-(n+1)\frac{H\left(\frac{l+1}{n+1},\frac{r+1}{n+1}\right)}{H(\alpha,1-\alpha)}\leq$$

$$\leq\frac{(n+1)\log(n+1)}{H(\alpha,1-\alpha)}-2n,$$

since $\frac{H\left(\frac{l+1}{n+1},\frac{r+1}{n+1}\right)}{H(\alpha,1-\alpha)} \geq 1$.

Indeed, $H(x,1-x)$ is monotonically increasing in x for $0 < x \leq \frac{1}{2}$ since by denoting $f(x) = H(x,1-x) = H(1-x,x)$, f: $(0, \frac{1}{2}] \to \mathbf{R}$, we have

$f'(x) = \log\frac{1-x}{x} \geq 0$ for every $x \in (0, \frac{1}{2}$.

If $\frac{l+1}{n+1} \leq \frac{1}{2}$, (2) follows from this property of the function f and from (1).

Otherwise, $\frac{l+1}{n+1} > \frac{1}{2}$ but in this case we have $\frac{r+1}{n+1} < \frac{1}{2}$ since their sum $\frac{l+1}{n+1} + \frac{r+1}{n+1} = 1$ and (1) implies also

$$\alpha \leq \frac{r+1}{n+1} \leq 1-\alpha \tag{3}$$

Now $H\left(\frac{l+1}{n+1}, \frac{r+1}{n+1}\right) = H\left(\frac{r+1}{n+1}, \frac{l+1}{n+1}\right) \geq H(\alpha, 1-\alpha)$ and (2) is proved. Notice that $f\left(\frac{1}{2}\right) = H\left(\frac{1}{2}, \frac{1}{2}\right) = 1$.

b) Let $T \in BB[\alpha]$ be a tree with n vertices, let k=height(T), and let $v_0, v_1, \ldots, v_{k-1}$ be a path from the root to a vertex v_{k-1} of depth (level) $k-1$. Let w_i be the number of leaves in the subtree with root v_i, $0 \leq i \leq k-1$.

Then $w_{k-1} = 2$ and $w_{i+1} \leq (1-\alpha)w_i$ for $0 \leq i < k-1$, since T is of bounded balance α.

Indeed, since $T \in BB[\alpha]$ it follows that both $|T_l| / |T|$ and $|T_r| / |T|$ are bounded above by $1-\alpha$. We deduce

$$2 = w_{k-1} \leq (1-\alpha)w_{k-2} \leq \ldots \leq (1-\alpha)^{k-1} w_0 = (1-\alpha)^{k-1}(n+1).$$

Taking logarithms finishes the proof since $\log(1-\alpha) = -\log(1/(1-\alpha)) < 0$. □

For $\alpha \approx 1-\sqrt{2}/2 \approx 0.2928$,

$l(T) \leq 1.15\left(1+\frac{1}{n}\right)\log(n+1) - 2$ and $h(T) \leq 2\log(n+1) - 1$.

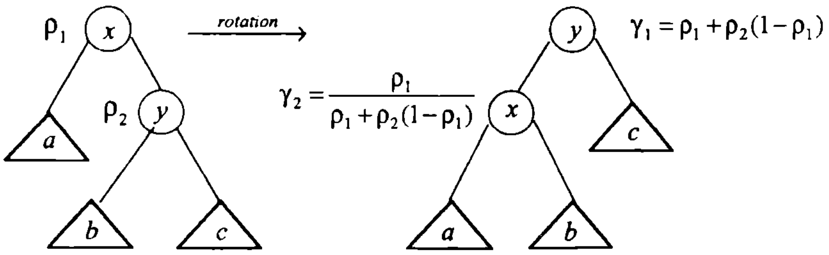
Hence a comparison with binary search algorithm shows that the average search time in trees in $BB\ [1-\sqrt{2}/2\ ]$ is at most 15% and that the maximal search time is at most by a factor of 2 above the optimum.

Operations Access, Insert and Delete are performed usually as for binary search trees.

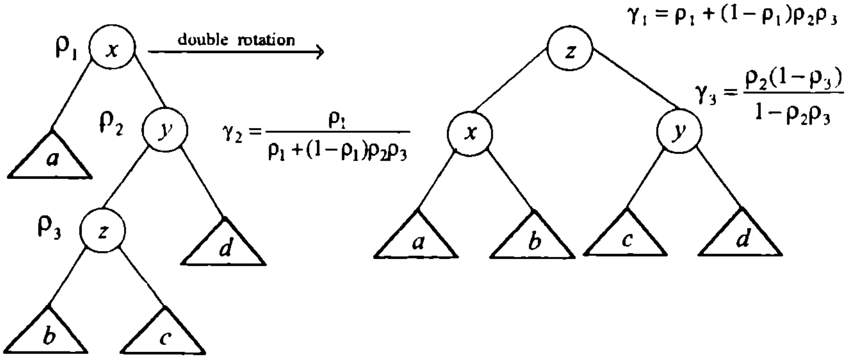
However, insertions and deletions can move the root balance for some vertices on the path of search outside the permissible range $[\alpha, 1-\alpha]$.

As for **AVL** trees, there are two transformations for remeding such a situation: rotation and double rotation. In the following figures internal vertices are drawn as circles and subtrees are drawn as triangles.

The root – balances are given beside each vertex. The figures show transformations „to the left". The symmetrical variants also exist.



ρ_1
x
rotation
ρ_2
y
a
b
c
$\gamma_1 = \rho_1 + \rho_2(1-\rho_1)$
$\gamma_2 = \frac{\rho_1}{\rho_1 + \rho_2(1-\rho_1)}$
y
x
c
a
b



ρ_1
x
double rotation
ρ_2
y
a
ρ_3
z
d
b
c
$\gamma_1 = \rho_1 + (1-\rho_1)\rho_2\rho_3$
$\gamma_2 = \frac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2\rho_3}$
$\gamma_3 = \frac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3}$
z
x
y
a
b
c
d

The root balances of the transformed trees can be computed from the old balances ρ_1, ρ_2 and ρ_3 as given in the figure. Let a, b, c be the number of leaves in the subtrees shown. Then for the rotation we have

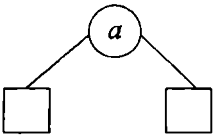
$$\rho_1 = \frac{a}{a+b+c}, \rho_2 = \frac{b}{b+c} \text{ and}$$

$$\gamma_1 = \rho_1 + \rho_2(1-\rho_1) = \frac{a}{a+b+c} + \frac{b}{b+c}\cdot\frac{b+c}{a+b+c} = \frac{a+b}{a+b+c};$$

$$\gamma_2 = \rho_1 / (\rho_1 + \rho_2(1-\rho_1)) = \frac{a}{a+b+c}\cdot\frac{1}{\gamma_1} = \frac{a}{a+b};$$

for double rotation the computation is similar.

Let us consider operation Insert first. Suppose that vertex a is added to the tree. Let $v_0, v_1, \ldots, v_k$ be the path from the root to vertex $v_k = a$. Operation Insert (a) creates the following subtree with root – balance 1/2.



We will now walk back the path towards the root and rebalance all vertices on this path. So let us assume that we reached vertex v_i and that the root balances of all proper descendants of v_i are in the range $[\alpha, 1-\alpha]$.

Then $0 \le i \le k-1$. If the root – balance of vertex v_i is still in the range $[\alpha, 1-\alpha]$ then we can move on to vertex v_{i-1}. If it is outside the range $[\alpha, 1-\alpha]$ we have to rebalance as described in the following lemma.

Lemma 1. *For all* $\alpha \in (1/4,\ 1-\sqrt{2}/2\ ]$ *there are constants* $d \in [\alpha, 1-\alpha]$ *and* $\delta \ge 0$ (*if* $\alpha < 1-\sqrt{2}/2$ *then* $\delta > 0$) *such that for* T *a binary tree with subtrees* T_l *and* T_r *and*

(1) T_l *and* T_r *are in* $BB[\alpha]$;

(2) $|T_l| / |T| < \alpha$ *and either*

(2.1) $|T_l| / (|T| - 1) \ge \alpha$ (*i.e. an insertion into the right subtree of* T *occured) or*

(2.2) $(|T_l| + 1) / (|T| + 1) \ge \alpha$ (*i.e. a deletion from the left subtree occured);*

(3) ρ_2 *is the root balance of* T_r,

we have:

(i) *if* $\rho_2 \leq d$ *then a rotation rebalances the tree, more precisely* $\gamma_1, \gamma_2 \in [(1+\delta)\alpha, 1-(1+\delta)\alpha]$ *where* γ_1, γ_2 *are as shown in the figure describing rotation.*

(ii) *if* $\rho_2 > d$ *then a double rotation rebalance the tree, more precisely* $\gamma_1, \gamma_2, \gamma_3 \in [(1+\delta)\alpha, 1-(1+\delta)\alpha]$ *where* $\gamma_1, \gamma_2, \gamma_3$ *are as shown in the figure describing double rotation.*

A complete proof is long and unelegant.

Notice only that one can find expressions for δ and d as functions of α.

□

Lemma 1 implies that a $BB[\alpha]$ – tree can be rebalanced after an insertion by means of rotations and double rotations. The transformations are restricted to the vertices on the path from the root to the inserted element. Thus height $(T) = O(\log n)$ transformations suffice; each transformation has a cost of $O(1)$.

We still have to clarify how to find the path from the inserted element back to the root and how to determine whether a vertex is out of balance. The path back to the root is easy to find. Notice that we traversed that very path when we searched for the leaf where the new element had to be inserted. We only have to store the vertices of this path in a stack; unstacking will lead us back to the root. This solves the first problem.

In order to solve the second problem we store in each vertex v of the tree not only its content, the pointers to the left and right son, but also its size, i.e. the number of leaves in the subtree with root v. So the format of a node representing a vertex is:

CONTENT	LLINK	RLINK	SIZE

The root balance of a vertex is then easily computed. Also the SIZE field is easily updated when we walk back the path of search to the root.

Theorem 2. *Let* $\alpha \in (1/4, 1-\sqrt{2}/2\,]$. *Then operations* Access (a,S), Insert (a,S), Delete (a,S) *and* Min (S) *take time* $O(\log n)$ *in* $BB[\alpha]$ *– trees, where* $n = |S|$.

Proof: An operation Insert (a,S) takes time $O(\log |S|)$. This is also true for operation Delete (a,S). Delete (a,S) removes one vertex and one leaf from the tree as described for binary search trees. (The vertex removed is not necessarily the vertex with content a). Let $v_0,\ldots,v_k$ be the path from the root v_0 to the father v_k of the removed vertex. We walk back to the root along this path and rebalance the tree as described above. The minimum of S can be found by always following left pointers starting at the root; once found the minimum can also be deleted in time $O(\log n)$. □

§ 8. Hashing

So far we have considered search methods based on comparing the given argument K to the keys in the table. Another possibility is to avoid this by doing some arithmetical calculation on K, computing a function $f(K)$ which is the location of K and the associated data in the table.

With direct addressing, an element with key K is stored in slot K. With hashing, this element is stored in slot $h(K)$; that is, a hash function h is used to compute the slot from the key K.

Here h maps the universe U of keys into the slots of a hash table $T[0\ldots m-1]$:

$$h : U \rightarrow \{0,1,\ldots, m-1\}$$

We say that an element with key K hashes to slot $h(K)$; we also say that $h(K)$ is the hash value of key K. The point of the hash function is to reduce the range of array indices that need to be handled. Instead of $|U|$ values, we need to handle only m values. Storage requirements are correspondingly reduced. Two keys may hash to the same slot, or $K_i \neq K_j$ hash to the same value $h(K_i) = h(K_j)$.

Such an occurrence is called a collision, and there are effective techniques for resolving the conflict created by collisions.

Figure 1 illustrates the basic idea.

These search methods are commonly known as hashing or scatter storage techniques; the idea in hashing is to chop off some aspects of the key and use this partial information as the basis for searching. The value of hash function $h(K)$ is the address where the search for key K begins.

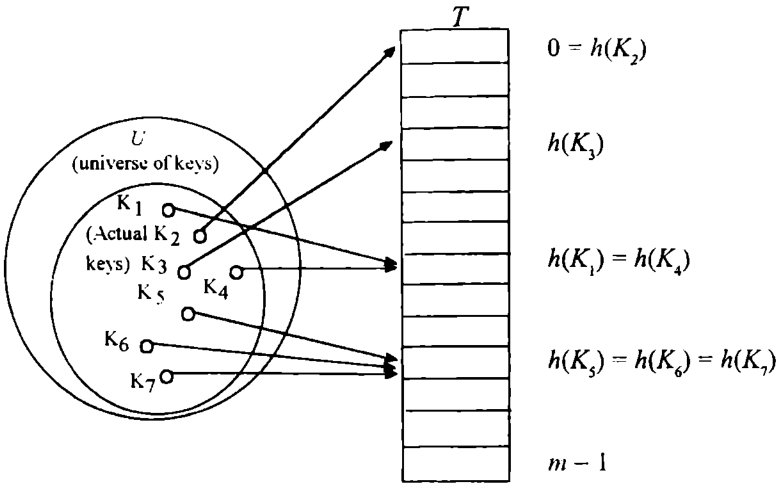


Fig. 1

In order to use a scatter table, a programmer must take two almost independent decisions: he must choose a hash function $h(K)$, and he must select a method for collision resolution. We shall consider these two aspects of the problem in turn.

Of course, the ideal solution would be to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h.

One idea is to make h appear to be „random", thus avoiding collisions or at least minimizing their number. Of course, a hash function h must be deterministic in that a given input K should always produce the same output $h(K)$.

If $|U| > m$, however, there must be two keys that have the same hash value; avoiding collisions altogether is therefore impossible.

Thus, while a well–designed hash function can minimize the number of collisions, we still need a method for resolving the collisions that do occur.

§ 9. Hash functions

In this section, we discuss some issues regarding the design of good hash functions and then present three schemes for their creation: hashing by division, hashing by multiplication, and universal hashing.

Let us assume that our hash function h takes on at most m different values, or $h(K) \in \{0,1,..., m-1\}$ for all keys K. A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots.

More formally, let us assume that each key is drawn independently from U according to a probability distribution P; that is, $P(k)$ is the probability that k is drawn. Then the assumption of simple uniform hashing is that

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{for } j = 0,1,...,m-1 \qquad (1)$$

Unfortunately, it is generally not possible to check this condition, since P is usually unknown. Sometimes we do know the distribution P. For example, suppose the keys are known to be random real numbers K, independently and uniformly distributed in the range $0 \leq K < 1$.

In this case, the hash function $h(K) = \lfloor Km \rfloor$ can be shown to satisfy equation (1).

In practice, heuristic techniques can be used to create a hash function that is likely to perform well. Qualitative information about P is sometimes useful in this design process.

For example, consider a compiler's symbol table, in which the keys are arbitrary character strings representing identifiers in a program. It is common for closely related symbols, such as $X1$, $X2$ and $X3$, to occur in the same program.

A good hash function would minimize the chance that such variants hash to the same slot.

A common approach is to derive the hash value in a way that is expected to be independent of any patterns that might exist in the data.

For example, the „division method" computes the hash value as the remainder when the key is divided by a specified prime number.

We note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are „close" in some sense to yield hash values that are far apart.

Most hash functions assume that the universe of keys is the set N = {0,1,2,...} of natural numbers. Thus, if the keys are not natural numbers, a way must be found to interpret them as natural numbers. For example, a key that is a character string can be interpreted as an integer expressed in suitable radix notation.

Thus, the identifier *pt* might be interpreted as the pair of decimal integers (112, 116), since $p = 112$ and $t = 116$ in the ASCII character set; then, expressed as a radix – 128 integer (an integer in basis 128), *pt* becomes $112 \cdot 128 + 116 = 14452$. It is usually straightforward in any given application to devise some such simple method for interpreting each key as a natural number. In what follows, we shall assume that the keys are natural numbers.

§ 10. The division method

In the division method for creating hash functions, we map a key K into one of m slots by taking the remainder of K divided by m. That is, the hash function is $h(K) = K \pmod m$. Since it requires only a single division operation, hashing by division is quite fast. When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since if $m = 2^p$, then $h(K)$ is just the p lowest–order bits of K. Unless it is known a priori that the probability distribution on keys makes all low-order p-bit patterns equally likely, it is better to make the hash function depend on all the bits of the key.

Good values for m are primes not too close to exact powers of 2.

For example, suppose we wish to allocate a hash table, to hold roughly $n = 2000$ character strings, where a character has 8 bits.

We don't mind examining an average of 3 elements in an unsuccessful search, so we allocate a hash table of size $m = 701$.

The number 701 is chosen because it is a prime near 2000/3 but not near any power of 2. Treating each key K as an integer, our hash function would be $h(K) = K \pmod{701}$.

§ 11. The multiplication method

The multiplication method for creating hash functions operates in two steps. First, we multiply the key K by a constant A in the range $0 < A < 1$ and extract the fractional part of KA. Then, we multiply this value by m and take the integer part of the result. In short, the hash function is $h(K) = \lfloor m(KA \bmod 1)\rfloor$, where $KA \bmod 1$ means the fractional part of KA, that is, $KA - \lfloor KA \rfloor$.

We typically choose m to be a power of 2, i.e., $m = 2^p$ for some integer p, since we can then easily implement the function on most computers as follows. Suppose that the word size of the machine is w bits ($p \le w$) and that K fits into a single word ($K \le 2^w - 1$).

Referring to figure 2, we first multiply K by the w – bit integer $A \bullet 2^w$ (suppose that $A \bullet 2^w \in \mathbf{Z}$). The result is a $2w$ – bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The

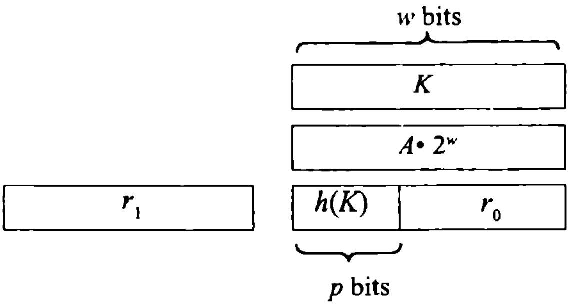


Fig. 2

desired p – bit hash value consists of the p most significant bits of r_0. Indeed, suppose that the binary representation of AK is:

$$AK = \underbrace{r_1}, \underbrace{r_0}_{\underbrace{\qquad}_{p}} \cdots$$

Then $AK \bmod 1$ is 0, $\underbrace{\qquad\qquad}_{r_0}$.. and 2^pAK is $\overbrace{\underbrace{\qquad\qquad}_{p}, \dots}^{r_0}$ and $h(K)$ is the binary word consisting of the p most significant bits of r_0.

Since $A \bullet 2^w$ is an integer, it follows that $AK2^w = (A \bullet 2^w)K = \underbrace{\qquad}_{r_1}\underbrace{\overset{p \text{ bits}}{\boxed{\qquad}}}_{r_0}$ as is represented in the last line in figure 2. If $\lfloor A \bullet 2^w \rfloor \neq A \bullet 2^w$ then the method represented in figure 2 is not longer valid.

For example, let $A = 2^{-1} + 2^{-2}$, $p = 2$, $w = 4$, $K = 2^3$.

Then $h(K) = \lfloor 2^2(2^{-1} + 2^{-2}) \rfloor = \lfloor 2 + 1 \rfloor = 3$;

$K \cdot \lfloor A \cdot 2^w \rfloor = 8 \lfloor 1 + 2^{-1} \rfloor = 8 = 1\underbrace{00}0$, hence $r_1 = 0, r_0 = 1000$ and $h(2^3) \neq 10_2 = 2$.

Although this method works with any value of the constant A, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. For example, if

$A = \varphi = \dfrac{\sqrt{5} - 1}{2} \approx 0.618034$, the golden ratio, then this choice of function h is called the Fibonacci hashing and it works reasonably well for sets of keys in arithmetic progressions.

An interesting technique is based on algebraic coding theory; the idea is analogous to the division method above, but we divide by a polynomial modulo 2 instead of dividing by an integer. For this method, $m = 2^s$, and we make use of an s-th degree polynomial $P(x) = x^s + p_{s-1}x^{s-1} + \ldots + p_0$. An n – digit binary key $K = (k_{n-1} \ldots k_1 k_0)_2$ can be regarded as the polynomial $K(x) = k_{n-1}x^{n-1} + \ldots + k_1 x + k_0$, and we compute the remainder

$K(x) \bmod P(x) = h_{s-1}x^{s-1} + \ldots + h_1 x + h_0$ using polynomial arithmetic modulo 2.

Then $h(K) = (h_{s-1}, \ldots, h_1, h_0)_2$.

If $P(x)$ is chosen properly, this hash function can be guaranteed to avoid collisions between nearly-equal keys. For example, if $n = 15$, $s = 10$, and $P(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$, it can be shown that $h(K_1) \neq h(K_2)$ whenever K_1 and K_2 are distinct keys that differ in fewer than seven bit positions.

§ 12. Universal hashing

For a fixed hash function we can choose n keys that all hash to the same slot, yielding an average retrieval time of $O(n)$.

Any fixed hash function is vulnerable to this sort of worst-case behaviour; the only effective way to improve the situation is to choose the hash functions randomly in a way that is independent of the keys that are actually going to be stored.

This approach, called universal hashing, yields good performance on the average, no matter what keys are chosen.

The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed class of functions. As in the case of quicksort, randomization guarantees that no single input will always evoke worst-case behaviour. Because of the randomization, the algorithm behave differently on each execution, even for the same input. This approach guarantees good average-case performance, no matter what keys are provided as input. Returning to the example of a compiler's symbol table, we find that the programmer's choice of identifiers cannot now cause consistently poor hashing performance.

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, ..., m-1\}$. Such a collection is said to be universal if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between x and y when $x \neq y$ is exactly $1/m$, which is exactly the change of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0, 1,..., m-1\}$.

The following theorem shows that a universal class of hash functions gives good average-case behaviour.

Theorem 1. *If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m, where $n \leq m$, the expected number of collisions involving a particular key x is less than* 1.

Proof: For each pair y, z of distinct keys, let c_{yz} be a random variable that is 1 if $h(y) = h(z)$ (i.e., if y and z collide using h) and 0 otherwise. Since, by definition, a single pair of keys collides with probability $1/m$, we have $E[c_{yz}] = 1/m$. Let C_x be the total number of collisions involving key x in a hash table T of size m containing n keys. We deduce

$E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E\left[C_{xy}\right] = \frac{n-1}{m}$. Since $n \leq m$, we have $E[C_x] < 1$. □

But how easy is it to design a universal class of hash functions? Let us choose our table size m to be prime, as in the division method. We decompose a key x into $r+1$ bytes (i.e., characters, or fixed-width binary substrings), so that $x = <x_0, x_1, \ldots, x_r>$; the only requiremen is that the maximum value of a byte should be less than m. Let $a = < a_0, a_1, \ldots, a_r >$ denote a sequence of $r+1$ elements chosen randomly from he set $\{0, 1, \ldots, m-1\}$. We define a corresponding hash function $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \pmod m \qquad (2)$$

With this definition, $\mathcal{H} = \bigcup_a \{h_a\}$ has m^{r+1} members.

Theorem 2. *The class $\mathcal{H}$ defined by equation (2) is a universal class of hash functions.*

Proof: Consider any pair of distinct keys x, y. Assume that $x_0 \neq y_0$. (A similar argument can be made for a difference in any other position).

For any fixed values of $a_1, a_2, \ldots, a_r$, there is exactly one value of a_0 that satisfies the equation $h(x) = h(y)$; this a_0 is the solution to

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod m \qquad (3)$$

Indeed, since m is prime, the nonzero quantity $x_0 - y_0$ has a multiplicative inverse modulo m, and thus there is a unique solution for a_0 modulo m.

Therefore, each pair of keys x and y collides for exactly m^r values of a, since they collide exactly once for each possible value of $< a_1, a_2, \ldots, a_r >$ (i.e., for the unique value of a_0 determined from (3)). Since there are m^{r+1} possible values for the sequence a, (hence for hash functions $h_a \in \mathcal{H}$) keys x and y collide with probability $m^r / m^{r+1} = 1/m$. Therefore, $\mathcal{H}$ is universal. □

§ 13. Hashing with chaining

In chaining, we put all the elements that hash to the same slot in a linked list, as shown in figure 3. Slot j $(0 \leq j \leq m-1)$ contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL. In general, if there are n keys and m liked lists, the average list size is n/m; thus hashing decreases the amount of work needed for sequential searching by roughly a factor of m.

A set $S \subseteq U$ is represented as m linear lits; the i – th list contains all elements $x \in S$ with $h(x) = i$. Operation Access (x, S) is realized by the following program:

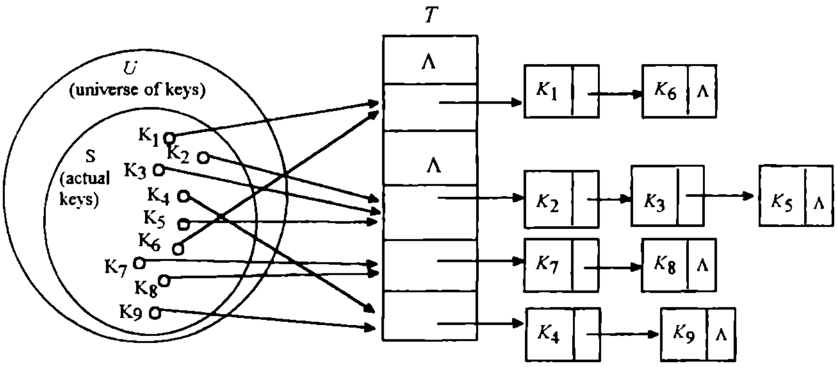


Fig. 3

1) compute $h(x)$

2) search for element x in list $T[h(x)]$.

Operations Insert (x, S) and Delete (x, S) are implemented similarly. We only have to add x to or delete x from list $T[h(x)]$.

It is often a good idea to keep the individual lists in order by key, so that insertions and unsuccessful searches go faster. Alternatively we can make use of the „self-organizing file'' concept; instead of keeping the lists in order by key, they may be kept in order according to the time of most recent occurrence.

For the sake of speed we would like to make m rather large. But when m is large, many of the lists will be empty and much of the space for the m list heads will be wasted. This suggests another approach, when the records are small: we can overlap the record storage with the list heads, making room for a total of m records and m links instead of for n records and $m+n$ links. The following algorithm is a convenient way to solve the problem.

Algorithm C. (C h a i n e d s c a t t e r t a b l e s e a r c h a n d i n s e r t i o n). This algorithm searches an m-node table, looking for a given key K. If K is not in the table and the table is not full, K is inserted.

The nodes of the table are denoted by TABLE $[i]$, for $0 \leq i \leq m$, and they are of two distinguishable types, empty and occupied. An occupied node contains a key field KEY $[i]$, a link field LINK $[i]$, and possibly other fields. The algorithm makes use of a hash function $h(K)$. An auxiliary variable R is also used, to help find empty spaces; when the table is empty, we have $R = m + 1$, and as insertions are made it will always be true that TABLE $[j]$ is occupied for all j in the range $R \leq j \leq m$. By convention, TABLE [0] will always be empty.

1. Set $i \leftarrow h(K) + 1$ (now $1 \leq i \leq m$).

2. If TABLE $[i]$ is empty, go to **6**. (Otherwise TABLE $[i]$ is occupied; we will look at the list of occupied nodes which starts here).

3. If $K =$ KEY $[i]$, the algorithm terminates successfully.

4. If LINK $[i] \neq \Lambda$, set $i \leftarrow$ LINK $[i]$ and go back to **3**.

5. (The search was unsuccessful, and we want to find an empty position in the table). Decrease R one or more times until finding a value such that TABLE $[R]$ is empty. If $R = 0$, the algorithm terminates with overflow (there are no empty nodes left); otherwise set LINK $[i] \leftarrow R$, $i \leftarrow R$.

6. (Insert a new key). Mark TABLE $[i]$ as an occupied node, with KEY $[i] \leftarrow K$ and LINK $[i] \leftarrow \Lambda$.

This algorithm allows several lists to coalesce, so that records need not be moved after they have been inserted into the table.

For example, see fig. 4, where $m = 7$ and keys
5, 19, 42, 75, 23, 18, 50
were inserted in this order, for $h(K) = K \pmod 7$.

So 18 appears in the same list as 5, 19 and 75, but we have $h(5) = h(19) = h(75) = 5$, and $h(18) = 4$.

We shall make some complexity considerations in the case when no two lists coalesce.

The time complexity of hashing with chaining is easy to determine: the time to evaluate hash function h plus the time to search through list $T[h(x)]$.

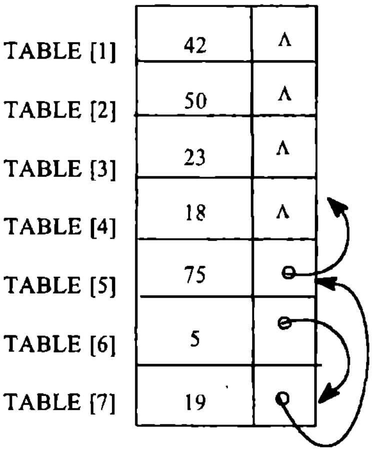


Fig. 4.

We assume in this section that h can ve evaluated in constant time and therefore define the cost of an operation referring to key x as

$O(1 + \delta_h(x, S))$ where S is the set of stored elements and

$$\delta_h(x,S) = \sum_{y \in S} \delta_h(x,y), \text{and } \delta_h(x,y) = \begin{cases} 1, \text{if } x \neq y \text{ and} \\ \quad h(x) = h(y) \\ 0, \text{otherwise} \end{cases}$$

The worst case complexity of hashing is easily determined. The worst case occurs when the hash function h restricted to set S is a constant, i.e. $h(x) = i_0$, for all $x \in S$. Then hashing deteriorates to searching through a linear list; any one of the three operations costs $O(|S|)$ time units.

Theorem 1. *The time complexity (in the worst case) of operations* Access (x, S), Insert (x, S) *and* Delete (x, S) *is* $O(|S|)$.

Average case behaviour is much better. We analyse the complexity of a sequence of n insertions, deletions and accesses starting with an empty table, i.e. of a sequence $Op_1(x_1)$,..., $Op_n(x_n)$, where $Op_k \in$ {Insert, Delete, Access} and $x_k \in U$, under the following probability assumptions:

1) Hash function $h : U \to \{0, 1, ..., m-1\}$ distributes the universe uniformly over the interval $\{0, ..., m-1\}$, i.e. for all $i, i' \in \{0, ..., m-1\}$: $|h^{-1}(i)| = |h^{-1}(i')|$.

2) All elements of U are equally likely as argument of any one of the operations in the sequence, i.e. argument of the k– th operation of the sequence is equal to a fixed $x \in U$ with probability $1/|U|$.

Our two assumptions imply that value $h(x_k)$ of the hash function on the argument of the k–th operation is uniformely distributed in $\{0, 1,..., m-1\}$, i.e. $P(h(x_k) = i) = 1/m$ for all $k \in \{1,...,n\}$ and $i \in \{0,..., m-1\}$. We call this the assumption of simple uniform hashing.

Theorem 2. *Under the assumption of simple uniform hashing, a sequence of n insertions, deletions and access – operations takes time* $O(n(1+ \beta/2))$ *where* $\beta = n/m$ *is the load factor of the table.*

Proof: We will first compute the expected cost of the $(k+1)$–st operation. Assume that $h(x_{k+1}) = i$, i.e. the $(k+1)$–st operation accesses the i-th list. Let $P(\, l_k(i) = j)$ be the probability that the i-th list has length j after the k-th operation. Then

$$EC_{k+1} \le \sum_{j\ge 0} P(l_k(i)=j)(j+1),$$

where EC_{k+1} is the expected cost of the $(k+1)$ – st operation. Notice that

$$P(l_k(i)=j) \le \frac{\binom{k}{j}(m-1)^{k-j}}{m^k} = \binom{k}{j}(1/m)^j(1-1/m)^{k-j}$$

with equality if the first k operations are insertions. (see also pp. 87-88 for a similar formula).
Hence

$$EC_{k+1} \le 1 + \sum_{j\ge 0}\binom{k}{j}(1/m)^j(1-1/m)^{k-j}\cdot j$$

$$= 1 + \frac{k}{m}\sum_{j\ge 1}\binom{k-1}{j-1}(1/m)^{j-1}(1-1/m)^{k-j}$$

$$= 1 + \frac{k}{m}\text{, since } \quad j\binom{k}{j} = k\binom{k-1}{j-1}.$$

Thus the total expected cost of n operations is

$$\sum_{k=1}^{n} EC_k = O\left(\sum_{k=1}^{n}\left(1+\frac{k-1}{m}\right)\right) =$$

$$= O\left(n + \frac{n(n-1)}{2m}\right) = O\left(n + n\frac{\beta}{2} - \frac{n}{2m}\right) = O\left(n\left(1+\frac{\beta}{2}\right)\right). \quad \Box$$

We will next discuss the probability assumptions used in the analysis of hashing. The first assumption is easily satisfied. Suppose $U = \{0,\ldots,N-1\}$, and $m|N$. Then $h(x) = x \pmod m$ will satisfy the first requirement: hash function h distributes the universe U uniformly over the hash table for the division method. If m does not divide N but N is much larger than m then the division method almost satisfies assumption 1).

The second assumption is more critical because it postulates a certain behaviour of the user of the hash function. In general, the exact conditions of latter use are not known when the hash function is designed and therefore one has to be very careful about applying theorem 2.

We discuss one particular application now and come back to the general problem.

Symboltabels in compilers are often realized by hash tabels. Identifiers are strings over the alphabet $\{A, B, C, ...\}$, i.e. $U = \{A, B, C, ...\}^*$.

The usage of identifiers is definitely not uniformly distributed over U. For example, identifiers $I1$, $I2$, $I3$, $J1$, $J2$, ... are very popular and XYZ is not. We can use this nonuniformity to obtain even better behaviour than predicted by theorem 2.

Inside a computer identifiers are represented as bit-strings; usually 8 bits (a byte) is used to represent one character.

In other words, we assign a number $\mathrm{num}(C) \in \{0, 1, ..., 2^8 - 1 = 255\}$ to each character of the alphabet and interpret a string $C_r C_{r-1} \dots C_0$ as a number in base 256, namely $\sum_{i=0}^{r} \mathrm{num}(C_i)\, 256^i$; moreover, consecutive numbers are usually assigned to characters 1, 2, 3,... Then strings $I0$, $I1$, $I2$, and $X0$, $X1$, $X2$ lead to arithmetic progressions of the form $(a+i)$ and $(b+i)$ respectively, where $i = 0, 1, 2, ...$ and $256 \mid a-b$. Because identifiers of the form $I0$, $I1$, $I2$, and $X0$, $X1$, $X2$ are used so frequently, we want that.

$$h(a+i) \neq h(b+j) \text{ for } 0 \leq i, j \leq 9 \text{ and } 256 \mid a-b.$$

If $h(x) = x \pmod m$ then we want

$$(b-a) + (j-i) \neq 0 \pmod m.$$

Since $256 \mid b-a$ we should choose m such that m does not divide numbers of the form $256\, c+d$, where $|d| \leq 9$. In this way one can obtain even better practical performance than predicted by theorem 2.

§ 14. Hashing with open addressing

Another way to resolve the problem of collisions is to do away with the links entirely, simply looking at various entries of the table one by one until either finding the key K or finding an empty position. The idea is to formulate some rule by which every key K determines a „probe sequence'', namely a sequence of table positions which are to be inspected whenever K is inserted or looked up. If we encounter an open position while searching for K, using the probe sequence determined by K, we can conclude that K is not in the table, since the same sequence of probes will be made every time K is processed. This general class of methods was named open addressing by W. W. Peterson [IBM J. Research & Development 1 (1957), 130 – 146].

Thus, in open addressing all elements are stored in the hash table itself; the hash table can „fill up'' so that no further insertions can be made. The load factor β can never exceed 1.

The advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we compute the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of positions for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

The simplest open addressing scheme, known as linear probing, uses the cyclic probe sequence

$$h(K), h(K)-1, \ldots, 0, m-1, m-2, \ldots, h(K)+1 \quad (1)$$

as in the following algorithm.

Algorithm L. (Open scatter table search and insertion). This algorithm searches an m– node table, looking for a given key K. If K is not in the table and the table is not full, K is inserted. The nodes of the table are denoted by TABLE [i], for $0 \le i \le m-1$, and they are of two distinguishable types, empty and occupied. An occupied node contains a key, called KEY [i], and possibly other fields. An auxiliary variable n is used to keep track of how many nodes are occupied; this variable is considered to be part of the table, and it is increased by 1 whenever a new key is inserted.

This algorithm makes use of a hash function $h(K)$ and it uses the linear probing sequence (1) to address the table.

1. Set $i \leftarrow h(K)$. (Now $0 \le i \le m-1$).

2. If TABLE [i] is empty, go to **4**. Otherwise if KEY [i] = K, the algorithm terminates successfully.

3. Set $i \leftarrow i-1$; if now $i < 0$, set $i \leftarrow i + m$. Go back to step **2**.

4. (The search was unsuccessful). If $n = m-1$, the algorithm terminates with overflow. (This algorithm considers the table to be full when $n = m - 1$, not when $n = m$). Otherwise set $n \leftarrow n+1$, mark TABLE [i] occupied, and set KEY [i] $\leftarrow K$.

For example, see fig. 1 where $m = 7$ and keys 5, 19, 42, 75, 23, 18 were inserted by Algorithm L in this order with hash function

$h(K) = K(\text{mod } 7)$

Now $n = m - 1 = 6$ and the table is full; no other insertion can be made.

Experience with linear probing shows that the algorithm works fine until the table begins to get full; but eventually the process slows down, with long drawn – out searches becoming increasingly frequent. Consequently the performance of linear probing degrades rapidly when n approaches m since separate lists are combined into long lists where the search is slow.

In fact, when $n = m-1$, there is only one vacant space in the table, so the average number of probes in an unsuccessful search is $(m+1)/ 2$.

TABLE [0]	42
TABLE [1]	18
TABLE [2]	23
TABLE [3]	75
TABLE [4]	19
TABLE [5]	5
TABLE [6]	

Fig. 1

A way to protect against the consecutive hash code problem is to use the following idea:

instead of being fixed in the order (1) implying consecutive positions of the table, the sequence of positions probed depends upon the key being inserted.

To determine which positions to probe, we extend the hash function to include the probe number (starting from 0) as a second input. Thus, the hash function becomes

$$h{:}U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}.$$

With open addressing, we require that for every key K, the probe sequence $< h(K,0), h(K, 1),\ldots,h(K, m-1) >$ be a permutation of $< 0, 1,$

,..., $m-1$ >, so that every hash – table position is eventually considered as a slot for a new key as the table fills up. Following this strategy for insertion and searching the algorithm **L** becomes:

Algorithm L1. (Open scatter table search and insertion using a two variable hash function).

```
1. Set $i \leftarrow 0$.
2. Set $j \leftarrow h(K, i)$.
3. If TABLE [$j$] is empty, go to 5.
   Otherwise, if KEY [$j$] = $K$, the algorithm terminates successfully.
4. If $i = m-1$, the algorithm terminates with overflow. Otherwise
   set $i \leftarrow i+1$ and go back to step 2.
5. Mark TABLE [$j$] occupied, and set KEY [$j$] $\leftarrow K$. (The key $K$
   was inserted into the table).
```

Since the algorithm for searching for key K probes the same sequence of slots that the insertion algorithm examined when key K was inserted, the search can terminate (unsuccessfully) when it finds an empty slot, since K would have been inserted there and not later in its probe sequence. (Note that this argument assumes that keys are not deleted from the hash table).

Deletion from an open – address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty. Doing so might make it impossible to retrieve any key K during whose insertion we had probed slot i and found it occupied. For this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Three techniques are commonly used to compute the probe sequences required for open addressing: linear probing, quadratic probing, and double hashing. These techniques all guarantee that $< h(K, 0), h(K, 1), ..., h(K, m-1) >$ is a permutation of $< 0, 1, ..., m-1 >$ for each key K.

Linear probing. Given an ordinary hash function $h': U \rightarrow \{0, 1, ..., m-1\}$, the method of linear probing uses the hash function $h(K, i) = $
$= h'(K) + i \pmod m$ for $i = 0, 1, ..., m-1$. Note that (1) corresponds to $h(K, i) = h'(K) - i \pmod m$.

Given key K, the first slot probed is $T[h'(K)]$.

We next probe slot $T[h'(K) + 1]$, and so on up to slot $T[m-1]$. Then we wrap arround to slots $T[0]$, $T[1]$,..., until we finally probe slot $T[h'(K) - 1]$.

Linear probing is easy to implement, but it suffers from a problem known as primary clustering. Long runs of occupied slots build up, increasing the average search time.

Quadratic probing uses a hash function of the form

$$h(K, i) = h'(K) + c_1 i + c_2 i^2 \pmod m,$$

where h' is an auxiliary hash function, c_1, $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$.

The initial position probed is $T[h'(K)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained.

Also, if two keys have same initial probe position, then their probe sequences are the same, since $h(K_1, 0) = h(K_2, 0)$ implies $h(K_1, i) =$ $= h(K_2, i)$ for any i. This leads to a milder form of clustering, called secondary clustering.

As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

One way to select parameters c_1 and c_2 is to impose that $c_1 i + c_2 i^2 =$ $= 1 + 2 + \ldots + i$, i.e., $c_1 i + c_2 i^2 = \frac{i(i+1)}{2}$, which implies that $c_1 = c_2 = \frac{1}{2}$.

Lemma 1. *If $m = 2^s$ for an integer $s \geq 1$ and*

$$h(K,i) = h'(K) + \frac{1}{2}i + \frac{1}{2}i^2 \pmod{2^s},$$

where $h'(K)$ is an auxiliary hash function, then for each key K, $< h(K, 0)$, $h(K, 1)$,..., $h(K, m-1) >$ is a permutation of $< 0, 1, \ldots, m-1 >$.

Proof: We shall prove that for each $i = 0, 1, \ldots, m-1$ all values $h'(K) + \frac{1}{2}i + \frac{1}{2}i^2 \pmod{2^s}$ are pairwise different, which will prove the assertion. Suppose, to the contrary, that there exist two indices $i, j \in \{0, \ldots, m-1\}$, $i > j$, such that $h'(K) + \frac{1}{2}i + \frac{1}{2}i^2 \equiv h'(K) + \frac{1}{2}j + \frac{1}{2}j^2$ $\pmod{2^s}$. This implies that there exists an integer $p > 0$ such that

$$\frac{1}{2}\,(i^2 - j^2 + i - j) = p2^s, \text{ or } (i-j)(i+j+1) = p2^{s+1}. \tag{2}$$

In this equality $i-j$ and $i+j$ have the same parity, since their difference is $2j$, an even number. Hence the greatest power of 2 in the prime number factorization of $(i-j)(i+j+1)$ is less than or equal to that in the prime number factorization of the numbers in the set $\{1, ..., i+j+1\}$. But $i+j+1 \le m-1+m-2+1 = 2m-2 = 2^{s+1}-2$. It follows that this greatest power of 2 is 2^s, which contradicts (2).

The property is proved. □

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form:

$$h(K, i) = h_1(K) + ih_2(K) \pmod m,$$

where h_1 and h_2 are auxiliary hash functions and $h_2(K) \in \{1,..., m-1\}$ for every K. The initial position probed is $T[h_1(K)]$; successive probe positions are offset from previous positions by the amount $h_2(K)$ modulo m.

Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key K, since the initial probe position, the offset, or both, may vary.

Figure 2 gives an example of insertion by double hashing.

Let $m = 7$ (a prime number),

$h_1(x) = x \pmod 7$ and

$h_2(x) = 1 + (x \pmod 4)$.

If we insert 5, 19, 42, 75, 23, 18, 50 in that order, we obtain

$h(5, i) = 5 + 2i \pmod 7$, $i = 0$ works

$h(19, i) = 5 + 4i \pmod 7$, $i = 1$ works

$h(42, i) = 3i \pmod 7$, $i = 0$ works

$h(75, i) = 5 + 4i \pmod 7$, $i = 2$ works

$h(23, i) = 2 + 4i \pmod 7$, $i = 2$ works

$h(18, i) = 4 + 3i \pmod 7$, $i = 0$ works

$h(50, i) = 1 + 3i \pmod 7$, $i = 0$ works

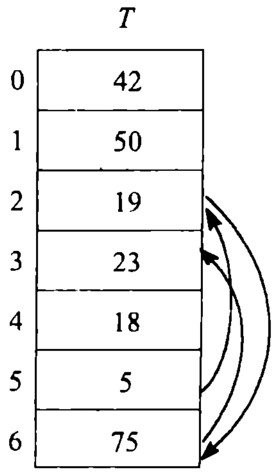


Figure 2

The value $h_2(K)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. Otherwise, if m and $h_2(K)$ have greatest common divisor $d > 1$ for some key K, then a search for key K would examine only $(1/d)$ th of the hash table.

Lemma 2. If $h(K,i) = h_1(K) + ih_2(K) \pmod m$ *where* $h_2(K) \in \{1,\dots, m-1\}$ *for every key* K *and the greatest common divisor* $(h_2(K), m) = 1$, *then for each key* K, $< h(K,0), h(K,1),\dots, h(K, m-1) >$ *is a permutation of* $< 0, 1,\dots, m-1 >$.

Proof: As for Lemma 1 we shall show that for each $i = 0,1,\dots, m-1$, all values $h_1(K) + ih_2(K) \pmod m$ are pairwise different. If there exist two indices $i,j \in \{0,1,\dots, m-1\}$, $i > j$, such that

$$h_1(K) + ih_2(K) \equiv h_1(K) + jh_2(K) \pmod m,$$

then there exists an integer $p > 0$ such that

$$(i-j)h_2(K) = pm \tag{3}$$

But (3) cannot hold since $1 \le i-j \le m-1$ and $(h_2(K), m) = 1$.□

A convenient way to ensure these conditions on h_2 is to let m be a power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m. For example, we could choose m prime and let

$$h_1(K) = K \pmod m$$
$$h_2(K) = 1 + (K \pmod{m'}),$$

where m' is chosen to be slightly less than m (say, $m-1$ or $m-2$).

By summarizing the discussion above, hashing with open addressing does not require any additional space. However, its performance becomes poorly when the load factor is nearly one and it does not support deletion.

Analysis of open – address hashing. Our analysis of open addressing, like our analysis of chaining, is expressed in terms of the load factor β of the hash table, as n and m go to infinity. Recall that if n elements are stored in a table with m positions (slots), the average number of elements per slot is $\beta = n/m$.

Of course, with open addressing, we have at most one element per slot, and thus $n \le m$, which implies $\beta \le 1$.

We assume that uniform hashing is used. In this idealized scheme, the probe sequence $< h(K,0), h(K, 1), \ldots, h(K, m-1) >$ for each key K is equally likely to be any permutation on $< 0,1,\ldots, m-1 >$, or this probe sequence is a random permutation of the set $\{0,1,\ldots m-1\}$. That is, each possible probe sequence is equally likely to be used as the probe sequence for an insertion or a search. Of course, a given key has a unique fixed probe sequence associated with it; what is meant here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyse the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

Theorem 3. *Given an open-address hash table with load factor* $\beta = n/m < 1$, *the expected number of probes in an unsuccessful search is at most* $\frac{1}{1-\beta}$ *assuming uniform hashing.*

Proof: In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty.

Let us define $p_i = P$ {exactly i probes access occupied slots} for $i = 0,1, \ldots$ For $i > n$, we have $p_i = 0$, since we can find at most n slots already occupied. Note that p_i does not depend on the probe sequence by our assumption of uniform hashing. Thus, the expected number of probes is $1+\sum_{i=0}^{\infty} ip_i$. We define also $q_i = P$ {at least i probes access occupied slots} for $i = 0, 1, 2,\ldots$ Since $p_i = q_i - q_{i+1}$ it follows that

$$\sum_{i=0}^{\infty} ip_i = \sum_{i=0}^{\infty} q_i$$

Now we will evaluate the value of q_i for $i \geq 1$. The probability that the first probe accesses an occupied slot is n/m; thus, $q_1 = \frac{n}{m}$.

With uniform hashing, a second probe, if necessary, is to one of the remaining $m - 1$ unprobed slots, $n - 1$ of which are occupied. We make a second probe only if the first probe accesses an occupied slot; thus, $q_2 = \frac{n}{m} \cdot \frac{n-1}{m-1}$.

In general, the i–th probe is made only if the first $i - 1$ probes access occupied slots, and the slot probed is equally likely to be any of the remaining $m - i + 1$ slots, $n - i + 1$ of which are occupied.

Thus, $q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \le \left(\frac{n}{m}\right)^i = \beta^i$

for $i = 1, 2, \ldots, n$, since $\frac{n-j}{m-j} \le \frac{n}{m}$ if $n \le m$ and $j \ge 0$.

After n probes, all n occupied slots have been seen and will not be probed again, and thus $q_i = 0$ for $i > n$. Given the assumption that $\beta < 1$, the average number of probes in an unsuccessful search is

$$1 + \sum_{i=0}^{\infty} i p_i = 1 + \sum_{i=1}^{\infty} q_i \le 1 + \beta + \beta^2 + \ldots = \frac{1}{1-\beta} \quad \square$$

The last equation has an intuitive interpretation: one probe is always made; with probability approximately β a second probe is needed; with probability approximately β^2 a third probe is needed, and so on.

If β is a constant, Theorem 3 predicts that an unsuccessful search runs in $O(1)$ time.

For example, if the hash table is half full ($\beta = 0.5$), the average number of probes in an unsuccessful search is $1/(1- 0.5) = 2$; if it is 90 percent full, the average number of probes is $1/(1 - 0.9) = 10$.

The cost of operation Insert (K) is 1 + min $\{i$: $T[(h(K,i)]$ is not occupied$\}$. Theorem 3 gives us the performance of the insertion procedure almost immediately.

Corollary 4. *Inserting an element into an open-address hash table with load factor β requires at most $\frac{1}{1-\beta}$ probes on average, assuming uniform hashing.*

Proof: An element is inserted only if there is available space in the table, and thus $\beta < 1$.

Inserting a key requires an unsuccessful search followed by placement of the key in the first empty slot found. Thus, the expected number of probes is $\frac{1}{1-\beta}$. □

Theorem 5. *Given an open-address hash table with load factor* $\beta < 1$, *the expected number of probes in a successful search is at most*

$$\frac{1}{\beta}\ln\frac{1}{1-\beta}+\frac{1}{\beta},$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Proof : A search for a key K follows the same probe sequence as was followed when the element with key K was inserted. By Corollary 4, if K was the $(i+1)$ - st key inserted into the hash table, the expected number of probes made in a search for K is at most $\frac{1}{1-i/m}=\frac{m}{m-i}$. Averaging over all n keys in the hash table gives us the average number of probes in a successful search:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\beta}\left(H_m-H_{m-n}\right),$$

where $H_i=\sum_{j=1}^{i}\frac{1}{j}$ is the i-th harmonic number. Using the bounds $\ln i \le H_i \le \ln i + 1$, we obtain

$$\frac{1}{\beta}\left(H_m-H_{m-n}\right)\le\frac{1}{\beta}(\ln m+1-\ln(m-n))$$

$$=\frac{1}{\beta}\ln\frac{m}{m-n}+\frac{1}{\beta}$$

$$=\frac{1}{\beta}\ln\frac{1}{1-\beta}+\frac{1}{\beta}$$

for a bound on the expected number of probes in a successful search.□

Corollary 4 and Theorem 5 state that Insert and Access time will go up steeply as β approaches 1, and that open addressing works fine as long as β is bounded away from one, say $\beta \le 0.9$.

β	0.5	0.9	0.95	0.99	0.999
$\frac{1}{1-\beta}$	2	10	20	100	1000
$\frac{1}{\beta}\ln\frac{1}{1-\beta}+\frac{1}{\beta}$	3.38	3.66	4.20	5.66	7.9

For example, it follows that if the hash table is half full, the expected number of probes is less than 3.387; if the hash table is 90 percent full, the expected number of probes is less than 3.670.

§ 15. d - heaps

A heap (or, a priority queue) is a data structure for efficiently storing and manipulating a collection H of elements (or objects) when each element $i \in H$ has an associated real number, denoted by key (i). We want to perform the following operations on the elements in the heap H:

create (H). Create an empty heap H.

insert (i,H). Insert an element i in the heap.

find-min (i,H). Find an element i with the minimum key in the heap.

delete-min (i,H). Delete the element i with the minimum key from the heap.

delete (i,H). Delete an arbitrary element i from the heap.

decrease-key (i, value, H). Decrease the key of element i to a smaller value, denoted by value.

increase-key (i, value, H). Increase the key of element i to a larger value, denoted by value.

In this section we discuss the d-heap and binary heap data structures (the binary heap is a well-known special case of the d-heap with d=2). In the next section we describe a more efficient (and also more complex) heap known as the Fibonacci heap. In our discussion of heaps, we shall use the word „element" and „node" interchangeably.

Heaps find a variety of applications: Two such applications are Dijkstra's algorithm for the shortest path problem and Prim's algorithm for the minimum spanning tree problem.

We have seen another important application of heaps in the sorting of n numbers in a nondecreasing order: First, we create an empty heap. Then, one by one, we add n numbers to the heap by performing n insert operations, letting the key for the i – th entry be one of the numbers we wish to sort. Next, we repeat the following step iteratively: Select an element i with the minimum key using the operation find-min and then delete it from the heap using the operation delete-min. We terminate this procedure when the heap is empty. It is easy to see that we delete the elements from the heap in a nondecreasing order of their values.

Definition and properties of a d-heap. In a d - heap, we store the nodes of the heap as a rooted tree whose arcs represent a predecessor-successor (or parent-child) relationship.

We store the rooted tree using predecessor indices and sets of successors, as follows:

pred (i): the predecessor (or the parent) of node i in the d-heap. The root node has no predecessor, so we set its predecessor equal to zero.

succ (i): the set of successors (or children) of node i in the d-heap.

In the d-heap we define the depth of a node i (or its level number, the root having level zero) as the number of arcs in the unique path from node i to the root. For example, in the d - heap shown in Figure 1, node 5 has a depth of 0 and nodes 9, 8, and 15 have a depth of 1 etc.

Each node in the d-heap has d successors (with eventually a single exception on the last level) and these successors we assume to be ordered from left to right. We add nodes to the heap in an increasing order of depth values, and for the same depth value we add nodes from left to right, and this property we maintain inductively. We refer to this

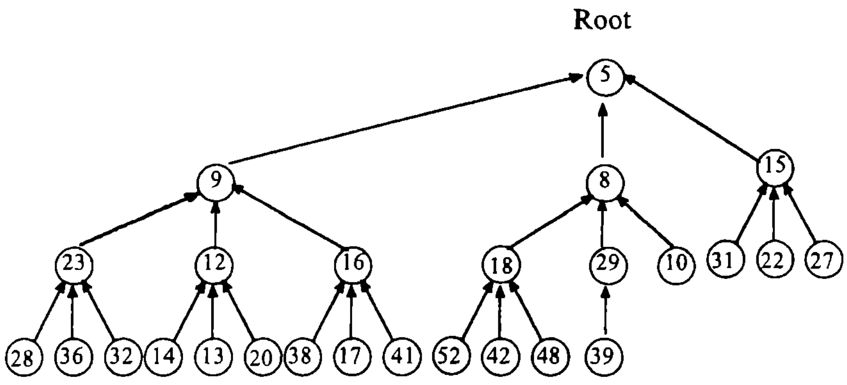


Figure 1: Example of a d-heap for $d = 3$

property as the contiguity property. In the example given in Figure 1 we assume for convenience that key $(i) = i$ for each $i = 1$ to 52 (in this particular example we have stored only a subset of the nodes in the heap). Note that nodes 12, 48 and 8 have the predecessors 9, 18, and 5, respectively.

The contiguity property implies the following results:

Property 1

(a) At most d^k nodes have depth k.

(b) At most $(d^{k+1}-1) / (d-1)$ nodes have depth between 0 and k.

(c) The depth of a d-heap containing n nodes is at most $\lfloor \log_d n \rfloor + 1$.

If we denote by h the height of the d - heap, then for each level k, $0 \le k \le h-1$ we have d^k nodes and on the level h we have at most d^h nodes, hence (a) is verified; the number of nodes having depth between 0 and k is at most $1+d+d^2+...+d^k = \dfrac{d^{k+1}-1}{d-1}$; we get $n \ge d^{h-1}$, hence $h \le \lfloor \log_d n \rfloor + 1$.

For the d-heap in Figure 1 this inequality is an equality since $3 = \lfloor \log_3 26 \rfloor + 1$.

Storing a d-heap. The structure of a d-heap permits us to store it as an array and manipulate it quite efficiently. We order the nodes in the increasing values of their depths, and we order the nodes with the same depth from left to right. We then store the nodes, in order, in an array DHEAP. For example, if we apply this method to the d - heap shown in Figure 1, then

DHEAP={5, 9, 8, 15, 23, 12, 16, 18, 29, 10, 31, 22, 27, 28, 36, 32, 14, 13, 20, 38, 17, 41, 52, 42, 48, 39}.

We also maintain an array position that contains the position of each node. For this example, position (8) = 3 and position (23) = 5. We maintain an additional parameter last that specifies the number of nodes stored in the array DHEAP. For this example, last = 26. This storage scheme has one rather nice property that permits us to easily access the predecessors and successors of any node:

Property 2

(a) The predecessor of the node in position i is contained in position $\lceil (i-1)/d \rceil$.

(b) The successors of the internal node in position i are contained in positions $id-d+2$, $id-d+3$,...,min ($id+1$, last).

For example, node 18 is in position 8, so its predecessor is in position $\lceil (8-1)/3 \rceil = 3$ and its successors are in positions $3 \cdot 8 - 3 + 2 = 23$ to $3 \cdot 8 + 1 = 25$.

This property implies that if we maintain the array DHEAP, we need not explicitly maintain the predecessor index and the set of successor indices of a node. We can compute these when required during the course of an algorithm. A heap always satisfies the following property, which we subsequently refer to as the heap order property.

Property 3. The key of node i in the heap is less than or equal to the key of each of its successors. That is, for each node i, key $(i) \le$ key(j) for every $j \in$ SUCC (i).

We notice that we might violate this property while performing a heap operation but will always satisfy it at the end of any heap operation.

The example shown in Figure 1 satisfies the heap order property if for every node in the heap we assume that key $(i) = i$.

The following result is an immediate consequence of the heap order property.

Property 4. The root node of the d-heap has the smallest key.

In the d - heap data structure, we reduce each heap operation into a sequence of a fundamental operation, each called swap (i,j). The operation swap (i, j) swaps (or interchanges) nodes i and j. Figure 2 gives an example of a swap. In terms of the array used to store a d-heap, as a result of applying swap (i,j), we store node i at the position where node j was stored, and store node j at the position where node i was stored. For example, if we perform swap (4,6) in the DHEAP = {5, 6, 7, 4, 8, 11, 12, 9}, as shown in Figure 2, then the new array representation of the d - heap becomes DHEAP = {5, 4, 7, 6, 8, 11, 12, 9}. Clearly, the swap operation requires $O(1)$ time.

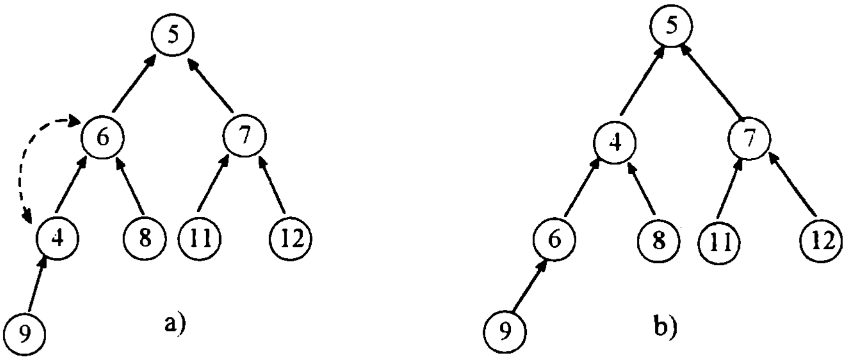


Figure 2. Example of swap (4,6): a) heap before the swap; b) heap after the swap.

Restoring the heap order property. In the course of applying an algorithm, we will frequently change the value of some key and so temporarily violate the heap order property. How can we restore this property?

Suppose that we decrease the key of some node i. Let j = pred (i). If after the change in the value of key (i), key (j) $\leq$ key (i), the heap still satisfies the heap order property and we are done. However, if key (j) > key (i), we need to restore the heap order property.

The procedure siftup (i) accomplishes this task.

```
procedure siftup (i);
    begin
        while i is not a root node and key (i) < key (pred (i))
            do swap (i, pred (i));
    end;
```

An inductive argument shows that at the termination of the siftup procedure, the heap satisfies the heap order property. The procedure siftup requires $O(\log_d n)$ time because each execution of the while loop decreases the depth of node i by one unit and, by Property 1, its original depth is $O(\log_d n)$.

Suppose next that we increase the key of some node i. If after the change in the value of key (i), key $(i) \leq$ key (j) for all $j \in$ SUCC (i), the heap still satisfies the heap order property and we are done; otherwise, we need to restore the heap order property.

The procedure siftdown (i) described below accomplishes this task. In the description we let minchild (i) denote the node with smallest key in SUCC (i).

```
procedure siftdown (i);
  begin
    while i is not a leaf node and key (i) > key (minchild (i)) do
        swap (i, minchild (i));
  end;
```

An inductive argument will again show that at the termination of the siftdown procedure, the heap satisfies the heap order property. The procedure requires $O(d \log_d n)$ time because each execution of the *while* loop increases the depth of node i by one unit and each execution requires $O(d)$ time to compute minchild (i).

Performing heap operations. We are now in a position to describe how we can perform various operations in the d-heap.

– find-min (i,H). The root node of the heap is the node with the minimum key and it is located at the first position of the array DHEAP. Therefore, this operation requires $O(1)$ time.

– insert (i,H). We increment last by one and store the new node i at the last position of the array DHEAP. Then we execute the procedure siftup (i) to restore the heap order property. Clearly, this operation requires $O(\log_d n)$ time.

– decrease-key (i, value, H). We decrease the key of node i and execute the procedure siftup (i) to restore the heap order property. This operations requires $O(\log_d n)$ time.

– delete- min (i, H). Clearly, node i is the root node of the heap. Let node j be the node stored at the last position of the array DHEAP. We first perform swap (i, j) and then decrease last by 1. Next, we perform siftdown (j) to restore the heap order property. Clearly, this heap operation requires $O(d \log_d n)$ time.

The remaining two heap operations, delete (i, H) and increase-key (i, value, H) can be performed in a similar way in $O(d \log_d n)$ time. We summarize our discussion as follows:

Theorem 1. *The d-heap data structure requires $O(1)$ time to perform the operation find-min, and $O(\log_d n)$ time to perform the operations* insert *and* decrease-key, *and $O(d \log_d n)$ time to perform the operations* delete-min, delete, and increase-key.

A binary heap is a d-heap with $d = 2$. For binary heaps, this theorem assumes the following special form.

Theorem 2. *The binary heap data structure requires $O(1)$ time to perform the operation find - min, and $O(\log n)$ time to perform each of the operations* insert, delete, delete-min, decrease-key, *and* increase-key.

As an example of applying heaps, consider a sorting algorithm; while sorting n numbers, we perform n inserts, n find-mins, and n delete-mins. Consequently, the running time of the sorting algorithm using d-heaps is $O(nd \log_d n)$, which is $O(n \log n)$ for any fixed value of d.

§ 16. Fibonacci heaps

The Fibonacci heap is a data structure that allows the heap operations to be performed more efficiently than d-heaps. This data structure performs the operations insert, find-min, and decrease-key in $O(1)$ amortized time and the operations delete-min, delete, and increase-key in $O(\log n)$ amortized time. Recall that the amortized complexity of an operation is the average worst-case complexity of performing that operation.

In other words, the amortized complexity of an operation is $O(g(n))$ if for a sequence of k (sufficiently large) operations, the total time required by these operations is $O(kg(n))$. Fibonacci heaps were developed by Fredman and Tarjan in 1984.

Some properties of Fibonacci numbers. Researchers have given the Fibonacci heap data structure its name because the proof of its time bounds uses properties of the well-known Fibonacci numbers. Before discussing the data structure, we first discuss these properties.

The Fibonacci numbers are defined recursively as $F(1) = 1$, $F(2) = 1$, and $F(k) = F(k-1) + F(k-2)$, for all $k \geq 3$. These numbers satisfy the following properties:

Proposition 1. The following properties hold:
(a) For $k \geq 3$, $F(k) \geq 2^{(k-1)/2}$
(b) $F(k) = 1 + F(1) + F(2) + ... + F(k-2)$.

Proof: The facts that $F(k) = F(k-1) + F(k-2)$ and $F(k-1) \geq F(k-2)$ imply that $F(k) \geq 2F(k-2)$.

Consequently, if k is odd, $F(k) \geq 2F(k-2) \geq 2^2F(k-4) \geq$ $\geq 2^3F(k-6) \geq ... \geq 2^{(k-1)/2}F(1) = 2^{(k-1)/2}$. If k is even, we argue by induction. The claim is true if $k = 4$. Suppose it is true for even numbers less than k. We have

$F(k) = F(k-1) + F(k-2) \geq 2^{(k-2)/2} + 2^{(k-3)/2}$

by the result for k odd and the induction hypothesis. But then $F(k) \geq 2^{(k-3)/2}[2^{1/2} + 1] > 2^{(k-1)/2}$ and so by induction the conclusion is true for all $k \geq 3$.

To prove part (b) let us observe that $1 + F(1) = F(2) + F(1) = F(3)$; $F(3) + F(2) = F(4)$;...; $F(k-1) + F(k-2) = F(k)$. □

Proposition 2. Suppose that a series of numbers $G(k)$ satisfies the properties that $G(1) = 1$, $G(2) = 1$, and $G(k) \geq 1 + G(1) + G(2) + ... + G(k-2)$ for all $k \geq 3$. Then $G(k) \geq F(k)$.

Proof: We prove inductively that $G(k) \geq F(k)$ for all k. This claim certainly is true for $k = 1$ and $k = 2$. Let us assume that it is true for all values of k from 1 through $q - 1$. Then $G(q) \geq 1 + G(1) + G(2) + ... + G(q-2) \geq 1 + F(1) + F(2) + ... + F(q-2) = F(q)$, the equality following from Proposition 1(b). □

Defining and storing a Fibonacci heap. As we noted earlier, a heap stores a set of elements, each with a real-valued key. A Fibonacci heap is a collection of directed rooted in-trees; each node i in the tree represents an element i and each arc (i,j) represents a predecessor-successor (parent-child) relationship: node j is the predecessor (parent) of node i. Figure 1 gives an example of a Fibonacci heap.

To represent a Fibonacci heap numerically (i.e., in a computer) and to manipulate it effectively, we need the following data structure:

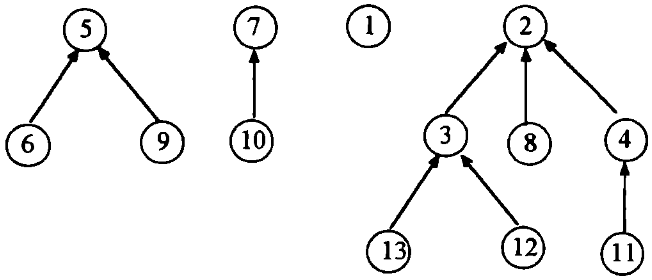


Figure 1: Fibonacci heap

pred (i): the predecessor (or the parent) of node i in the Fibonacci heap;

We refer to a node with no parent as a root node and we set its predecessor to zero. This convention permits us to determine whether a node is a root node or a nonroot node by looking at the node's predecessors index.

We also need the following data structures: SUCC (i): the set of successors (or children) of node i . We maintain this set as a doubly linked list.

rank (i): the number of successors of node i (i.e., rank (i) = = |SUCC (i)|).

minkey: the node with the minimum key.

Below is given this data structure for the rooted trees given in Figure 1.

i	1	2	3	4	5	6	7	8	9	10	11	12	13
pred (i)	0	0	2	2	0	5	0	2	5	7	4	3	3
SUCC (i)	∅	{3,8,4}	{13,12}	{11}	{6,9}	∅	{10}	∅	∅	∅	∅	∅	∅
rank (i)	0	3	2	1	2	0	1	0	0	0	0	0	0

We need additional data structures to support various heap operations; we will introduce these data structures later, when we require them.

A subtree hanging at any node i of any rooted tree is the subtree with root i; it contains the node i, its successors, successors of its successors, and so on.

For example, in Figure 1, the subtree hanging at node 5 contains the nodes 5, 6, and 9.

Linking and cutting. In using the Fibonacci heap data structure, we reduce each heap operation into a sequence of two fundamental operations: link (i,j) and cut (i).

We apply the operation link (i,j) to two (distinct) root nodes i and j of equal rank; it merges the two trees rooted at these nodes into a single tree. The operation cut (i) cuts node i from its predecessor and makes i a root node.

- link (i,j). If key $(j) \le$ key (i), then add arc (i,j) to the Fibonacci heap (thus making node j the predecessor of node i).

If key $(j) >$ key (i), then add arc (j,i) to the heap.

- cut (i). Delete arc $(i$, pred $(i))$ from the heap (thus making node i a root node).

We illustrate these two operations on the examples shown in Figure 2. For simplicity, we assume that for every node i, key $(i) = i$.

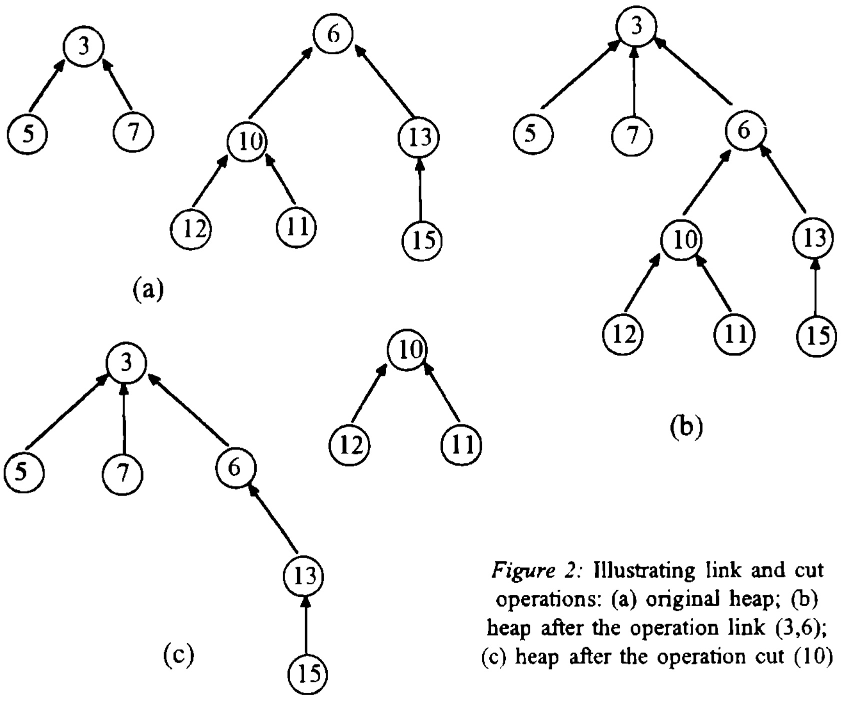


Figure 2: Illustrating link and cut operations: (a) original heap; (b) heap after the operation link (3,6); (c) heap after the operation cut (10)

Notice that the link operation increases the rank of node i or of node j by 1. Moreover, each of these operations changes the pred and SUCC and rank information for at most two nodes; consequently, we can perform them in $O(1)$ time. Later in this section we describe the additional data structures that we maintain to manipulate the Fibonacci heap effectively; we can also modify them in constant time as we perform a link and a cut operation. We have deduced the following property.

Proposition 3. *The operations link(i,j) and cut(i) require $O(1)$ time to execute.*

While manipulating the Fibonacci heap data structure, we perform a sequence of links and cuts. There is a close relationship between the number of links and cuts. To observe this relationship, consider a function φ defined as the number of rooted trees in Fibonacci heap. Each link operation decreases φ by 1 and each cut operation increases φ by 1.

Initially we have n trees, each consisting of the root. The following result holds:

Proposition 4. *The number of links is at most $n-1$ plus the number of cuts.*

Invariants in Fibonacci heaps. The Fibonacci heap data structure maintains a set of rooted trees that change dynamically as we perform various linking and cutting operations. These rooted trees satisfy certain invariants that are essential for deriving the claimed time bounds for the heap operations. The nodes of the Fibonacci heap always satisfy the heap order property (which states that the key of a node is less than or equal to the keys of its successors).

The Fibonacci heap also satisfies the following two properties:

Property 1. *Each nonroot node has lost at most one successor after becoming a nonroot node.*

Property 2. *No two root nodes have the same rank.*

As before, although we might violate these invariants at intermediate steps of some heap operations, the heap will satisfy them at the conclusion of each heap operation.

One important consequence of properties 1 and 2 is that the maximum possible rank of any node is $2\log n + 1$. We establish this result next.

Proposition 5. *Any node in the Fibonacci heap with n nodes has rank at most* $2\log n + 1$.

Proof: Let $G(k)$ denote the minimum number of nodes contained in a subtree hanging at a node of rank k in a Fibonacci heap. We shall prove that $G(k) \geq F(k)$.

Indeed, let w be a node in a Fibonacci heap with rank k. Arrange the successors of node w in the same order in which the previous operations linked them to w, from the earliest to the latest. We claim that the rank of the i – th successor of w is at least $i - 2$. To establish this result, let y be the i – th successor of node w and consider the moment when y was linked to w. Just before this link operation, w had at least $i - 1$ successors since y is the i – th successor (It might have had more than $i - 1$ successors at that time, some having been cut since then). Since at the time of this link operation nodes y and w both have the same rank, node y had at least $i - 1$ successors just before we performed this link operation.

Furthermore, notice that since that time node y has lost at most one successor (from property 1). Therefore, node y (which is the i – th successor of node w) has rank at least $i - 2$. As a result, the subtree hanging at node w contains at least
$1 + 2 + G(1) + G(2) + ... + G(k-2) > 1 + G(1) + G(2) + ... + G(k-2)$
nodes.

To summarize, we have shown that

$$G(k) \geq 1 + G(1) + G(2) + ... + G(k-2),$$

which in view of Proposition 2 implies that $G(k) \geq F(k)$. Since no subtree can contain more than n nodes, we have

$$n \geq G(k) \geq F(k) \geq 2^{(k-1)/2}$$

by Proposition 1, which implies that $k \leq 2\log n + 1$. □

The following property follows directly from Property 2 and Proposition 5.

Proposition 6. *A Fibonacci heap with n nodes contains at most* $1 + 2\log n$ *rooted trees*.

We next discuss how to restore properties 1 and 2 if they become violated at intermediate steps of a heap operation.

Restoring Property 1. To restore Property 1, we maintain an additional index lost (i) for every node i, defined as follows.

lost (*i*): For a nonroot *i*, lost (*i*) represents the number of successors the node has lost after it became a nonroot node. For a root node *i*, lost(*i*) = 0.

Suppose that while manipulating a Fibonacci heap, we perform the operation cut (*i*). We refer to this cut as the actual cut. Let *j* = pred (*i*). In this operation, node *j* loses a successor. If node *j* is a nonroot node, we increment lost (*j*) by 1. If lost (*j*) becomes two, Property 1 requires that we make node *j* a root node. In that case we perform cut (*j*) and make *j* a root node. Let *k* = pred (*j*). This cut increases lost (*k*) by 1. If *k* is a nonroot node and lost (*k*) = 2, we must make it a root node as well, and so on.

Thus an actual cut might lead to several cuts due to a cascading effect: We keep performing these cuts until we reach a node that has not lost any successor so far or is a root node. We refer to these additional cuts that are triggered by an actual cut as cascading cuts, and the entire sequence of steps following an actual cut as multicascading.

We illustrate this process on the Fibonacci heap shown in Figure 3(a) where we suppose that lost (11) = 1 and lost (8) = 1. Suppose that we cut node 17 from its predecessor.

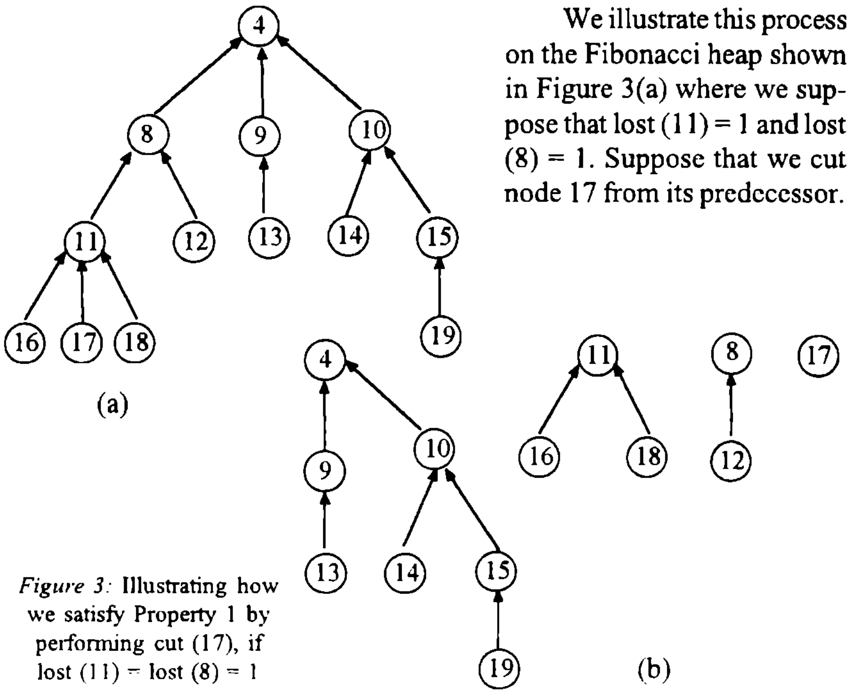


Figure 3: Illustrating how we satisfy Property 1 by performing cut (17), if lost (11) = lost (8) = 1

This operation also requires that we also cut nodes 11 and 8 from their predecessors. Figure 3(b) shows the resulting Fibonacci heap that satisfies Property 1. We now summarize the preceding discussion.

Proposition 7. *If we perform an actual cut in a Fibonacci heap, we might also need to perform several cascading cuts so that the heap again satisfies Property 1; the time needed for these operations is proportional to the total number of cuts performed.*

Suppose that we perform a number of actual cuts at different times while manipulating a Fibonacci heap and that these cuts cause additional cascading cuts. What is the relationship between the total number of actual cuts and the total number of cascading cuts? We shall show that the total number of cascading cuts cannot exceed the total number of actual cuts. To prove this result, consider the potential function $\varphi = \sum_{i \text{ in heap}} \text{lost}(i)$. Suppose that we perform cut (i) and j = pred (i). This operation sets lost (i) to zero and increases lost (j) by one if j is a nonroot node. If the cut is an actual cut, lost (i) equals 0 or 1 before the cut, and if it is a cascading cut, lost (j) equals 1 before the cut.

Therefore, an actual cut increases lost (i) + lost (j), and hence the value of the potential function φ by at most one, and a cascading cut decreases lost (i) + lost (j) by at least one. If we start with a potential value of zero, the total decreases in the potential function are bounded by the total increases. The following property is now apparent.

Proposition 8. *The total number of cascading cuts is less than or equal to the total number of actual cuts.*

Restoring Property 2. The Property 2 requires that no two root nodes have the same rank. To maintain this property, we need the following index for every possible rank $k = 1, 2, \ldots, 2\lfloor \log n \rfloor + 1$:

bucket (k): If the Fibonacci heap contains no root node with rank equal to k, then bucket (k) = 0; and if some root node i has a rank equal to k, then bucket (k) = i.

Suppose that while manipulating a Fibonacci heap, we create a root node j of rank k and the heap already contains another root node i with the same rank. Then we repeat the following procedure to restore Property 2.

We perform the operation link (i,j), which merges the two rooted trees into a new tree of rank $k+1$. Suppose that node l is the root of the new tree. Then by looking at bucket $(k+1)$, we check to see whether the heap already contains a root node of rank $k+1$. If not, we are done. Otherwise, we perform another link operation to create another rooted tree of rank $k+2$ and check whether the heap already contains a root node of rank $k+2$.

We repeat this process until we satisfy Property 2. We refer to this sequence of steps following the addition of a new root as multilinking.

We illustrate this process of re-establishing Property 2 on a numerical example. Consider the Fibonacci heap shown in Figure 4(*a*), assuming that the key of node i equals i. Suppose that we add a new rooted tree containing a singleton node 10. The heap already contains another root node of rank 0, namely node 9. Thus we perform a link operation on nodes 9 and 10, obtaining the rooted trees shown in Figure 4(*b*). Now two trees in the heap , with roots 7 and 9, have rank 1.

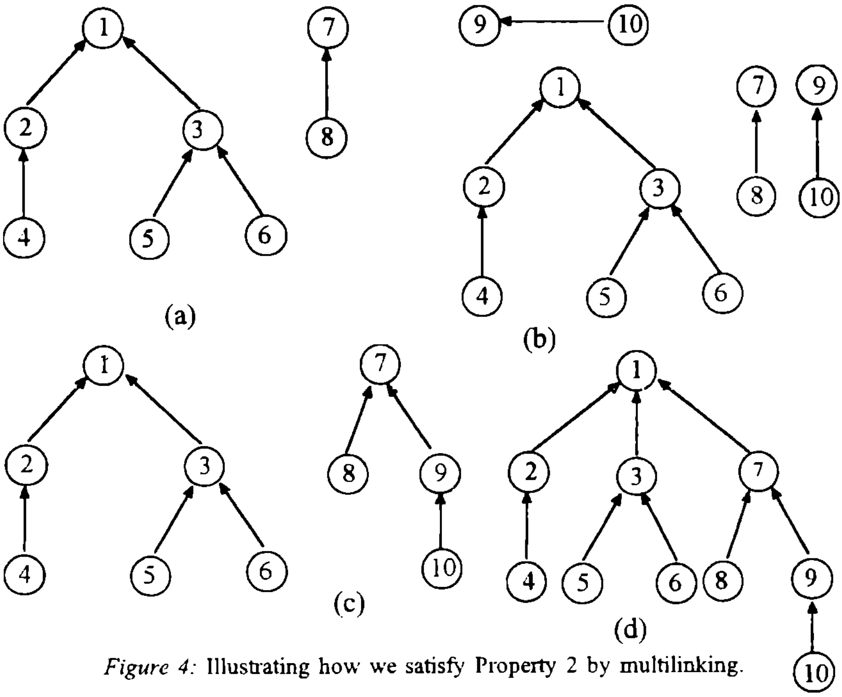


Figure 4: Illustrating how we satisfy Property 2 by multilinking.

We perform another link operation, producing the structure shown in Figure 4(*c*). But now two trees, with roots 1 and 7, have rank 2. We perform another link operation, producing the structure shown in Figure 4(*d*). At this point, Property 2 is fullfiled and we terminate.

We summarize the preceding discussion as follows:

Proposition 9. *If we add a new rooted tree to a Fibonacci heap, we might need to perform several links to restore Property* 2; *the time needed for these operations is proportional to the total number of links.*

Heap operations. Finally, we show how we perform various heap operations using the Fibonacci heap data structure and indicate the amount of time they take.

• find-min (i,H). We simply return i = minkey, since the variable minkey contains the node with the minimum key.

• insert (i,H). We create a new singleton root node i and add it to H. After we have performed this operation, the heap might violate Property 2, in which case we perform multilinking to restore the invariant.

• decrease-key (i, value, H). We first decrease the key of node i and set it equal to value. After we have decreased the key of node i, every node in the subtree hanging at node i still satisfies the heap order property; the predecessor of node i might, however, violate this property. Let j = pred (i). If key (j) ≤ value, we are done. Otherwise, we perform an actual cut, cut (i), make node i a root node, and update minkey. After we have performed the cut, the heap might violate Property 1, so we perform multicascading to restore this invariant. The resulting cascading cuts generate new rooted trees whose roots we store in a list, LIST. Then one by one, we remove a root node from LIST, add it to the previous set of roots, and perform multilinking to satisfy Property 2. We terminate when LIST becomes empty.

• delete-min (i,H). We first set i = minkey. Then one by one, we scan each node $l \in$ SUCC (i), perform an actual cut, cut (l), and update minkey.

We apply multilinking after performing each such actual cut. When we have cut each node in SUCC (i), we scan through all root nodes (which are stored in bucket (k), for $k = 0,1,\ldots,2\lfloor \log n \rfloor + 1$), identify the root node h with minimum key, and set minkey = h.

Recall that $|\text{SUCC}(i)| \leq 2 \log n + 1$, because Proposition 5 implies that each node has at most $2 \log n + 1$ successors. Therefore, the delete-min operation performs $O(\log n)$ actual cuts, followed by a number of cascading cuts and links. Then we scan through $O(\log n)$ root nodes to identify the root with the minimum key. Summarizing, for the heap operation: find-min (i,H), step Return i = minkey, time taken is $O(1)$; for insert (i,H); add a new singleton node i, time: $O(1)$; for decrease-key (i, value, H); decrease the key of node i: $O(1)$; perform cut (i) and update minkey: $O(1)$; for delete-min (i,H): perform cut (l) for each node $l \in$ SUCC (i): $O(\log n)$ and compute minkey by scanning all root nodes: time taken $O(\log n)$.

We now consider the time required for multicascading and multilinking. We claim that the time taken by these two steps is $O(\log n)$, so we can ignore this time further.

To establish this claim, we use the following facts: (1) Proposition 8, which states that the number of cascading cuts is no more than the number of actual cuts; and (2) Proposition 4, which states that the number of links is no more than $n - 1$ plus the number of actual and cascading cuts. Consequently, if we perform a sufficiently large number of operations (relative to n), the number of actual cuts will count the number of links and cascading cuts within a constant factor; therefore, in the big O notation, we can ignore the time required for the latter operations.

In a similar way we can prove that the operations delete and increase-key also require $O(\log n)$ amortized time.

We summarize the discussion in this section as follows:

Theorem 10. *The Fibonacci heap data structure requires $O(1)$ amortized time to perform each of the operations* insert, find-min, *and* decrease-key, *and $O(\log n)$ amortized time to perform each of the operations* delete-min, delete, *and* increase-key.

It is clear that the role of data structure is critical in designing efficient algorithms and in writing computer programs for implementing algorithms. We illustrate this idea further by considering Dijkstra's algorithm for the shortest path problem.

Let $G = (V,E)$ be an undirected graph and let l be a function assigning a nonnegative length to each edge. Extend l to domain $V \times V$ by defining $l(v,v) = 0$ and $l(u,v) = \infty$ if $(u,v) \notin E$. Define the length of a path

$p = e_1 e_2 \ldots e_n$ written as a sequence of edges to be $l(p) = \sum_{i=1}^{n} l(e_i)$. For $u,v \in V$, define the distance $d(u,v)$ from u to v to be the length of a shortest path from u to v, or ∞ if no such path exists. Notice that $d(u,u) = 0$ for every $u \in V$.

The single-source shortest path problem is to find, given $s \in V$, the value of $d(s,u)$ for every other vertex u in the graph.

There is an algorithm due to Dijkstra that solves this problem; we will give an $O(m + n \log n)$ implementation using Fibonacci heaps, where $m = |E|$ and $n = |V|$.

This algorithm is a type of greedy algorithm: it builds a set X vertex by vertex, always taking vertices closest to X.

Dijkstra's Algorithm

```
    $X \leftarrow \{s\}$;
    $D(s) \leftarrow 0$;
for each $u \in V \setminus \{s\}$ do
    $D(u) \leftarrow l(s,u)$;
while $X \neq V$ do
        let $u \in V \setminus X$ such that $D(u)$ is minimum
        $X \leftarrow X \cup \{u\}$; for each edge $(u,v)$ with $v \in V \setminus X$ do
        $D(v) \leftarrow \min(D(v), D(u) + l(u,v))$
end while
```

The final value of $D(u)$ is $d(s,u)$. This algorithm can be proved correct by showing that the following two invariants are maintained by the while loop:

- for any u, $D(u)$ is the distance from s to u along a shortest path through only vertices in X;
- for any $u \in X$, $v \notin X$, $D(u) \leq D(v)$ holds.

Dijkstra's algorithm performs n inserts, n find - mins, n delete - mins, and at most m decrease - key operations.

The time requirements of the heap operations imply that the algorithm requires $O(m + n \log n)$ time, plus the time for $O(n \log n)$ actual cuts, plus the time for multicascading and multilinking. Using the facts that the number of cascading cuts and links are no more than twice the number of actual cuts, and that each actual cut requires $O(1)$ time, we immediately see that the shortest path algorithm runs in $O(m + n \log n)$ time.

The following algorithm, known as Prim's algorithm, produces a minimum spanning tree T in a connected undirected graph with edge weights.

Initially, we choose an arbitrary vertex and let T be the tree consisting of that vertex and no edges. We then repeat the following step $n-1$ times: find an edge of minimum weight with exactly one endpoint in T and include that edge in T.

Using the Fibonacci heap, there exists an implementation of this algorithm that runs in time $O(m + n \log n)$.

§ 17. Splay trees

A splay tree is a data structure invented by Sleator and Tarjan (1983) for maintaining a set of elements drawn from a totally ordered set. Splay trees are a particular kind of self-organizing tree structure.

The most interesting aspect of the structure is that, unlike balanced tree schemes such as AVL trees, it is not necessary to rebalance the tree explicitly after every operation – it happens automatically.

Data are represented at all nodes of a splay tree; they are distinct and drawn from a totally ordered set U having a weight function $w : U \rightarrow \mathbf{N}^*$. The data items will always be maintained in inorder; that is, for any node x, the elements occupying the left subtree of x are all less than x, and those occupying the right subtree of x are all greater than x. Splay trees support the following operations:

· Access (x, T): if item x is in tree T then return a pointer to its location, otherwise return nil.

· Insert (x, T): insert x into tree T and return the resulting tree (i.e. a pointer to its root) .

· Delete (x, T): delete x from tree T and return the resulting tree.

· Join 2 (T_1, T_2): return a tree representing the items in T_1 followed by the items in T_2, destroying T_1 and T_2 (this assumes that all elements of T_1 are smaller than all elements of T_2).

· Join 3 (T_1, x, T_2): return a tree representing the items in T_1 followed by x, followed by the items in T_2, destroying T_1 and T_2 (this assumes that all elements of T_1 are smaller than x which in turn is smaller than all items in T_2).

· Split (x, T): return two trees T_1, and T_2: T_1 contains all items of T smaller than x and T_2 contains all items of T larger than x (this assumes that x is in tree T); tree T is destroyed.

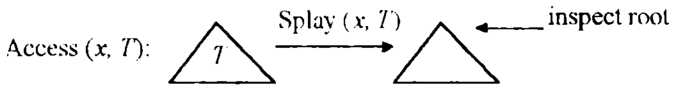
· Change weight (x, T, δ): change the weight of element x by δ. It is assumed that x belongs to tree T. The operation returns a tree representing the same set of elements as tree T.

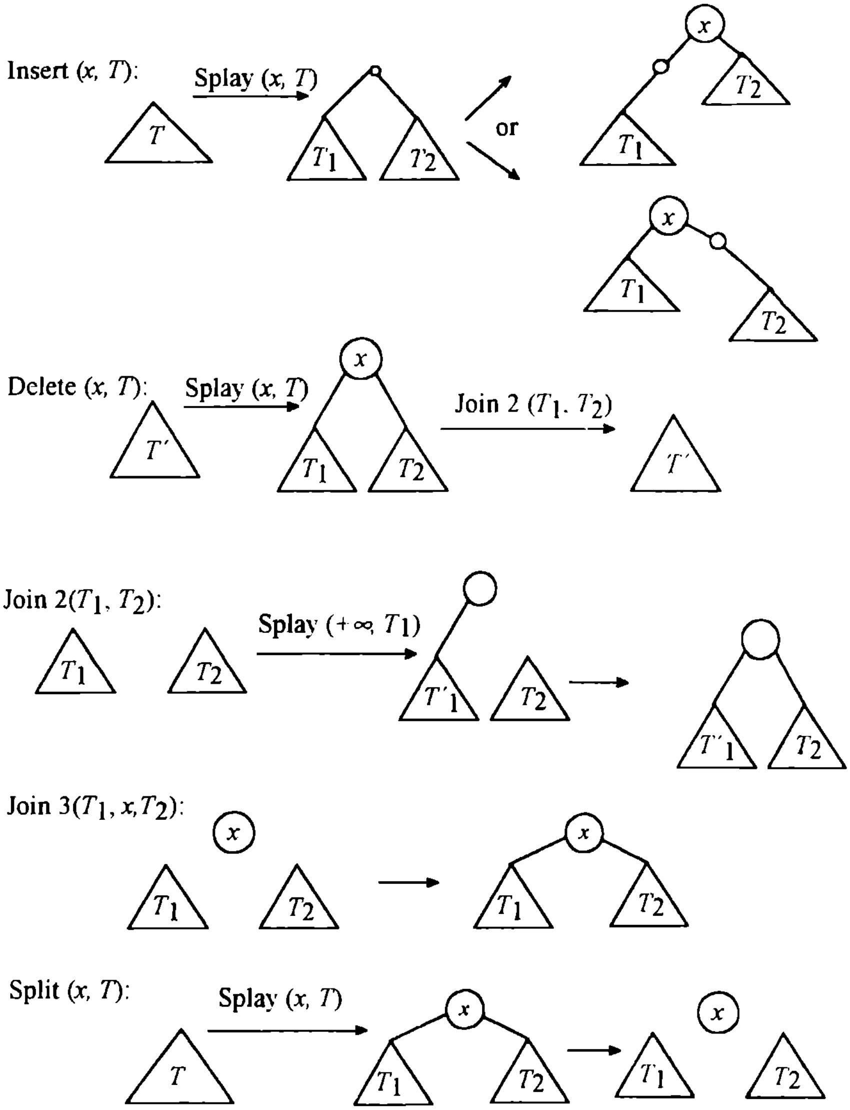
All these operations are implemented in terms of a single basic operation, called operation Splay, which is unique to splay-trees and gives them their name:

Splay (x, T): returns a tree representing the same set of elements as T. If x is in the tree, then x becomes the root. If x is not in the tree then either the immediate predecessor x^- of x ($x^- = \max \{k \in T \mid k < x\}$) or the immediate successor x^+ of x in T becomes the root. ($x^+ = \min \{k \in T \mid k > x\}$). This operation destroys T.

All of the operations mentioned above can be performed with a constant number of splays in addition to a constant number of other low-level operations such as pointer manipulations and comparisons and can be reduced to operation Splay. For example, in order to do Access (x, T) we do Splay (x, T) and then inspect the root. Notice that x is stored in tree T iff x is stored in the root of the tree returned by Splay (x, T). To do Insert (x, T) we first do Splay (x, T), then split the resulting tree into one containing all items less than x and one containing all items greater than x, one breaks one of the links leaving the root and then build a new tree with the root storing x and the two trees being the left and right subtree.

To do Delete (x, T) we do Splay (x, T), discard the root and join the two subtrees T_1, T_2 by Join 2 (T_1, T_2). To do Join 2 (T_1, T_2) we do Splay ($+\infty$, T_1) where $+\infty$ is assumed to be larger than all elements of U and then make T_2 the right son of the root of the resulting tree. Notice that Splay ($+\infty$, T_1) makes the largest element of T_1 the root and hence creates a tree with an empty right subtree. To do Join 3 (T_1, x, T_2) we make T_1 and T_2 the subtrees of a tree with root x. To do Split (x, T) we do Splay (x, T) and then break the two links leaving the root. Finally, to do Change weight (x, t, δ) we do Splay (x, T). The following diagram illustrates how all other operations are reduced to Splay:





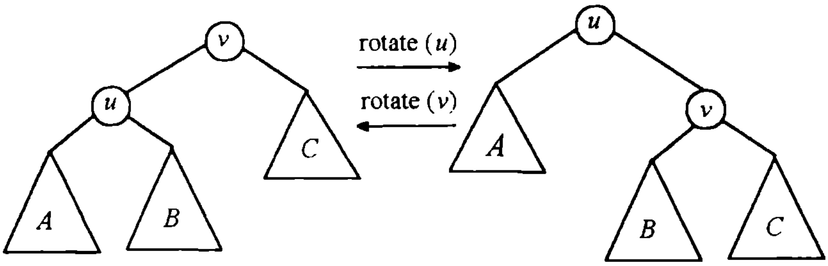
Change weight (x, T, δ): Splay (x, T)

It remains to describe the operation Splay (x, T). We first locate the node which is made the root by the splay operation.

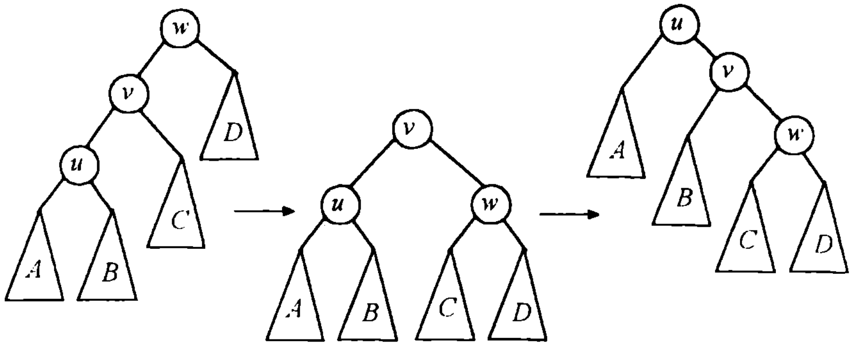
```
$v \leftarrow$ ROOT;
while $v \neq \Lambda$ and key ($v$) $\neq x$
 do $u \leftarrow v$;
        if $x <$ key ($v$)
        then $v \leftarrow$ LLINK ($v$) else $v \leftarrow$ RLINK ($v$) fi
od
if $v \neq \Lambda$ then $u \leftarrow v$ fi
```

If x is stored in tree T then clearly u points to the node containing x. If x is not stored in tree T then u points to the last non-nil node on the search path, i.e. to a node containing either the predecessor or the successor of x. We will make node u the root of the tree by a sequence of rotations. More precisely, we move u to the root by a sequence of splay steps. For the splay step we distinguish three cases.

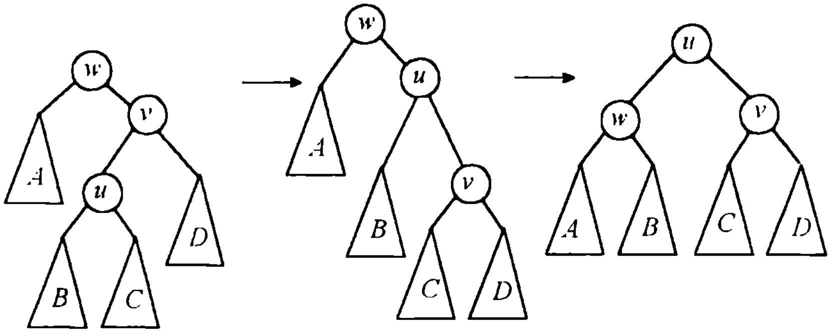
Case 1: Node u has a father but no grandfather. Then we perform a rotation at v = father (u) and terminate the splay operation. The operation rotate (u) moves u up and v down and changes a few pointers. A very simple but important observation to make at this point is that the rotate operation preserves inorder numbering.



Case 2: If u has a father v and a grandfather, and if u and v are either both left children or both right children, we first rotate (v) and then rotate (u).



Case 3: If u has a father v and a grandfather, and if one of u, v is a left child and the other is a right child, we first rotate (u) and then rotate (u) again.

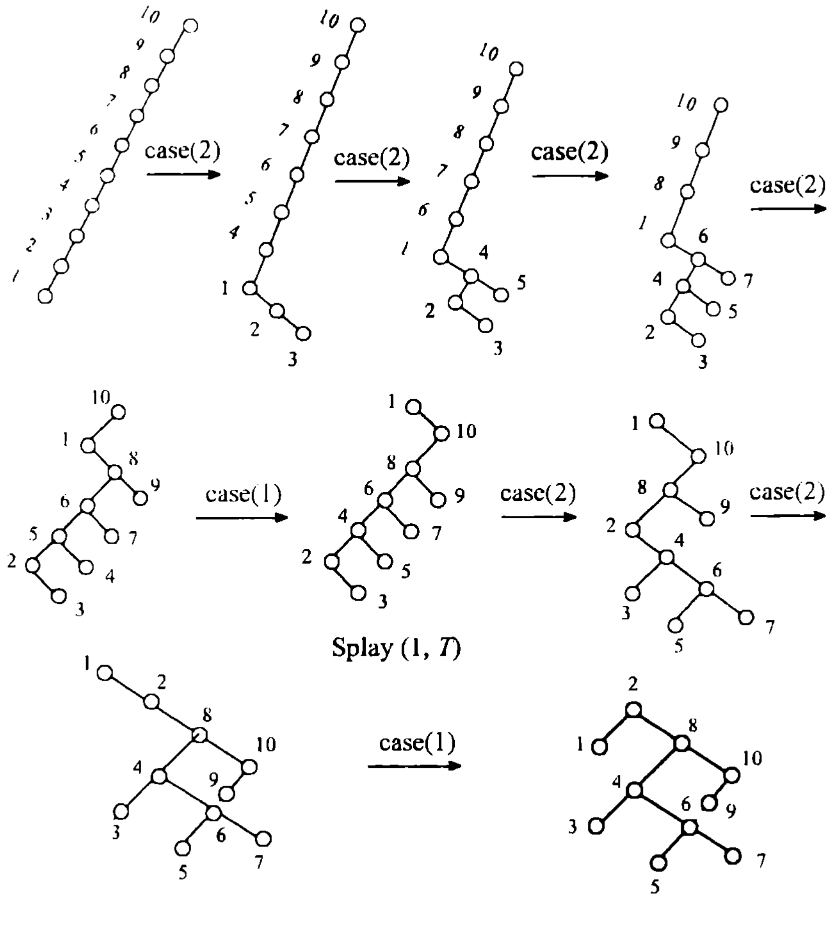


Remark: *It is very important that the rotations in case 2 are applied in this unconventional order. Note that this order moves node u and its subtrees A and B closer to the root. This is also true for case 3 and will be very important for the analysis.*

This finishes the description of the splay operation. The following figure shows an example.

Apply Splay (1, T) to the following tree T:

Applying splay to node 2 in the tree denoted by Splay (1, T) we get another tree which is denoted Splay (2, Splay (1, T)) and so on.



Splay (1, T)

Splay (2, Splay (1, T))

Analysis. Before we can analyse splay trees we need some more notation. For $x \in U$ we write $w(x)$ to denote the weight of element x ($w(x) > 0$). For v a node of a splay tree we write $\text{tw}(v)$ to denote the sum of the weights of all elements which are stored in descendants of node v. Notice that the total weight of a node changes over time. If T is a splay tree we write $\text{tw}(T)$ instead of $tw(\text{root }(T))$.

Let

$$\text{bal}(T) = \sum_{v \text{ node of } T} \lfloor \log tw(v) \rfloor$$

or with the definition of a rank $r(v) = \lfloor \log tw(v) \rfloor$ of a node v, bal $(T) = \sum_{v \text{ node of } T} r(v)$. We define $r(T) = \lfloor \log tw(T) \rfloor$. For the following lemma we assume that each splay step takes time 1 (for the constant number of low - level operations such as pointer manipulations and comparisons).

Lemma 1. *The amortized cost of operation* Splay (x, T) *is at most* $1 + 3\,(r(T) - r(u))$ *where u is the node of T which is made the root by the splay operation.*

Proof: We shall prove that if u, v and w are as defined in the figures illustrating the three cases of the splay step, then the amortized cost of case 1 is at most $1 + 3\,(r(v) - r(u))$, and of cases 2 and 3 is at most $3(r(w) - r(u))$.

Notice also that the amortized cost of operation Splay (x, T) is equal to the number of splay steps plus the difference in balance of the tree after this operation.

In this proof we use $r'(u)$, $r'(v)$, $r'(w)$ to denote the ranks of the various nodes after the splay step. Notice that $r'(u) = r(v)$ în case 1 and $r'(u) = r(w)$ in cases 2 and 3.

We will frequently use the following simple observation about ranks. If z is a node with sons z_1 and z_2 and $r(z_1) = r(z_2)$, then $r(z) \geq r(z_1) + 1$. This follows since $r(z_1) = r(z_2) = k$ implies

$$2^k \leq tw(z_1),\ tw(z_2) < 2^{k+1}$$

and hence $tw(z) \geq tw(z_1) + tw(z_2) \geq 2^{k+1}$.

Thus $r(z) \geq r(z_1) + 1$. Also $r(\text{father}(z)) \geq r(z)$ for all nodes z.

We are now ready to discuss the various cases of the splay step. Notice that the actual cost is one in all three cases.

Case 1: The amortized cost of this splay step is

$$\begin{aligned} & 1 + r'(v) + r'(u) - r(v) - r(u) \\ & = 1 + r'(v) - r(u), \text{ since } r'(u) = r(v) \\ & \leq 1 + r(v) - r(u)\,, \text{ since } r'(v) \leq r'(u) = r(v) \\ & \leq 1 + 3(r(v) - r(u))\,, \text{ since } r(v) \geq r(u). \end{aligned}$$

Case 3: The amortized cost of this splay step is

$$1 + r'(u) + r'(v) + r'(w) - r(u) - r(v) - r(w) =$$
$$= 1 + r'(v) + r'(w) - r(u) - r(v), \text{ since } r'(u) = r(w).$$

Assume first that $r(w) > r(u)$. Then we conclude further, using $r'(v) \leq r'(u) = r(w)$, $r'(w) \leq r'(u) = r(w)$, $r(v) \geq r(u)$ and $1 \leq r(w) - r(u)$, that the amortized cost is bounded by

$$r(w) - r(u) + r(w) + r(w) - r(u) - r(u) = 3(r(w) - r(u)).$$

This finishes the proof if $r(w) > r(u)$.

Assume next that $r(w) = r(u)$. Then $r(w) = r(v) = r(u) = r'(u)$. Also $r'(w) \leq r'(u) = r(u)$ and $r'(v) \leq r'(u) = r(u)$. Hence $r'(w) - r(u) \leq 0$ and $r'(v) - r(v) \leq 0$ since the subtree with root v after rotation contains subtrees C and D and before rotation it had contain B, C, D and node u.

If these two inequalities are equalities we have $r'(w) = r'(v)$ and by the observation above $r'(u) \geq r'(w) + 1$, which implies that $r'(w) - r(u) = r'(w) - r'(u) < 0$ and this inequality shows that at least one of the inequalities $r'(w) - r(u) \leq 0$ and $r'(v) - r(v) \leq 0$ is strict. In this case

$$1 + r'(v) - r(v) + r'(w) - r(u) \leq 0 = 3(r(w) - r(u)).$$

Case 2: The amortized cost of this splay step is

$$1 + r'(u) + r'(v) + r'(w) - r(u) - r(v) - r(w) = 1 + r'(v) +$$
$$+ r'(w) - r(u) - r(v), \text{ since } r'(u) = r(w).$$

Assume first that $r(w) > r(u)$. Then we conclude further, using $1 \leq r(w) - r(u)$, $r(u) \leq r(v) \leq r(w)$ and $r'(w) \leq r'(v) \leq r'(u) = r(w)$, that the amortized cost is

$$\leq r(w) - r(u) + r(w) + r(w) - r(u) - r(u) = 3(r(w) - r(u)).$$

Assume next that $r(w) = r(u)$. Then $r(w) = r(v) = r(u) = r'(u)$ and $r'(u) \geq r'(v) \geq r'(w)$.

If $r'(u) > r'(w)$ then $r'(w) - r'(u) \leq -1$ or $r'(w) - r(u) \leq -1$ since $r'(u) = r(u)$ and $r'(v) \leq r'(u) = r(v)$. Hence the amortized cost is bounded by zero ($= 3(r(w) - r(u))$ and we are done. So assume $r'(u) = r'(w)$. Consider the middle tree in the figure illustrating case 2. We use $F(u)$, $F(v)$, $F(w)$ to denote the ranks in this tree. We have $F(u) = r(u)$, $F(w) = r'(w)$ and $F(v) = r(w)$. If $r'(w) = r'(u)$ then

$F(w) = r'(w) = r'(u) = r(w) = r(u) = F(u)$ and hence $F(v) > F(w)$ and therefore $r'(u) > r'(w)$, a contradiction. Hence $r'(u) > r'(w)$ always. This finishes the proof in case 2.

The proof of lemma 1 is completed by summing the costs of the individual splay steps. Notice that the sum telescopes.□

The amortized cost of the other operations is now readily computed.

Theorem 2. *The amortized cost of* Splay (x, T) is O (log $tw(T)$ / $tw(x)$); *The amortized cost of* Access (x, T) is O (log $tw(T)$ / $tw(x)$);

The amortized cost of Delete (x,T) *is* $O\left(\log\frac{tw(T)}{\min\left(tw(x),tw\left(x^-\right)\right)}\right)$, *where* x^- *is the predecessor of* x *in tree* T; *The amortized cost of Join* 2 (T_1, T_2) *is* $O\left(\log\frac{tw\left(T_1\right)+tw\ \left(T_2\right)}{tw(x)}\right)$ *where* x *is the largest element in tree* T_1; *The amortized cost of* Join 3 (T_1, x, T_2) *is* $O\left(\log\frac{tw\left(T_1\right)+tw\left(T_2\right)}{w(x)}\right)$;

The amortized cost of Insert (x, T) *is* O (log$tw'(T)$ / min $(tw(x^-)$, $tw(x^+)$, $w(x)$)), *where* x^- *is the predecessor*, x^+ *is the successor of* x *in the final tree and* $tw'(T)$ *is the weight of* T *after the operation*;

The amortized cost of Split (x, T) *is* O (log $tw(T)$ / $tw(x)$); *The amortized cost of* Change weight (x, T, δ) is O (log $(tw(T) + \delta)$ / $tw(x)$).

Proof: The bound for the Splay operation was shown in lemma 1. Operation Access is identical to Splay and the cost of Split (x, T) is the cost of Splay (x, T) plus 1. The bound on the amortized cost of Join 2 (T_1, T_2) is O (log $tw(T_1)$ / $tw(x)$ + 1 + log $(tw(T_1) + tw(T_2))$ − log $tw(T_1)$) = O (log $(tw(T_1) + tw(T_2))$ / $tw(x)$) where x is the largest element in tree T_1. In this bound, the first term accounts for the cost of Splay $(+\infty, T_1)$ and the second and third term account for the actual cost of making a son of x and the change in the rank of x caused by making T_2 a son of x. The cost of Join 3 (T_1, x, T_2) is O (1 + log $(tw(T_1) + w(x) + tw(T_2))$ − log $w(x)$)) where the last term accounts for the rank change of x. The amortized cost of Delete (x, T) is O (log $tw(T)$ / $tw(x)$ +log $tw(T)$ / $tw(x^-)$) where the first term accounts for the cost of Splay (x, T) and the second term accounts for the Join 2 operation. The cost of Insert is O (log $tw(T)$ / min ($tw(x^-)$, $tw(x^+)$) + 1 + log ($tw(T)$ + $w(x)$) – log $w(x)$) = O (log ($tw(T)$ + $w(x)$) / min ($tw(x^-)$, $tw(x^+)$, $w(x)$)) where the first term accounts for the splay operation and the last term accounts for the rank change of x. Finally, the cost of Change weight (x, T, δ) is O (log $tw(T)$ / $tw(x)$) + O (log ($tw(T)$ + δ) – – log tw (T)) = O (log ($tw(T)$ + δ) / $tw(x)$) where the first term accounts for the Splay operation and the second term accounts for the rank change of x.

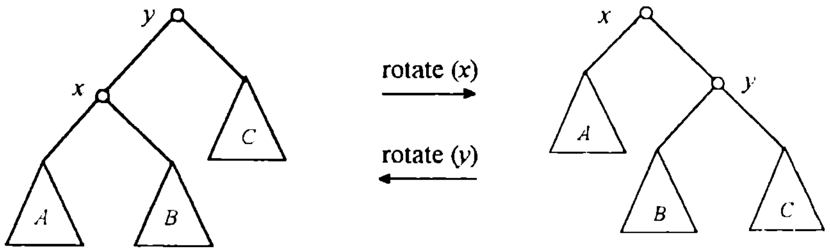
In order to complete the analysis, we must consider the effects of insertion, deletion, join and split on the ranks of nodes. To simplify matters, let us define the individual weight of every item to be 1. Then every node has a rank in the range $\{0,1,\ldots,\lfloor \log n \rfloor\}$, and lemma 1 gives a bound of 3 $\lfloor \log n \rfloor$+ 1 for splaying. The actual cost of a sequence of m operations $Op_1,\ldots,Op_m$ starting with weight function w: $U \to \mathbf{R}_+$, $w(x) = 1$ for every $x \in U$ and a forest of single node trees is bounded by the sum of the amortized costs of the operations and the initial balance $\sum_{x \in U} \lfloor \log w(x) \rfloor = 0$. By theorem 2 the amortized cost of a single operation is O (log n). Thus we have the following corollary:

Corollary 3. *The total time required for a sequence of m sorted set operations using splay (self – adjusting) trees, starting with no sorted sets is O* (*m* log *n*), *where n is the number of items.*

§ 18. Random search trees

In this lecture we will describe a probabilistic data structure that allows insertions, deletions, and membership tests (among other operations) in expected logarithmic time and is closely related to the self-adjusting (splay) trees presented in the last lecture.

Consider a binary tree, not necessarily balanced, with nodes drawn from a totally ordered set, ordered in inorder; that is, if i is in the left subtree of k and j is in the right subtree of k, then $i < k < j$. Recall that the rotate operation discussed in the previous lecture preserves this order.



Now suppose that each element k has a unique priority $p(k)$ drawn from some other totally ordered set, and that the elements are ordered in heap order according to priority; that is, an element of maximum priority in any subtree is found at the root of that subtree. A tree in which the data values k are ordered in inorder and the priorities $p(k)$ are ordered in heap order is called a treap (for tree-heap). It may not be obvious that treaps always exist for every priority assignment. Moreover, if the priorities are distinct, then the treap is unique.

Lemma 1. *Let X and Y be totally ordered sets, and let p be a function assigning a distinct priority in Y to each element of X. Then there exists a unique treap with nodes X and priorities p.*

Proof: Let k be the unique element of X of maximum priority; this must be the root. Partition the remaining elements into two sets

$$\{i \in X : i < k\} \text{ and } \{i \in X : i > k\}$$

Inductively build the unique treaps out of these two sets and make them the left and right subtrees of k, respectively. □

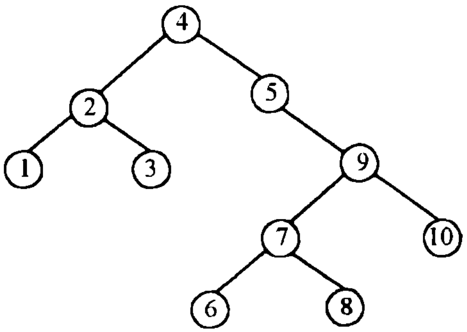
A random treap is a treap in which the priorities have been assigned randomly. This is best done in practice by calling a random number generator each time a new element m is presented for insertion into the treap to assign a random priority to m. Under some highly idealized but reasonable assumptions about the random number generator (which gives a uniformly distributed random real number in the interval $[0,1)$, and successive calls are statistically independent), two elements receive the same priority with probability zero, and if all elements in the treap are sorted by priority, then every permutation is equally likely. When a new element m is presented for insertion or to test membership, we start at the root and work our way down some path in the treap as in any binary search tree, comparing m to elements along the path to see which way to go to find m' s appropriate inorder position. If we see m on the path on the way down, we can answer the membership query affirmatively. If we make it all the way down without seeing m, we can answer the membership query negatively. If m is to be inserted, we attach m as a new leaf in its appropriate inorder position. At that point we call the random number generator to assign a random priority $p(m)$, which by Lemma 1 specifies a unique position in the treap. We then rotate m upward as long as its priority is greater than that of its parent, or until m becomes the root. At that point the tree is in heap order with respect to the priorities and in inorder with respect to the data values.

To delete m, we first find m by searching down from the root as for any binary search tree, then rotate m down until it is a leaf, taking care to choose the direction of rotation so as to maintain heap order. For example, if the children of m are j and k and $p(j) > p(k)$, then we rotate m down in the direction of j, since the rotate operation will make j an ancestor of k. When m becomes a leaf, we prune it off. The beauty of this approach is that the position of any element in the treap is determined once and for all at the time it is inserted, and it stays put at that level until it is deleted; there is not a lot of restructuring going on as with splay trees. Moreover, as we will show below, the expected number of rotations for an insertion or deletion is at most two. We now show that, averaged over all random priority assignments, the expected time for any insertion, deletion or membership test is $O(\log n)$. We will do the analysis for deletes only; it is not difficult to see that the time bound for membership tests and insertions is proportionally no worse than for deletions. Suppose that at the moment, the treap contains n data items (without loos of generality, say $\{1, 2,\dots,n\}$), and we wish to delete m. The priorities have been chosen randomly, so that if the set $\{1,2,\dots,n\}$ is sorted in decreasing order by priority to obtain a permutation s of $\{1,2,\dots,n\}$, every s is equally likely. In order to locate m in the treap, we follow the path from the root down to m. The amount of time to do this is proportional to the length of the path. Let us calculate the expected length of this path, averaged over all possible random permutations σ.

Let $L(m) = \{1, 2,...,m\}$ and $G(m) = \{m, m + 1,...,n\}$. Let A be the set of ancestors of m, including m itself. The definitions of $L(m)$ and $G(m)$ do not depend on σ, but the definition of A does. Let X be the random variable defined as follows: X = length of the path from the root down to $m = |L(m) \cap A| + |G(m) \cap A| - 2$($m$ is counted in both $L(m)$ and $G(m)$). We are interested in $E(X)$, the expected value of X; by linearity of expectation, we have $E(X) = E(|L(m) \cap A|) + E(|G(m) \cap A|) - 2$. By symmetry, it will suffice to calculate $E(|L(m) \cap A|)$. Notice that if the elements of $L(m)$ are sorted in descending order by priority, then: every permutation of $L(m)$ is equally likely; and an element of $L(m)$ is in A if and only if it is greater than all previous elements of $L(m)$ in sorted order. In other words, permute $L(m)$ randomly, then scan the resulting list from left to right, checking off those elements k that are larger than anything to the left of k; the quantity $E(|L(m) \cap A|)$ is the expected number of checks.

Example. Let $n = 10$ and $m = 8$. Suppose that when priorities are assigned randomly to $\{1,2,...,10\}$ and these elements are sorted in decreasing order by priority, we get the permutation: $\sigma = (4,5,9,2,1,7,3,10,8,6)$.

This results in the following treap:



Then $L(m) = \{1,2,3,4,5,6,7,8\}$. If we restrict the random permutation σ to this set, we obtain the permutation $(4,5,2,1,7,3,8,6)$. Scanning in this permutation from left to right and checking only those elements k that are greater than all elements to the left of k, we get the sequence

(4,5,7,8). This is exactly the sequence of elements in $L(m)$ appearing on the path from the root down to m in the treap. A similar argument using $G(m)$ gives the sequence (9,8) which is the sequence of elements in $G(m)$ appearing on the path from the root down to m in the treap. The length of the path is the sum of the two lengths of these sequences less 2, i.e. $4 + 2 - 2 = 4$. We are thus left with the problem of determining the expected value of the random variable Y_m, the number of checks obtained when scanning a random permutation of $\{1,2,\ldots,m\}$ from left to right and checking every element that is greater than anything to its left.

Suppose we permute $\{1,\ldots,m\}$ randomly to get the random permutation σ. Deleting 1 from σ, we get a random permutation σ' of $\{2,3,\ldots,m\}$. Notice that an element other than 1 is checked when scanning σ if and only if it is checked when scanning σ'; thus the presence or absence of 1 does not affect whether other element is checked since 1 is the smallest element.

Thus the expected number of checks on elements other than 1 is the same in σ as in σ', or $E(Y_{m-1})$. The element 1 is checked if and only if it occurs first in σ, and this occurs with probability $\frac{1}{m}$. Thus the expected number of checks on the element 1, averaged over all permutations, is $\frac{1}{m}$. By linearity of expectation,

$$E(Y_m) = E(Y_{m-1}) + \frac{1}{m}.$$

Since $E(Y_1) = 1$ the solution of this recurrence is $E(Y_m) = \sum_{k=1}^{m} \frac{1}{k} = H_m$, the m – th harmonic number and $H_m \sim \ln m$; in particular we have $H_m = O(\ln m)$.

A similar analysis allows us to calculate the expected number of rotations necessary to delete m from its position in the treap; the result is $\frac{m-1}{m} + \frac{n-m}{n-m+1} < 2$.

§ 19. Multidimensional data structures

By reconsidering searching problems in higher dimensional space, a number of problems become interesting only in higher dimensions. Let U be some ordered set and let $S \subseteq U^d$ for some d. An element $x \in S$ is a d – tuple $(x_0, x_1, \ldots, x_{d-1})$. The simplest searching problem is to specify a point $y \in U^d$ and to ask whether $y \in S$; this is called an exact match query and can in principle be solved by the methods that use a balanced search tree since U^d can be totally ordered by lexicographic order. A very general form of query is to specify a region $R \subseteq U^d$ and to ask for all points in $R \cap S$. General region queries can only be solved by exhaustive search of set S. Special and more tractable cases are obtained by restricting the query region R to some subclass of regions. Restricting R to polygons gives us polygon searching, restricting it further to rectangles with sides parallel to the axes gives us range searching, and finally restricting the class of rectangles even further gives us partial match retrieval. In one-dimensional space balanced trees solve all these problems efficiently. In higher dimensions we will need different data structures for different types of queries: d-dimensional trees, range trees and polygon trees. It seems to be very difficult to deal with insertions and deletions to balance these structures after insertions and deletions in many dimensions.

Multidimensional searching problems appear in numerous applications, most notably database systems. In these applications U is an arbitrary ordered set, e.g. a set of names or a set of possible incomes. Region queries arise in these applications in a natural way. E.g. in a database containing information about persons, say name, income, and number of children, we might ask for all persons with:number of children=3, a partial match query; number of children = 3, 2000 ≤ income ≤ 3000, a range query; income = 2000+1000 • (number of children), a polygon query.

Definition. Let T_1, T_2, T_3 be sets. A searching problem Q of type T_1, T_2, T_3 is a function Q: $T_1 \times 2^{T_2} \rightarrow T_3$.

A searching problem takes a point in T_1 and a subset of T_2 and produces an answer in T_3. For example, in the member problem we have $T_1 = T_2$, $T_3 = \{\text{true, false}\}$ and $Q(x, S)$ = „$x \in S$" In the nearest neighbour problem in the plane we have $T_1 = T_2 = \mathbf{R}^2$, $T_3 = \mathbf{R}$ and $Q(x, S) = \delta(x, y)$, where $y \in S$ and $\delta(x, y) \leq \delta(x, z)$ for all $z \in S$. Here δ is some metric. In the inside the convex hull problem we have $T_1 = T_2 = \mathbf{R}^2$, $T_3 = \{\text{true,false}\}$ and $Q(x, S)$ = „is x inside the convex hull of point set S ".

A statistic data structure S for a searching problem supports only query operation Q, i.e. for every $S \subseteq T_2$ one can build a static data structure S such that function $Q(x, S)$: $T_1 \rightarrow T_3$ can be computed efficiently. A semi-dynamic data structure D for a searching problem supports in addition operation Insert, i.e. we can not only query D but also insert new points into D. A dynamic structure supports Insert and Delete. There exists a general method for turning static data structures into semi-dynamic data structures; this method is only applicable to a subclass of searching problems, the decomposable searching problems.

Let U_i, $0 \leq i < d$, be an ordered set and let $U = U_0 \times U_1 \times \ldots \times U_{d-1}$. An element $x = (x_0, \ldots, x_{d-1}) \in U$ is also called point or record or d – tuple; it is customary to talk about points in geometric applications and about records in database applications. Components x_i are also called coordinates or attributes. A region searching problem is specified by a set $\Gamma \subseteq 2^U$ of regions in U. The problem is then to organize a static set $S \subseteq U$ such that queries of the form " list all elements in $S \cap R$ " or " count the number of points in $S \cap R$ " can be answered efficiently for arbitrary $R \in \Gamma$. Since region searching problems are decomposable searching problems, we have dynamic solutions for region searching problems once a static solution is found.

We address four types of region queries.

a) Orthogonal Range Queries : Here Γ is a set of hypercubes in U, i.e. $\Gamma_{OR} = \{R ; R = [l_0, h_0] \times [l_1, h_1] \times \ldots \times [l_{d-1}, h_{d-1}]$ where $l_i, h_i \in U_i$ and $l_i \leq h_i\}$.

b) Partial Match Queries: Here Γ is the set of degenerated hypercubes where every side is either a single point or all of U_i, i.e.

$\Gamma_{PM} = \{R; R = [l_0, h_0] \times [l_1, h_1] \times \ldots \times [l_{d-1}, h_{d-1}]$ where $l_i, h_i \in U_i$ and either $l_i = h_i$ or $l_i = -\infty$ and $h_i = \infty$ for every i $\}$

If $l_i = h_i$ then the i – th coordinate is specified, otherwise it is unspecified.

c) Exact Match Queries: Here Γ is the set of singletons, i.e. $\Gamma_{EM} = \{R; R = \{x\}$ for some $x \in U\}$.

d) Polygon Queries: Polygon queries are only defined for $U = \mathbf{R}^2$. We have

$\Gamma_p = \{R; R$ is a simple polygonal region in $\mathbf{R}^2\}$.

There seems to be no single data structure doing well on all of them and we therefore mention three data structures: d – dimensional trees, polygon trees, and range trees. d – dimensional trees and polygon trees use linear space and solve partial match queries and polygon queries in time $O(n^\varepsilon)$ where ε depends on the type of the problem. Range trees allow us to solve orthogonal range queries in time $O((\log n)^d)$ but they use non-linear space $O(n(\log n)^{d-1})$. In fact they exhibit a tradeoff between speed and space. In one – dimensional space we could solve a large number of problems in linear space and logarithmic time, in higher dimensions all data structures mentioned above either use non – linear space or use "rootic " time $O(n^\varepsilon)$ for some ε, $0 < \varepsilon < 1$.

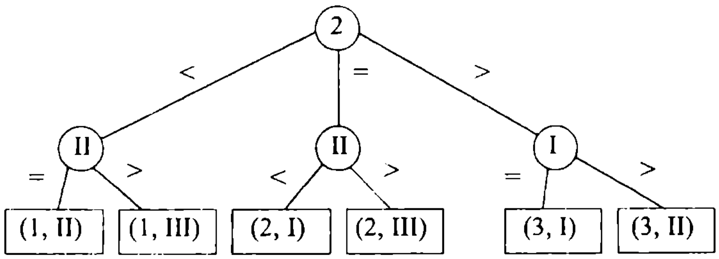
We present d-dimensional trees that are a straightforward, yet powerful extension of one-dimensional search trees. At every level of a dd – tree we split the set according to one of the coordinates. Fairness demands that we use the different coordinates with the same frequency; this is most easily achieved if we go through the coordinates in cyclic order.

Definition. Let $S \subseteq U_0 \times \ldots \times U_{d-1}$, $|S| = n$. A dd–tree for S (starting at coordinate i) is defined as follows:

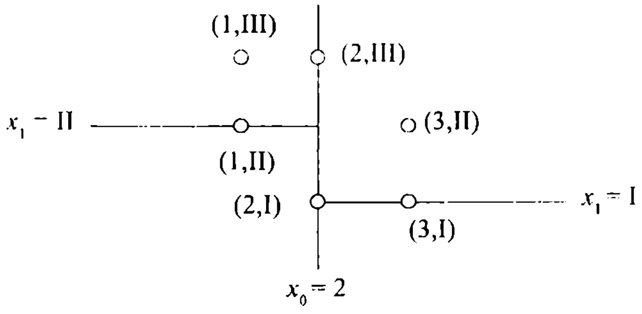
1) If $n = d = 1$ then it consist of *a single leaf* labelled by the unique element $x \in S$.

2) If $d > 1$ or $n > 1$ then it consists of *a root* labelled by some element $d_i \in U_i$ and three subtrees $T_<$, $T_=$, and $T_>$. Here $T_<$ is a dd–tree starting at coordinate $(i+1)(\mathrm{mod}\ d)$ for set $S_< = \{x \in S : x = (x_0, \ldots, x_{d-1})$ and $x_i < d_i\}$, $T_>$ is a dd – tree starting at coordinate $(i+1)(\mathrm{mod}\ d)$ for set $S_> = \{x \in S; x = (x_0, \ldots, x_{d-1})$ and $x_i > d_i\}$ and $T_=$ is a $(d-1)$–dimensional tree starting at coordinate i (mod $d-1$) for set $S_= = \{(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{d-1}); x = (x_0, \ldots, x_{i-1}, d_i, x_{i+1}, \ldots, x_d) \in S\}$.

The figure below shows a $2d$ – tree for set $S = \{(1,\mathrm{II}), (1,\mathrm{III}), (2,\mathrm{I}), (2,\mathrm{III}), (3,\mathrm{I}), (3,\mathrm{II})\}$ starting at coordinate 0. Here $U_0 = U_1 = \{1, 2, 3\}$. Arabic and roman numerals are used to distinguish coordinates.



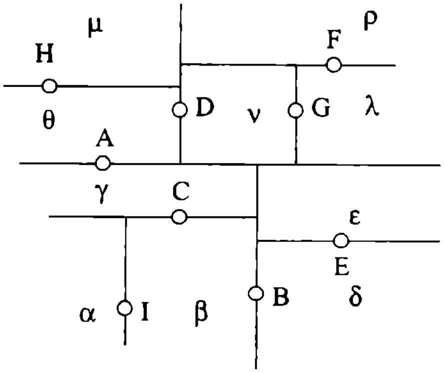
It is very helpful to visualize $2d$ – trees as subdivisions of the plane. The root node splits the plane by vertical line $x_0 = 2$ into three parts:

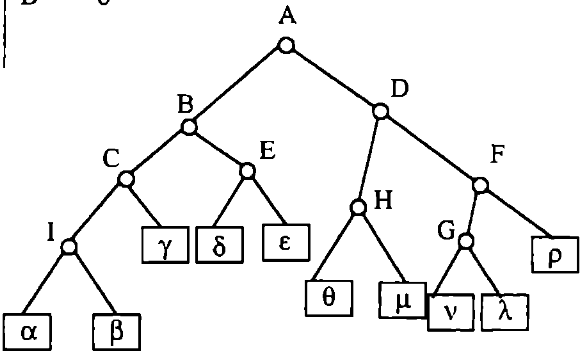


left halfplane, right halfplane and the line itself. The left son of the root then splits the left halfplane by horizontal line $x_1 = 2$, the right son splits the right halfplane by horizontal line $x_1 = I$.

In many cases it is more convenient to represent two-dimensional ($2d$) trees as binary trees that are dynamic, adaptable data structures, dividing up the geometric 2 - dimensional space in a manner convenient for use in range searching and other problems. The idea is to build binary search trees with points in the nodes, using the y and x coordinates of the points as keys in a strictly alternating sequence. The same algorithm can be used to insert points into $2d$ trees as in normal binary search trees, but at the root we use the y coordinate (if the point to be inserted has a smaller y coordinate than the point at the root, go left; otherwise go right), then at the next level we use the x coordinate, then at the next level the y coordinate, etc., alternating until an external node is encountered.

The significance of this technique is that it corresponds to dividing up the plane in a simple way: all the points below the point at the root go in the left subtree, all those above in the right subtree, etc. The figure below show how the plane is subdivided corresponding to the construction of the tree in the next figure.





First a horizontal line is drawn at the y – coordinate of A, the first node inserted. Then, since B is below A, it goes to the left subtree of A; the halfplane below A is divided with a vertical line at the x – coordinate of B. Then, since C is below A, we go left at the root, and since it is to the left of B we go left at B, and divide the portion of the plane below A and to the left of B with a horizontal line at the y – coordinate of C. The insertion of D is similar, but it goes to the right of A, since it is above it, etc.

Every external node of the tree corresponds to some rectangle α, β, γ, ..., λ, ρ in the plane and each region corresponds to an external node in the tree; each point lies on a horizontal or vertical line segment that defines the division made in the tree at that point.

The algorithm to construct $2d$ – trees is a straightforward modification of standard binary tree search to switch between x and y coordinates at each level.

The three sons of a node v in a dd–tree do not all have the same quality. The root of $T_=$ (the son via the = –pointer) represents a set of one smaller dimension. The roots of $T_<$ and $T_>$ (the sons via the < – pointer and the > – pointer) represent sets of the same dimension but generally smaller size. Thus every edge of a dd– tree reduces the complexity of the set represented: either in dimension or in size. In $1d$– trees, i.e. ordinary search trees, only reductions in size are required.

It is clear how to perform exact match queries in dd–trees. Start at the root, compare the search key with the value stored in the node and follow the correct pointer. Running time is proportional to the height of the tree. Our first task is therefore to derive bounds on the height of dd–trees.

Definition : a) Let T be a dd– tree and let v be a node of T. Then $S(v)$ is *the set of leaves in the subtree with root* v, $d(v)$ *is the depth of node* v, and $sd(v)$, *the number of* < – *pointers and* > – *pointers on the path from the root to* v, *is the strong depth of* v. Node x is *a proper son of node* v if it is a son via a < – or > – pointer.

b) A dd–tree is *ideal* if $|S(x)| \leq |S(v)|/2$ for every node v and all proper sons x of v.

Ideal dd–trees are a generalization of perfectly balanced $1d$–trees.

Lemma 1. *Let T be an ideal dd–tree for set S, $|S| = n$. Then*:

a) $d(v) \leq d + \log n$ *for every node v of T*;

b) $sd(v) \leq \log n$ *for every node v of T*.

Proof: a) follows from b) and the fact that at most d = – pointers can be on the path to any node v. Part b) is immediate from the definition of an ideal tree.

Theorem 1. *Let $S \subseteq U = U_0 \times \ldots \times U_{d-1}$, $|S| = n$.*

a) *An exact match query in an ideal dd–tree for S takes time* $O(d + \log n)$.

b) *An ideal dd–tree for S can be constructed in time* $O(n(d + \log n))$.

Proof : a) is immediate from Lemma 1a).

b) We describe a procedure which constructs ideal dd – trees in time $O(n(d + \log n))$. Let $S_0 = \{x_0;\ (x_0, \ldots, x_{d-1}) \in S\}$ be the multiset of

0 – th coordinates of S. We use the linear time median algorithm to find the median d_0 of S_0. d_0 will be the label of the root. Then clearly $|S_<| \le |S|/2$ and also $|S_>| \le |S|/2$ where $S_< = \{x \in S; x_0 < d_0\}$ and $S_> = \{x \in S; x_0 > d_0\}$. We use the same algorithm recursively to construct dd – trees for $S_<$ and $S_>$ (starting at coordinate 1) and a $(d-1)$–dimensional trees for $S_=$. This algorithm will clearly construct an ideal dd – tree T for S. The bound on the running time can be seen as follows. In every node v of T we spend $O(\,|S(v)|\,)$ steps to compute the median of a set of size $|S(v)|$. Furthermore, $S(v) \cap S(w) = \emptyset$ if v and w are nodes of the same depth and hence $\sum_{d(v)=k} |S(v)| \le n$ for every k, $0 \le k < d + \log n$. Thus the running time is bounded by

$$\sum_{v \text{ node of } T} O(|S(v)|) = O\left(\sum_{0 \le k \le d+\log n} \; \sum_{d(v)=k} |S(v)| \right) = O(n(d + \log n)) \,. \;\square$$

Insertions into dd–trees are a non-trivial problem. A first idea is to use an analogue to the naive insertion algorithm into one-dimensional trees. If x is to be inserted into tree T, search for x in T until a leaf is reached and replace that leaf by a small subtree with two leaves. Of course, the tree will not be ideal after the insertion in general. We might define weight – balanced dd – trees to remedy the situation, i.e. we choose some parameter α, say $\alpha = 1/4$, and require that $|S(x)| \le (1-\alpha)|S(v)|$ for every node v and all proper sons x of v. This is a generalization of $BB[\alpha]$ trees. Two problems arise, that illustrate a major difference between one-dimensional and multi-dimensional searching. The first problem is that although theorem 1 is true for weight-balanced dd – trees, query time in near-ideal dd – trees may have a different order than query time in ideal trees. More precisely, partial match in ideal $2d$ – trees has running time $O(\sqrt{n})$ but it can be shown that it has running time $O(n^{0.706})$ in weight-balanced dd – trees for $\alpha = 1/4$. Thus weight balanced dd – trees are only useful for exact match queries. A second problem is that weight-balanced dd – trees are hard to rebalance. Rotations are of no use since splitting is done with respect to different coordinates on different levels. Thus it is impossible to change the depth of a node as rotations do.

ASSIGNMENTS

LIST A

A1. Devise a way to represent circular lists inside a computer in such a way that the list can be traversed efficiently in both directions, yet only one link field is used per node.

Hint: Let the link field of node x_i contain $\mathrm{LOC}(x_{i+1}) - \mathrm{LOC}(x_{i-1})$. Two adjacent list heads are included in the circular list, to help get things started properly.

Design algorithms of insertion and deletion such that the list can be used as either a stack or a queue.

Ref: D. E. Knuth, *The art of computer programming, vol. 1*, (Reading, Massachusetts, 1969). *ex.* 18, *p.* 277.

A2. Find an optimum binary search tree for:

a) $n = 4; q_0 = \frac{2}{17}, q_1 = \frac{2}{17}, q_2 = \frac{3}{17}, q_3 = \frac{1}{17}, q_4 = \frac{1}{17};$

$p_1 = \frac{2}{17}, p_2 = \frac{1}{17}, p_3 = \frac{3}{17}, p_4 = \frac{2}{17}.$

b) $n = 4; q_0 = \frac{3}{17}, q_1 = \frac{1}{17}, q_2 = \frac{1}{17}, q_3 = \frac{2}{17}, q_4 = \frac{2}{17}; p_1 = \frac{3}{17},$

$p_2 = \frac{2}{17}, p_3 = \frac{1}{17}, p_4 = \frac{2}{17}.$

c) $n = 4; q_0 = \frac{1}{17}, q_1 = \frac{2}{17}, q_2 = \frac{3}{17}, q_3 = \frac{1}{17}, q_4 = \frac{2}{17}, p_1 = \frac{1}{17},$

$p_2 = \frac{3}{17}; p_3 = \frac{2}{17}, p_4 = \frac{2}{17}.$

Ref: D. E. Knuth, *The art of computer programming* (Addison - Wesley, 1973), vol. 3, pp. 434 - 435.

A3. Design an algorithm to add two sparse matrices: given matrices A and B, A will retain the sum:

$$a_{ij} \leftarrow a_{ij} + b_{ij}, \ \forall 1 \le i \le m; \ 1 \le j \le n.$$

The two input matrices should be represented as sparse matrices (circularly linked lists for each row and column; there are special list head nodes, for every row and column).

Ref: D. E. Knuth, *The art of computer programming* (Reading, Massachusetts, 1969), vol. 1, p. 300.

A4. Design an efficient algorithm which replace the N quantities $(R_1, \ldots, R_N)$ by $(R_{p(1)}, \ldots, R_{p(N)})$, respectively, given the values of $R_1, \ldots, R_N$ and the permutation $p(1) \ldots p(N)$ of $\{1, \ldots, N\}$. Try to avoid using excess memory space. (This problem arises if we wish to rearrange records in memory after an address table sort, without requiring room for storing $2N$ records).

Propose algorithms in both cases where $p(i)$ is a function of i which is tabulated and/or is to be computed.

What is expression of $p(i)$ in the case of matrix transposition (suppose that the matrix is $M \times N$) ?

Ref: D. E. Knuth, *The art of computer programming* (Addison-Wesley, 1973), vol. 3, ex. 10, p. 80; answer p. 595.

A5. Let P_n denote the number of possible outcomes when n elements are sorted with ties allowed, so that $(P_0, P_1, P_2, P_3, P_4, P_5, \ldots) =$ $= (1, 1, 3, 13, 75, 541, \ldots)$.

For example, when equality between keys is allowed, there are 13 possible outcomes when sorting three elements:

$K_1 = K_2 = K_3, K_1 = K_2 < K_3, K_1 = K_3 < K_2$

$K_2 = K_3 < K_1, K_1 < K_2 = K_3, K_2 < K_1 = K_3$

$K_3 < K_1 = K_2, K_1 < K_2 < K_3, K_1 < K_3 < K_2$

$K_2 < K_1 < K_3, K_2 < K_3 < K_1, K_3 < K_1 < K_2$

and $K_3 < K_2 < K_1$.

Prove that the generating function

$$P(z)=\sum_{n=0}^{\infty}\frac{P_n z^n}{n!} \quad \text{is equal to} \quad \frac{1}{2-e^z}.$$

Hint: Show that $P_n=\sum_{k>0}\binom{n}{k}P_{n-k}$ when $n>0$.

Ref: D. E. Knuth, *The art of computer programming* (Addison - Wesley, 1973), *vol*. 3, *ex*. 3, *p*. 195; answer p. 627.

A6. Given a file containing N 30 - bit binary words $x_1, x_2, \ldots, x_N$, how would you find the number of all pairs (x_i, x_j) such that $x_i = x_j$ except in at most two bit positions? Use an algorithm of complexity $O(N \log N)$.

Hint: Create a file with 31 N entries, forming 31 entries from each original word x_i by including x_i and the 30 words that differ from x_i in one position. Sort this extended file and look for duplicates.

Find a formula counting the looking for number of pairs as a function of the number of duplicates in the sorted file and propose an algorithm to do this (after the sorting of the file).

Ref: D. Knuth, *The art of computer programming* (Addison - Wesley, 1973), *vol*. 3, *ex*. 20, *p*. 9; answer p. 576.

A7. Given any permutation $p = a_1 a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$, let xch(p) be the minimum number of exchanges which will sort p into increasing order. Express xch(p) in terms of „simpler" characteristics of p, namely prove that

$$\text{xch}(p) = n - c(p),$$

where $c(p)$ is the number of cycles of p.

Propose an algorithm that sort p by using this minimum number of element transpositions.

Ref: D. Knuth, *The art of computer programming* (Addison - Wesley, 1973), *vol*.3, *ex*. 2, *p*. 134; answer p. 605.

A8. Suppose that, instead of sorting an entire file, we only want to determine the m-th smallest of a given set of n elements. Show that „quicksort" can be adapted to this purpose, avoiding many of the computations required to do a complete sort.

Ref: D. Knuth, *The art of computer programming* (Addison - Wesley, 1973), *vol.* 3, *ex.* 31, *p.* 136; answer p. 610.

See also C.A.R. Hoare, Comm. ACM 4 (1961), pp. 321 - 322.

A9 . a) Prove that every positive integer n has a unique representation as a sum of Fibonacci numbers $n = F_{a_1} + F_{a_2} + \ldots + F_{a_r}$, where $r \geq 1$, $a_j \geq a_{j+1} + 2$ for $1 \leq j \leq r - 1$, and $a_r \geq 2$.

b) Prove that in the Fibonacci tree of order k, the path from the root to node $\textcircled{n}$ has length $k + 1 - r - a_r$.

Ref: D. E. Knuth, *The art of computer programming, vol. 1: ex* 34, *p.* 85; answer p. 493.

D. E. Knuth, *The art of computer programming, vol. 3, ex.* 17, *p.* 421; answer p. 669.

A10. Binomial heaps.

Ref. D. C. Kozen, *The design and analysis of algorithms,* Springer - Verlag, New York, 1992.

ASSIGNMENTS

LIST B

B1. Splay trees.

Ref.: D. C. Kozen, *The design and analysis of algorithms,* Springer-Verlag, New York, 1992, *pp.* 58- 64.

B2. Random search trees.

Ref.: D. C. Kozen, *The design and analysis of algorithms,* Springer - Verlag, New York, 1992, *pp.* 65 - 70.

B3. The Huffman optimum tree in non-binary case + exercise 7.1, p. 145 of the reference.

Ref.: S. Even, *Algorithmic combinatorics,* Macmillan, New York and London, 1973, *pp.* 127 - 140.

B4. Show that we can find both the maximum and the minimum of a set of n elements, using at most $\left\lceil \frac{3n}{2} \right\rceil - 2$ comparisons, and the latter number cannot be lowered.

($\lceil x \rceil$ denotes the smallest integer greater than or equal to x).

Ref.: D. E. Knuth, *The art of computer programming* (Addison - Wesley, 1973), *vol.* 3., *ex.* 16, *p.* 220; answer *p.* 167; see also I. Pohl, *Comm. ACM* 15(1972), 462 - 464.

B5. Red - black trees + exercises 14.1 - 1 to 14.1 - 3 (properties, rotations and insertion only).

Ref.: T. Cormen, C. Leiserson, R. Rivest, *Introduction to algorithms*, MIT Press/ Mc Graw Hill, 1990, *pp.* 263 - 272.

B6. Determine the average internal path length of a binary tree with n nodes, assuming that each of the

$$\frac{1}{n+1}\binom{2n}{n}$$

trees is equally probable. Find the asymptotic value of this quantity.

Ref.: D. E. Knuth, *The art of computer programming, vol.* 1 (Reading, Massachusetts, 1969), *ex.* 5, *p.* 404; answer p. 590.

For the generating function of the number of binary trees with n nodes see pp. 388 - 389.

B7. Algorithm PATRICIA (*Practical Algorithm To Retrieve Information Coded In Alphanumeric*) + *ex.* 7, *p.* 258 of the first reference.

Ref.: 1. R. Sedgewick, *Algorithms*, 2nd edition, Addison - Wesley, 1988, pp. 253 - 257.

2. D. E. Knuth, *The art of computer programming (*Addison - Wesley, 1973), *vol.* 3, pp. 490 - 493.

B8. Algorithm F (*Fibonaccian search*) + *ex.* 14, *p.* 420; answer p. 669.

Ref.: D. E. Knuth, *The art of computer programming* (Addison - Wesley, 1973), *vol.* 3, pp. 414 - 415.

B9. Find a simple formula for a_n, the number of permutations on n elements that can be obtained with a stack like that in ex. 2 of the reference. Show that this problem is equivalent to many other combinatorial problems, such as the enumeration of binary trees, the number of ways to insert parentheses into a product of factors, and the number of ways to divide a polygon into triangles by non-intersecting diagonals (Euler).

Ref.: D. E. Knuth, *The art of computer programming, vol. 1 (*Reading, Massachusetts, 1969), *ex.* 4, *p.* 239, answer p. 531.
See also *pp.* 388 - 389 for the number of binary trees with n nodes.

B10. Fibonacci heaps and their applications to the implementation of Dijkstra's sigle-source shortest-path algorithm and of Prim's algorithm for minimum spanning trees.

Ref.: D. C. Kozen, *The design and analysis of algorithms,* Springer - Verlag, New York, 1992, *pp.* 25 - 26, 44 - 47, 222, 250.

Test 1

Answer four questions: 1, 2, (3A or 3B), (4A or 4B).

1. Design an algorithm which takes a circular list such as in Fig. 1 and reverses the direction of all the arrows.

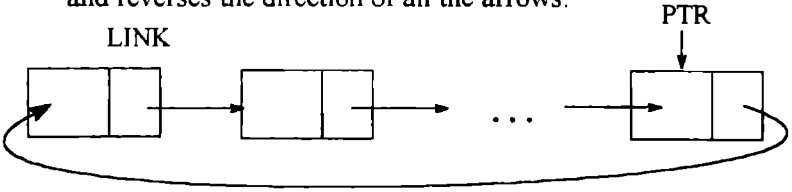


Figure 1

2. A tree is said to be k-ary if every internal vertex has exactly k sons. Find the number of leaves of a k-ary tree with n internal vertices. Justify your answer.

3A. Design an algorithm for a binary tree traversal in symmetric order using an auxiliary stack A. T is a pointer to the root of the tree and all vertices of the binary tree have fields: LLINK, RLINK and CONTENT.

3B. Design an algorithm for tree search and insertion for a given key K. *ROOT* points to the root of the tree ($ROOT \neq \Lambda$) and each node *NODE*(P) contains at least the following fields: *KEY* (P), *LLINK*(P) and *RLINK*(P).

4A. Apply Huffman's algorithm to get an optimum prefix binary code for weights: 1, 1, 2, 2, 2, 2, 3, 4, 5, 6.

4B. Find all optimum binary search trees for $n = 4$, $q_0 = 3$, $p_1 = 0$, $q_1 = 2$, $p_2 = 3$, $q_2 = 1$, $p_3 = 4$, $q_3 = 1$, $p_4 = 2$, $q_4 = 1$.

Note: Each problem is worth 2.5 points.

Test 2

Answer four questions: 1, 2, (3*A* or 3*B*), (4*A* or 4*B*).

1. Find the cyclomatic number (the number of independent cycles) for the graph in Fig. 1.

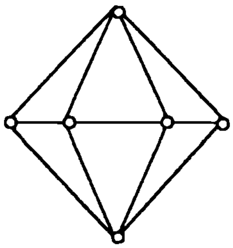


Fig. 1

2. Illustrate the binary decision tree associated to the binary search algorithm of a given key K in a table of records $R_1, \dots, R_{17}$ whose keys are in increasing order $K_1 < K_2 < \dots < K_{17}$.

3A. Design an algorithm to add two polynomials P, Q such that $Q \leftarrow P + Q$ and P, Q are represented as circular simply linked lists having a list head, in the decreasing lexicographic order of their monomials.

3B. If $S(n) = \min_T h(T)$, where T ranges over all decision trees for sorting n elements (or the minimum worst case complexity of any sorting algorithm), prove that

$$\lim_{n\to\infty} \frac{S(n)}{n \log_2 n} = 1$$

4A. Consider a linear simply linked list having a list head HEAD like that in Fig. 2.

Design an insertion and a deletion algorithm such that list becomes a queue. Try to avoid an extensive search into the list.

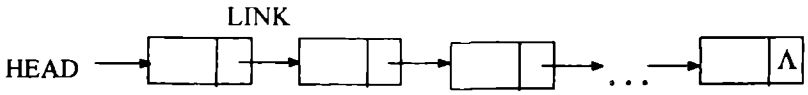


Fig. 2

Caution: The list may consist only of list head.

4B. Consider two linear simply linked lists having list heads HEAD1 and HEAD2, respectively. Design an algorithm to concatenate these two lists into a single list having list head HEAD1.

Caution: The same as for the problem 4*A*.

NOTE: Each problem is worth 2.5 points.

BIBLIOGRAPHY

1. R. K. AHUJA, T. L. MAGNANTI, J. B. ORLIN, *Network flows: Theory, algorithms and applications*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.
2. T. CORMEN, C. LEISERSON, R. RIVEST, *Introduction to algorithms,* MIT Press / Mc Graw Hill, 1990.
3. S. EVEN, *Algorithmic combinatorics,* Macmillan, New York and London, 1973.
4. D. E. KNUTH, *The art of computer programming: vol. 1,* Reading, Massachusetts, 1969; vol. 3, Addison-Wesley, 1973.
5. D. C. KOZEN, *The design and analysis of algorithms,* Springer - Verlag, New York, 1992.
6. K. MEHLHORN, *Data structures and algorithms:* vol. 1: *Sorting and searching;* vol. 3: *Multi -dimensional searching and computational geometry,* Springer - Verlag, Berlin - Heidelberg - New York - Tokyo, 1984.
7. R. SEDGEWICK, *Algorithms,* 2nd edition, Addison - Wesley, 1988.
8. R. E. TARJAN, *Data structures and network algorithms,* SIAM, Philadelphia, Pennsylvania, 1983.

Tiparul s-a executat sub cda 235/1996 la
Tipografia Editurii Universității din București

This book discusses some efficient techniques for data organization. It focuses on four main topics: structured data types (queues and stacks, arrays, traversing binary trees, Huffman trees, multilinked structures, dynamic storage allocation), sorting techniques (sorting by counting, insertion, selection, partitioning, merging, distribution, and lower bounds), searching techniques (self-organizing linear lists, serching ordered sets: binary search and search trees, balanced binary trees, weighted trees, weight-balanced trees; hashing with chaining and opening addressing, perfect hashing, d-heaps, Fibonacci heaps, splay trees, random search trees) and an introduction to multidimensional data structures.

The clear yet rigorous treatment of the topics is illustrated by many examples and algorithms and the text includes many figures.

The material is based on a one-semester graduate course in data structures taught at the University of Bucharest, Romania and at the University of Auckland, New Zealand. ***Data Structures*** is suitable as a text for a gaduate level course, it is aimed primarily at computer science students.

ISBN: 973 – 575 – 147 – X

Lei 18000