UNIVERSITÉ PARIS XI

U.E.R. MATHÉMATIQUE

91405 ORSAY   FRANCE

N° **218** - 76-67

A DATA STRUCTURE FOR MANIPULATING

PRIORITY QUEUES

Jean VUILLEMIN

Département d'Informatique
Université de Paris-Sud

91405 - ORSAY (France)

A DATA STRUCTURE FOR MANIPULATING PRIORITY QUEUES

Jean VUILLEMIN

Département d'Informatique
Université de Paris-Sud

91405 - ORSAY (France)

Abstract :

We describe a data structure which can be used for representing a collection of priority queues. The primitive operations are insertion, deletion, union, update and search for an item of earliest priority.

1. INTRODUCTION.

In order to design correct and efficient algorithms for solving a specific problem, it is often helpful to describe our first approaches to a solution in a language close to that in which the problem was formulated. One such language is that of set theory, augmented by primitive set manipulation operations. Once the algorithm is outlined in terms of these set operations, one can then look for data structures best fitted for representing each of the sets involved. This choice only depends upon the collection of primitive operations required for each set. It is thus important to establish a good catalogue of such data structures, and a summary of the state of the art on this question can be found in AHU [2]. In this paper, we add to this catalogue a data structure which allows efficient manipulation of priority queues.

A priority queue is a set ; each element of such a set has a name and a label. Names are used to uniquely identify set elements. Labels, or priorities are drawn from a totally ordered set. Elements of the priority queue can be thought of as awaiting service, the item with the smallest label always being served next. Ordinary stacks and queues are special cases of priority queues.

A variety of applications directly require using priority queues : job scheduling and page replacement in operating systems, discrete simulation languages where labels represent the time at which events are to occur, as well as various sorting problems. These are discussed for example in AHU [2], C [4], G [11], JD [15], K [17], MS [19], VD [24] and W [27]. Priority queues also play a central role in several good algorithms such as Huffman's and Hu's optimal code constructions (see K [17] and C [4]), Chartres's prime number generator and Brown's power series multiplication (described in K [17]) ;

applications have also been found in numerical analysis algorithms (see G [10] and

MS [19] for example) and in graph algorithms, for finding shortest paths (see

D [5] , AHU [2] and J [13]) and minimum cost spanning tree (see AHU [2] ,

CTY [3] , P [22] and V [25]) among others.

Typical applications require primitive operations among the following five :

INSERT, DELETE, MIN, UPDATE and UNION. The operation INSERT (name, label,

Q) adds an element to queue Q while DELETE (name) removes the element having

that name. Operation MIN (Q) returns the name of the element in Q having the

least label, and UPDATE (name, label) changes the label of the element named.

Finally, UNION (Q_1, Q_2) merges into Q_3 all elements of Q_1 and Q_2 ; the sets Q_1 and Q_2

become empty. In what follows, we assume that names are handled in a separate

dictionnary (see AHU [2] and K [17]) such as a hash-table or a balanced tree.

If deletions are restricted to elements extracted by MIN, such an auxiliary symbol

table can be dispensed with.

A truly elegant data structure, called a heap (see F [7] , GR [12] , K [17] ,

PS [21]) has been discovered by J.W. Williams and R.W. Floyd ; it handles a

sequence of n primitives INSERT, DELETE and MIN in 0 (n log n) elementary

operations[*] and absolutely minimal storage. For applications in which UNION is

necessary, more sophisticated data structures have been devised, such as 3-2 trees

(see AHU [2] and K [17]), AVL trees (AL [1] and K [17]), leftist trees

(C [4]), p-trees (JD [15]) and binary heap (F [9]).

The data structure we present here handles an arbitrary sequence of n pri-

mitives, each drawn among the five described above, in 0(n log n) machine opera-

tions and 0(n) memory cells. It also allows for an efficient treatment of a large

number of updates, which is crucial in connection with spanning tree algorithms :

[*] We assume here that indexing through the symbol table is done in constant time.

our data structure provides an implementation (described in V [25]) of the

Cheriton - Tarjan - Yao (CTY [3]) minimum cost spanning tree algorithm which

is much more straightforward than the original one.

The proposed data structure uses less storage than leftist, AVL or 3-2 trees;

in addition, when the primitive operations are carefully machine coded from the

programs given in paragraph 4, they yield worst case running times which compare

favorably to those of their competitors.

Besides these technical advantages, we feel the data

structure to be interesting in itself, because of its conceptual simplicity and of

the connections it establishes between various data manipulation problems.

2. BINOMIAL TREES AND FORESTS.

We describe here the underlying combinatorial structure, called binomial

trees. These are defined inductively by :

$$B_o = \bigcirc \quad \text{and} \quad B_{p+1} = \quad \text{for } p \geq 0 .$$

Thus $B_2 =$, and $B_3 =$ for
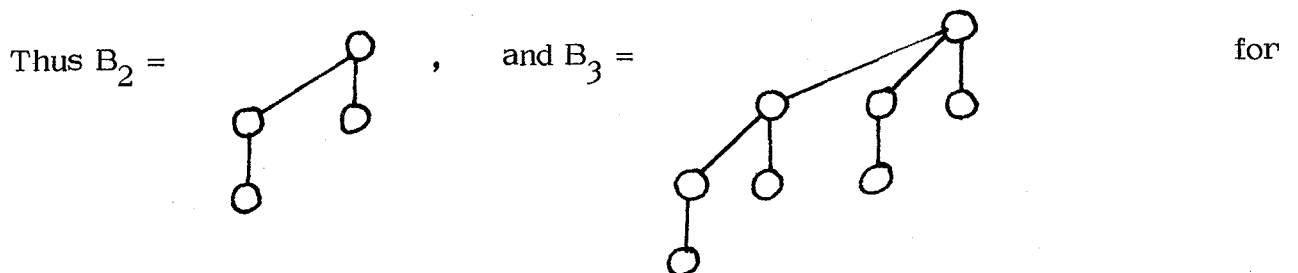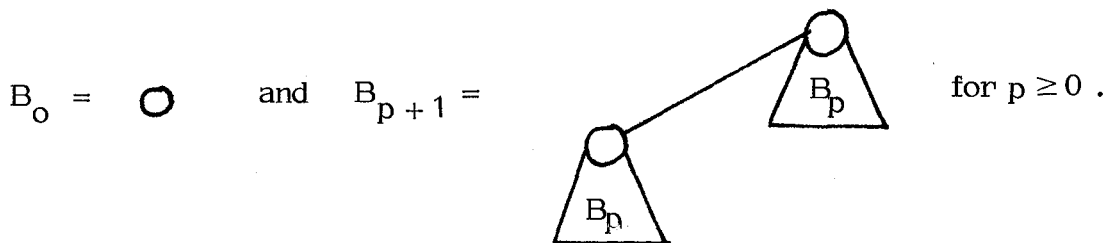
example. In order to discover some of the many combinatorial properties of the

B_i 's, Knuth (K [18]) suggests that we first label the nodes of the tree in

post-order, starting at zero, then associate with each node the binary representation of its label.
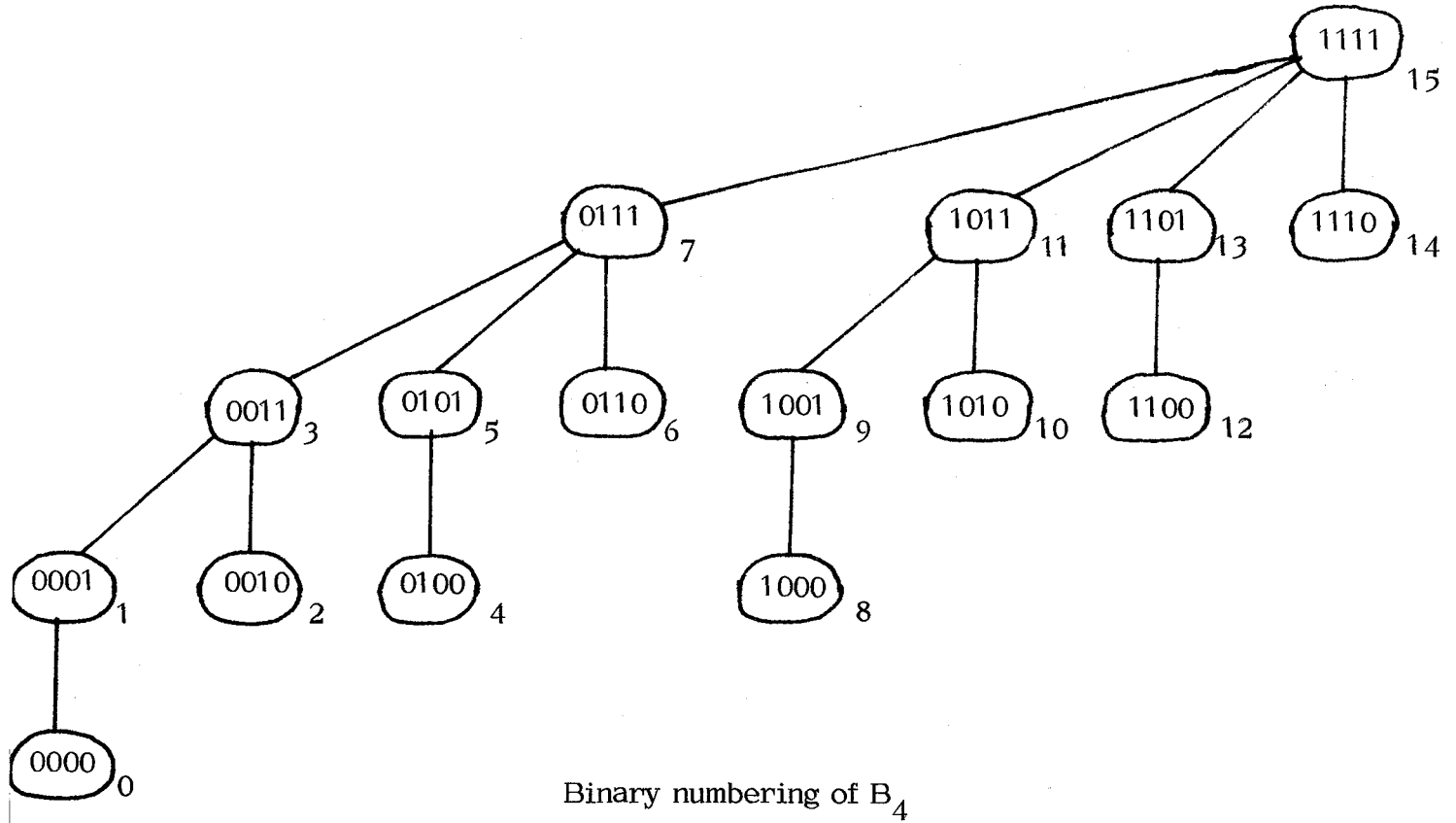

Binary numbering of B_4

Figure 1.

From this numbering, it is easy to establish that

- each B_p has 2^p nodes ;

- there are $\binom{p}{k}$ nodes at depth k in B_p which correspond to the various sequences of p bits having exactly k zeros ;

- the maximum depth of a node in B_p is p ;

- the number of children of a node is equal to the number of 1's following the last 0 in its binary numbering ; leaves thus correspond to even numbers ;

- in B_p, there is exactly one node, the root, having p children ; for $0 \leqslant k < p$ there are 2^{p-k-1} nodes having k children.
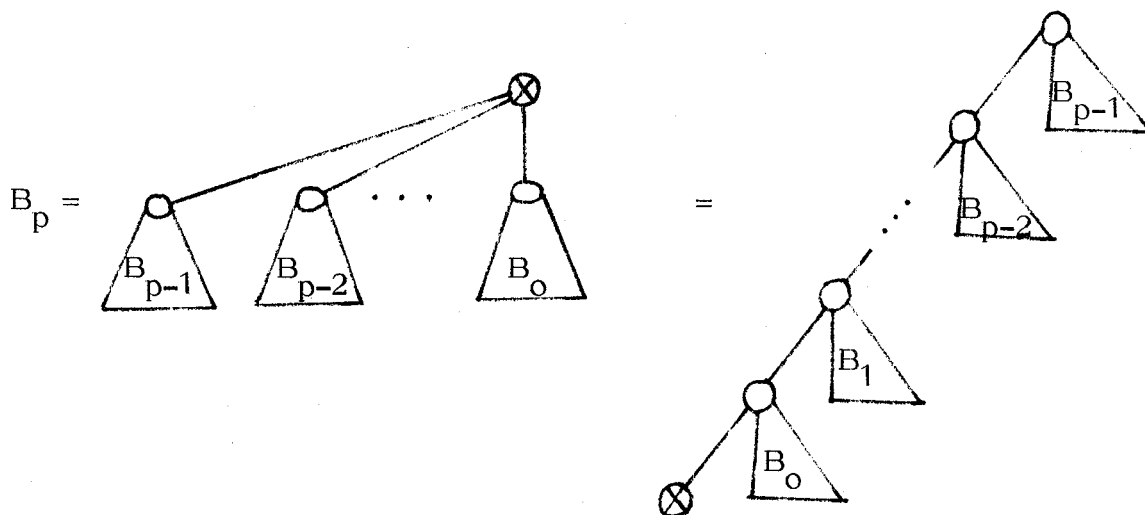
There are many ways of drawing B_p ; in particular :

Figure 2

In order to use binomial trees for representing sets whose number n of elements is not always a power of two, we consider the binary decomposition

$n = \sum\limits_{i \geqslant 0} b_i 2^i$ with $b_i \in \{0, 1\}$ of the number n and define a binomial forest F_n of order n as a finite collection of binomial trees, one B_i for each 1 in the binary decomposition of n. In symbols, $F_n = \{ B_i \ | i \geqslant 0, \ b_i = 1, \ n = \sum\limits_{i \geqslant 0} b_i 2^i \}$; we define the i-th component of F_n to be B_i if $b_i = 1$ and empty otherwise. For example $(12)_{10} = (1100)_2$ thus $F_{12} = \{B_3, B_2\}$; the first component of F_{12} is ϕ and its third B_3.

Figure 2 bis shows some small binomial forests.

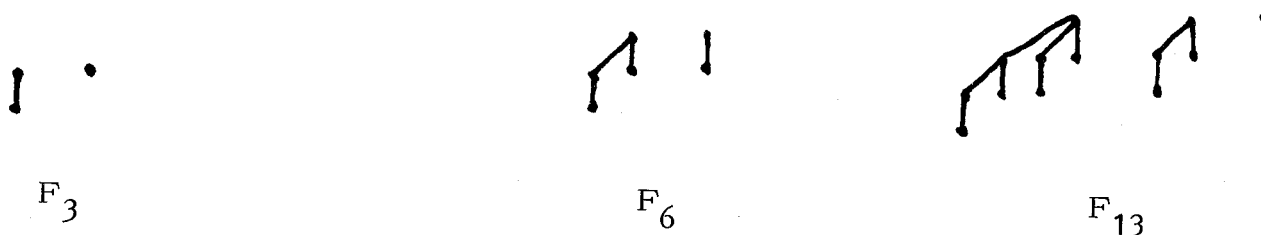F_3 F_6 F_{13}

Figure 2 bis.

Binomial trees and forests appear in various data manipulation problems.

They are used by Fisher (F [6]) in the worst case analysis of a simple data

structure for manipulating disjoint set unions (see also K [18]). They play a

crucial role in the linear time median algorithm of Paterson-Pippinger-Schönhage

(PPS [20]). The Ford-Johnson (FJ [8]) sorting algorithm can also be nicely

described (non-recursively) with the help of binomial trees : the algorithm first

builds a binomial forest, then sorts the partial order thus obtained through a

sequence of repeated "foldings" ; the data structure presented here can actually

by used for implementing the sorting algorithm in time O(n log n). An efficient

machine coding of binary search (see V [25]) uses binomial search trees.

3. BINOMIAL QUEUES.

A priority queue $Q = \{<\nu_1,\lambda_1>,\ldots,<\nu_p,\lambda_p>\}$ consisting of

p items is represented by a labeled binomial forest F_p : each item < name, label >

is stored in a different node of F_p subject to the constraint that, if node i is a

child of node j in F_p, the labels λ_i and λ_j of the items respectively associated

must satisfy $\lambda_i \geqslant \lambda_j$. This is called the "heap condition" by Knuth (K [17]),

and it arises through the natural "contraction" of perfect tournaments as shown in

figure 3 a) and b).

(a)

(b)
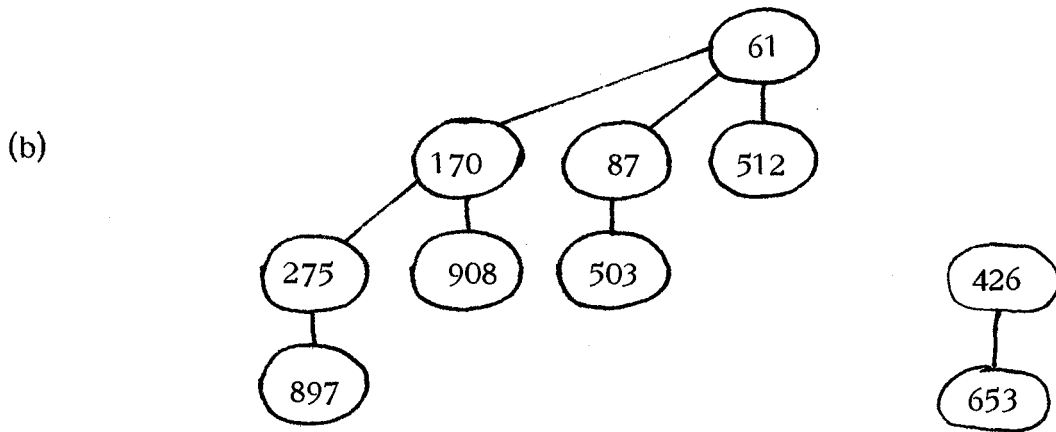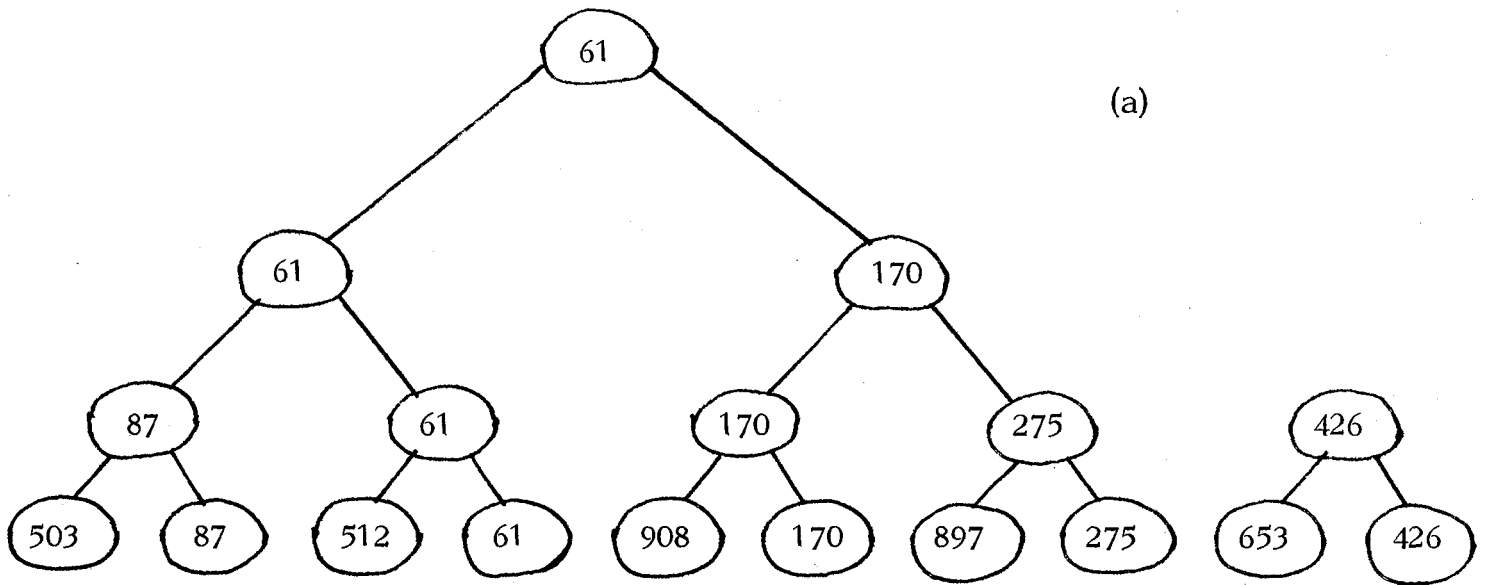
The collection {503, 87, 512, 61, 908, 170, 897, 275, 653, 426} of labels
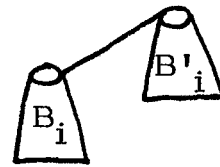
represented as :

(a) a "perfect" tournament

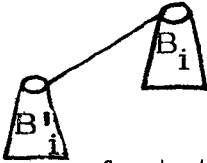(b) the same tournament "contracted" into a binomial queue F_{10}

Figure 3.

Such a labeled binomial forest will be called a binomial queue.

We now describe how to perform the UNION of two binomial queues F_n and $F_{n'}$. First consider the special case $n = n' = 2^i$, that is each priority queue is represented by a single labeled binomial tree, say B_i for F_n and B'_i for $F_{n'}$. The resulting forest $B_{i+1} = \text{UNION}(B_i, B'_i)$ also consists of a single labeled binomial tree with $2^{i+1} = n + n'$ nodes, defined as $B_{i+1} =$ [binomial tree with B'_i as root and B_i as child] if the label of the root of B'_i is smaller than the corresponding label in B_i and

$B_{i+1} =$ [binomial tree with B_i as root and B'_i as child] otherwise, in order to preserve the heap condition. We refer to this operation as coupling, and the general UNION procedure is a sequence of coupling.

For treating the general case, i.e. for n and n' arbitrary, it is convenient to use an analogy with the ordinary scheme for the binary addition of n and n'. The UNION proceeds from low order bits to high order bits, i.e. it treats the binomial trees composing F_n and $F_{n'}$ in order of increasing size. A

carry is propagated at each step ; at the i-th step of the algorithm, the carry is either empty or it is a labeled binomial tree B_i. The initial carry is empty. Step i has three operands which play a symmetric role. Each operand is either empty or it is a labeled binomial B_i. One of the operands is the carry, and the other two are the i-th components of F_n and $F_{n'}$ respectively. If all three operands are empty, the i-th component of the result UNION $(F_n, F_{n'})$ is empty as well as the carry propagated to the next step. If exactly one operand is non-empty, it constitutes the i-th component of the result and the carry is empty. If two operands are non empty , they are coupled according to the procedure described earlier in order to constitute the (i+1)-th carry ; the i-th component of the result is empty. When all three operands are non-empty, one of them arbitrarely constitutes the i-th component of the result and the remaining two are coupled in order to form the carry. The procedure starts at the 0-th step and stops when $i \geqslant 1 + \underline{\min} (\lfloor \log_2 n \rfloor , \lfloor \log_2 n' \rfloor)$ and no carry is propagated any further. The algorithm is pictured below with n = 7 and n' = 5.

		1 0 1			5
+		1 1 1			
carry		1 1 1		+	7
=		1 1 0 0		=	12

(a)

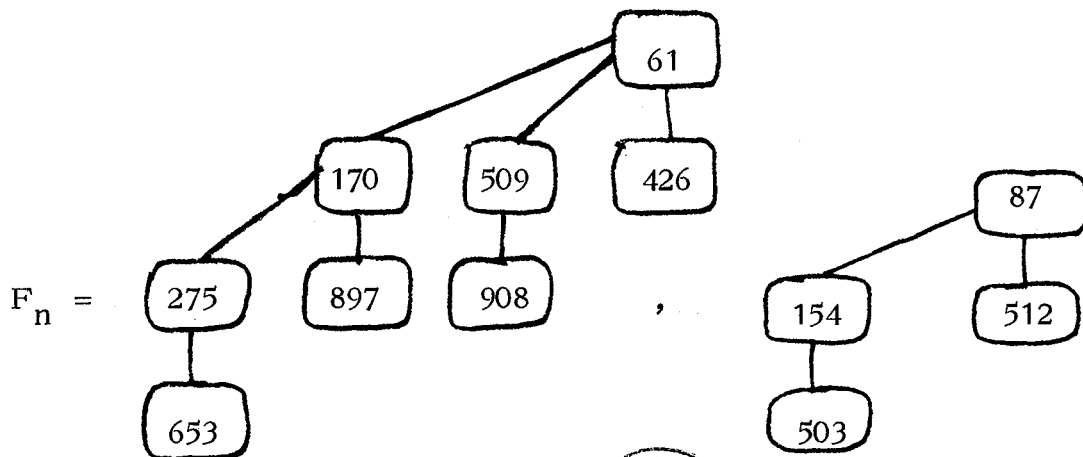|		B_2	$\emptyset$	B_0
+		B_2	B_1	B_0
carry	B_3	B_2	B_1	$\emptyset$
=	B_3	B_2	$\emptyset$	$\emptyset$

(b)
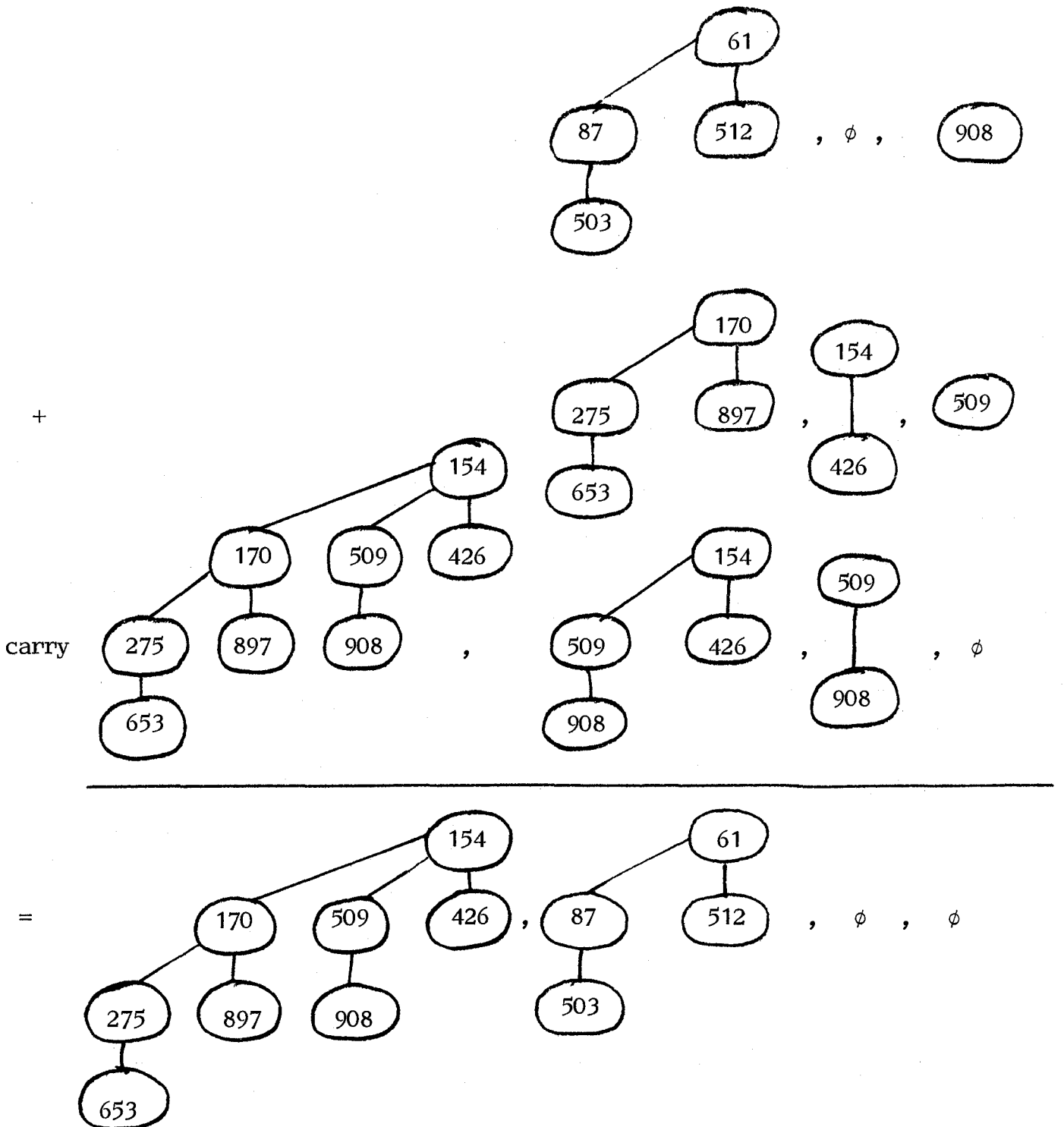
Binary addition (a) of 7 and 5, scheme (b) for UNION (F_7, F_5) and actual example (c)
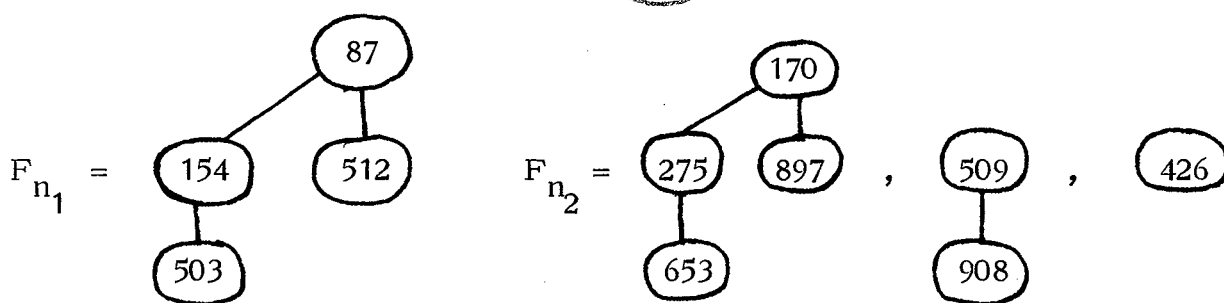
Figure 4

By considering a single item as forming a F_1 binomial queue,

INSERT can be treated as a special case of UNION. A forest F_n is naturally

constructed as the result of a sequence of n INSERT operations. The number of

comparisons required by this construction is equal, on one hand to the number of

carries propagated in the addition $\underbrace{1 + 1 + \ldots + 1}_{n \text{ times}}$, and, on the other hand, to

the number of edges in the graph of F_n. If ν (n) denotes the number of ones in the

binary decomposition of n, this last number is clearly equal to n - ν (n). It follows

that F_n is constructed in $n-\nu(n)$ comparisons which is of linear order $O(n)$.

As for UNION (n, n') exactly $\nu(n)+\nu(n')-\nu(n+n')$ comparisons are required,

which is of order $O(\log (n + n'))$.

In order to find the minimal label of F_n, we merely have to explore the

roots of the binomial trees composing F_n, and keep the name of a node having

minimal label among these. This involves $\nu(n)$ - 1 comparisons, which is of order

$O(\log n)$.

In many applications, DELETE is restricted to extracting the item m found

by MIN. Let B_i be the labeled binomial tree in F_n of which m is the root. We

first remove B_i from F_n, thus forming a labeled binomial forest F_{n_1} with

$n_1 = n - 2^i$. Then the root of B_i is cut. As we can see from Figure 2, what

remains is a "complete" forest F_{n_2} with $n_2 = 2^i - 1$. One then call UNION (F_{n_1}, F_{n_2})

in order to restore F_{n-1} in $O(\log n)$ comparisons again. This procedure is

described on the example below (Figure 5).

$$F_n = $$

(a) A labeled F_{12} ;

$$F_{n_1} = $$

$$F_{n_2} = $$

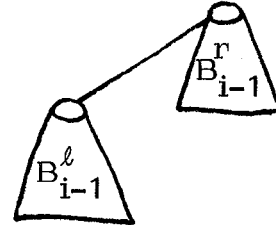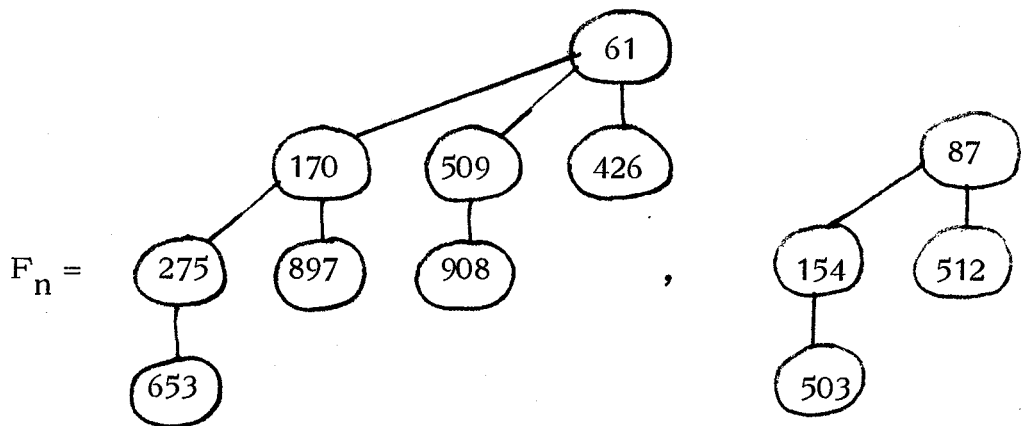(b) Broken up into a F_4 and a F_7 after removal of 61 ;

$$F_{n-1} = $$
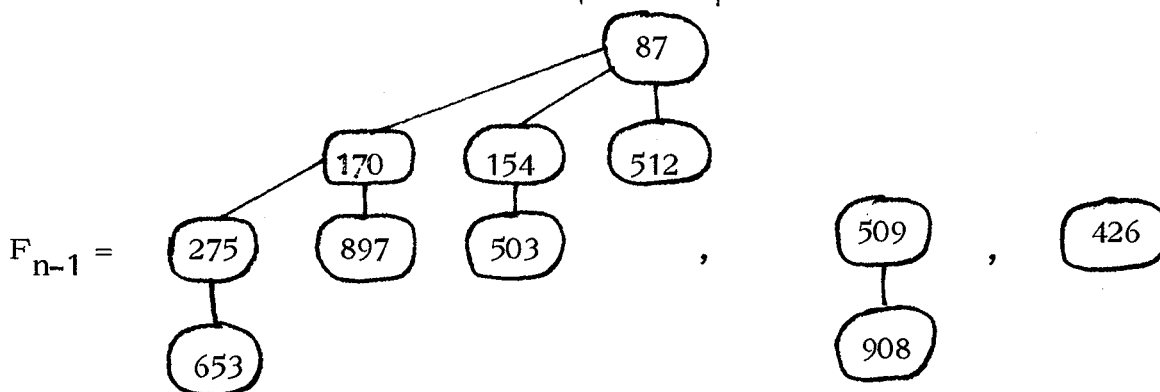
(c) $F_{11} = $ UNION (F_4, F_7) reconstituted.

Figure 5

If the item m to be removed by DELETE is not the root of one of the

components of F_n, the algorithm is slightly more complex. First, we determine

the component of F_n in which m lies, say B_i. As before, we remove B_i thus

forming F_{n_1} with $n_1 = n - 2^i$. We then consider $B_i =$

If m is in B_{i-1}^{ℓ}, we start constructing $F_{n_2} = \{B_{i-1}^{r}\}$ and decompose B_{i-1} by the

same technique ; otherwise, we further decompose B_{i-1}^{r} and set F_{n_2} to B_{i-1}^{ℓ}.

This continues until m becomes the root of the subtree B_j to be decomposed.

It is then "cut" from B_j, thus leaving a complete F_{2^j-1} which is added to the

binomial queue $F_{n_2} = \{B_{i-1}, B_{i-2}, \ldots, B_j\}$ already constructed in order to

form a complete F_{2^i-1}. This forest is then merged with F_{n_1}, using the UNION

procedure, in order to construct the resulting F_{n-1}. Again, this algorithm is

illustrated by an example in Figure 6.

(a) A labeled F_{12} ;

(b) Broken up into a F_4 and a F_7 after removal of 170 ;

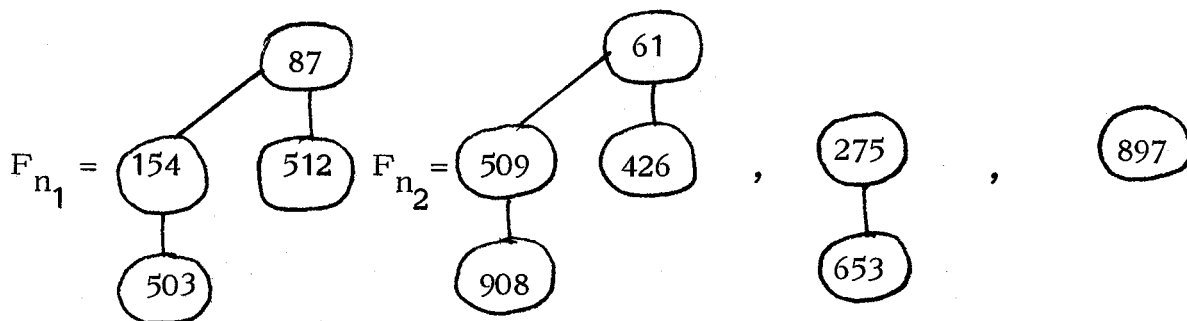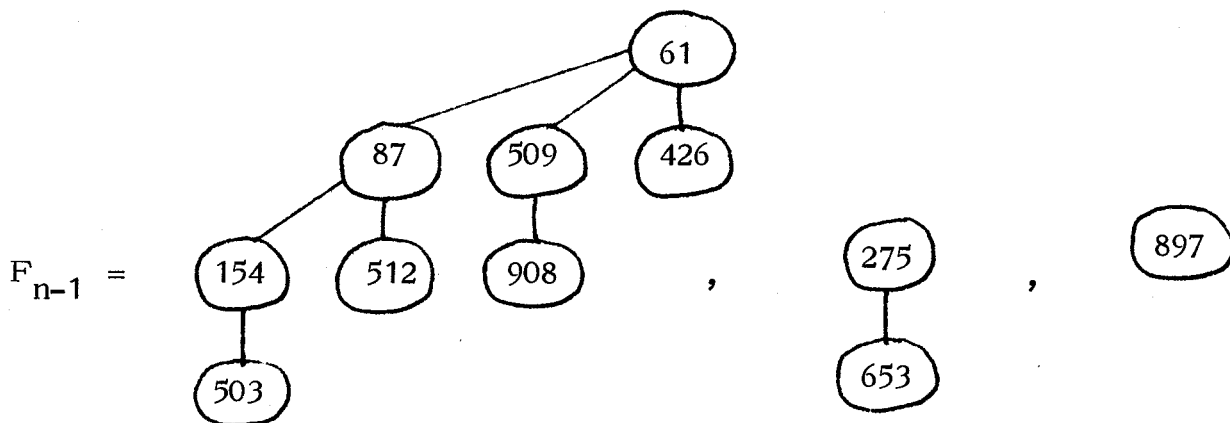

(c) F_{11} = UNION (F_4, F_7) reconstituted

Figure 6

As for our last primitive, UPDATE, the obvious way to realize it is to perform DELETE then INSERT in sequence. This requires $O(\log n)$ operations for each UPDATE . If we have to service an arbitrary sequence of primitive operations in which MIN is required less often than UPDATE and DELETE, there is a better way to proceed. The procedure UPDATE does not 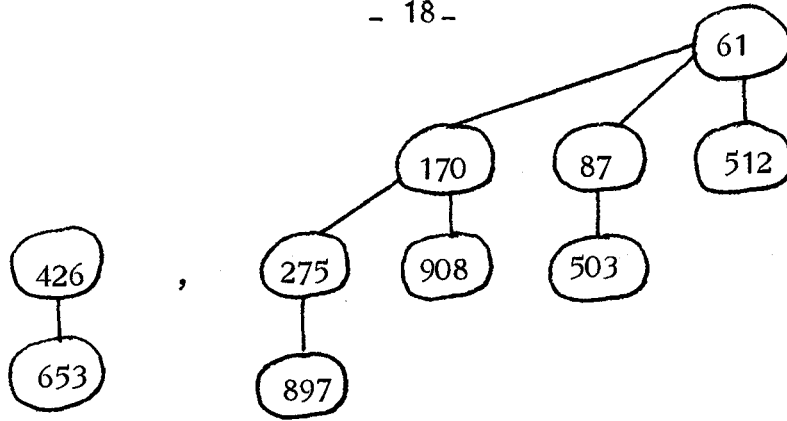attempt to restore the structure, but merely changes the label and marks the node. The DELETE procedure proceeds in the same way, changing the label to, say $+ \infty$. The algorithm for MIN then becomes more complicated : it explores all binomial trees in the forest F_n until finding unmarked nodes on all paths from the root of the tree to the leaves ; this cuts some subtrees and the forest is then reconstructed by merging all these subtrees together with the marked nodes having labels different from $+ \infty$. If μ marked nodes are met, it follows (see V [25]) from the properties described in section 2 that the number of trees cut is a most $\mu \log \frac{n}{\mu}$. If we consider an arbitrary sequence of n INSERT or UNION, u UPDATE or DELETE, and m MIN, an elementary analysis (see V [25] again) shows that such a sequence is treated in at most $O(n \log n) + O(u) + O(u \log \frac{n\,m}{u})$ operations which is better than the naïve method for $m \le u$.

Yet another way of treating UPDATE is described in J [14] .

4. IMPLEMENTATION OF BINOMIAL FORESTS AS BINARY TREES.

While the above discussion constitutes an adequate presentation of the priority queue primitives if our machine model is rather abstract (decision trees for example) it is by no means complete if one is to actually code these algorithms on a real life computer. One still has to solve some problems, the first of which concerns the machine representation of labeled binomial forests.

For this purpose, we represent binomial forests as binary trees through the well known "natural correspondence" described in K [16] : each node has fields llink and rlink such that llink points to the leftmost child of the node and rlink to the node's right sibling. This leaves some freedom for defining a node's right sibling. For the purpose of the UNION procedure, it is crucial to link small trees to larger siblings on the top level, i.e. for nodes having no parent, and to link large trees to smaller siblings on lower levels, as shown in Figure 7.

(a) A binomial queue F_{10} ;
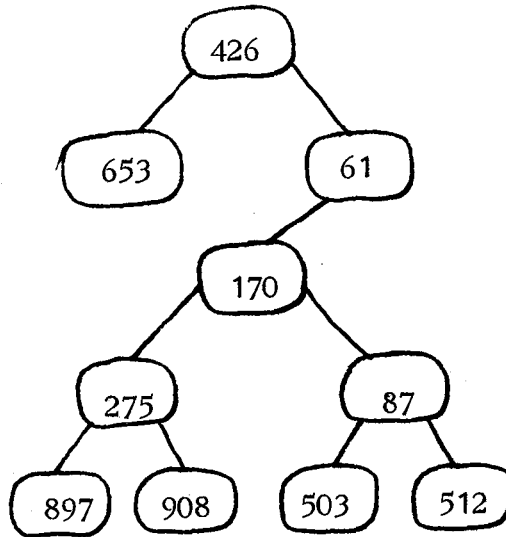
(b) The same F_{10} as a binary tree ;

	INFO	LLINK	RLINK
1	512	0	0
2	87	9	1
3	61	10	0
4	426	6	3
5	275	7	8
6	653	0	0
7	897	0	0
8	908	0	0
9	503	0	0
10	170	5	2

R

| 4 |

N

| 10 |

(c) A possible machine representation for this binary tree, using arrays ;
memory cells R and N contain respectively a pointer to the root of the tree
and its number of elements.

Figure 7 .

We can now give a formal description of our algorithms in an Algol-like language. Following Knuth (K [16]), we represent binary tree with three arrays INFO, LLINK, RLINK containing respectively the label, llink and rlink of the node under consideration. The value 0 represents an empty pointer.

We first describe the UNION procedure :

proc UNION(R1,N1,R2,N2)→(R3,N3):

{This procedure merges the two binomial queues F^1 and F^2, yielding F^3 for

result. Each F^i is represented as a binary tree : R_i is a pointer to its

root and N_i represents the number of elements in F^i.

The initial carry C is zero. Variable RES points to the part of F^3 being

currently constructed. Location RLINK[0] is used to keep R_3 and thus

should be available upon calling the procedure.}

(N3,C,RES)←(N1+N2,0,0);

while (min(N1,N2)≠0)∨(C≠0)

do NEXTBIT;(N1,N2)←($\lfloor N1/2 \rfloor$,$\lfloor N2/2 \rfloor$) od;

RLINK[RES]←if N1≠0 then R1 else R2 fi;

R3←RLINK[0]

endproc UNION.

Here NEXTBIT stands for a fragment of program that treats the eight

possible cases for the carry and the relevant component of F^1 and F^2 :

<u>macro</u> NEXTBIT :

 BITC←<u>if</u> C=0 <u>then</u> 0 <u>else</u> 1 <u>fi</u>;

 <u>case</u> < N1 <u>mod</u> 2,N2 <u>mod</u> 2,BITC>

  000:

  001: (C,RES,RLINK[RES])←(0,C,C)

  010: PROPRES(R2)

  011: PROPCARRY(R2)

  100: PROPRES(R1)

  101: PROPCARRY(R1)

  110: CONSTRUCTCARRY

  111: PROPRES(R1);PROPCARRY(R2)

 <u>endcase</u>

<u>endmacro</u> NEXTBIT.

  We then have to describe the various macro procedures composing NEXTBIT :

<u>macro</u> PROPRES(R):

{The number of bits is odd and one of then, namely R, must be added to the result.}

 (R,RES,RLINK[RES])←(RLINK[R],R,R)

<u>endmacro</u> PROPRES.

<u>macro</u> PROPCARRY(R):

{Bit R must be added to the carry and the carry propagated.}

<u>if</u> INFO[R]< INFO[C]

 <u>then</u> (C,R,LLINK[R],RLINK[C])←(R,RLINK[R],C,LLINK[R])

 <u>else</u> (R,LLINK[C],RLINK[R])←(RLINK[R],R,LLINK[C])

<u>fi</u>

<u>endmacro</u> PROPCARRY.

<u>macro</u> CONSTRUCTCARRY :

{Bits R1 and R2 are on and a carry must be constructed.}

<u>if</u> INFO[R1]< INFO[R2]

  then (C,R1,R2,LLINK[R1],RLINK[R2])←(R1,RLINK[R1],RLINK[R2],R2,LLINK[R1])

  <u>else</u> (C,R2,R1,LLINK[R2],RLINK[R1])←(R2,RLINK[R2],RLINK[R1],R1,LLINK[R2])

<u>fi</u>

<u>endmacro</u> CONSTRUCTCARRY.

This terminates the description of UNION. The procedure MIN being straight-forward, we omit its description. (If very frequent uses of MIN are requested, we can keep the value of the minimal label in a special register.)

As for DELETE, we simply treat EXTRACTMIN where the element having least label is first found, then removed. (The DELETE procedure for which we give no formal code, is very similar. A little complication arises from the necessity of keeping and updating upward parent links.)

<u>proc</u> EXTRACTMIN(R,N)→(R',N'):

{This procedure extracts from the non−empty labeled binomial forest F, the

element having minimal label. The resulting forest F' is obtained by merging

two forests F^1 and F^2 which we first construct.}

(M,PRED,R2,P,R)← (R,0,R,R,RLINK[R]);

<u>while</u> R$\neq$0

<u>do</u>   <u>if</u> INFO[R]< INFO[M] <u>then</u> (M,PRED)← (R,P) <u>fi</u>;

         (R,P)← (RLINK[R],R)

<u>od</u>   {INFO[M] is the minimal label in F.}

(R1,N1)← CONSTRUCT(LLINK[M]); N2← N−N1−1;

<u>if</u> PRED=0 <u>then</u> R2← RLINK[M] <u>else</u> RLINK[PRED]← RLINK[M] <u>fi</u>;

(R',N')← UNION(R1,N1,R2,N2)

<u>endproc</u> EXTRACTMIN.

The procedure EXTRACTMIN uses CONSTRUCT which transforme the

binary tree representations of a B_p into the binary tree representation of the complete

forest F_{2^p-1} obtained by removing the root of B_p.

<u>macro</u> CONSTRUCT(RAC)→(R,N):

<u>if</u> RAC=0

  <u>then</u> (R,N)← (0,0)

  <u>else</u> (R,SUCC,RLINK[RAC],P)← (RAC,RLINK[RAC],0,1);

      <u>while</u> SUCC$\neq$0

      <u>do</u> (R,SUCC,RLINK[SUCC],P)← (SUCC,RLINK[SUCC],R,2×P) <u>od</u>;

      N←2×P−1

<u>fi</u>

<u>endmacro</u> CONSTRUCT.

5. REFERENCES, BIBLIOGRAPHY.

AL   [1]   : Adel'son - Vel'skii G.M. and Landis Y.M.
                  An Algorithm for the Organisation of Information.
                  Dokl. Akad. Nauk. SSSR 146, 1962, p. 263-266.

AHU   [2]   : Aho A.V., Hopcroft J.E. and Ullman J.D.
                  The Design and Analysis of Computer Algorithms.
                  Addison-Wesley, Reading, Mass., 1974.

CTY   [3]   : Cheriton D., Tarjan R.E., Yao A.C.
                  Finding Minimum Spanning Trees.
                  Research. Rep. Computer Science, Stanford U., 1975.

C   [4]   : Crane C.A.
                  Linear Lists and Priority Queues as Balanced Binary Trees.
                  Rep. Stan-CS-72-259, Dpmt of Comp. Sci., Stanford U. 1972.

D   [5]   : Dijkstra E.W.
                  A Note on Two Problems in Connexion with Graphs.
                  Num. Math. 1, 1959, p. 269-271.

F   [6]   : Fisher M.J.
                  Efficiency of Equivalence Algorithms
                  in Miller R.E. , Thatcher J.W. (eds.)
                  Complexity of Computer Computations.
                  Plenum Press, New York, 1972, p. 158-168.

F   [7]   : Floyd R.W.
                  Algorithm 245.
                  Comm. ACM 7, 12, 1964, p. 701.

FJ   [8]   : Ford L.R. and Johnson S.M.
                  A Tournament Problem.
                  Amer. Math. Monthly 66, 1959, p. 387-389.

F   [9]   : Françon J.
                  Représentation d'une File de Priorités par un Arbre Binaire.
                  Université de Strasbourg, Dpmt. de Maths., Rapport 1975.

G   [10]: Gentleman W.M.
                  Row Elimination for Solving Sparse Linear Systems and Least Squares
                  Problems.
                  Dundee Biennial Conf. on Numerical Analysis, 1975.

G   [11] : Gonnet G.H.
                  Heaps Applied to Event Driven Mechanisms.
                  Comm. ACM 19, 7, 1976, p. 417-418.

GR   [12] : Gonnet G.H. and Rogers L.D.
                  An Algorithmic and Complexity Analysis of the Heap as a Data Structure.
                  Research Rep. CS 75-20. U. of Waterloo, 1975.

J   [13] : Johnson D.B.
Algorithms for Shortest Paths.
Ph. D. Thesis, Cornell U., Ithaca N.Y., 1973.

J   [14] : Johnson D.B.
Priority Queues with Update and Finding Minimum Spanning Trees.
Penn. State U. Tech. Rep. 170, 1975.

JD   [15] : Jonassen A. and Dahl O.J.
Analysis of an Algorithm for Priority Queue Administration.
BIT 15, 1975, p. 409-422.

K   [16] : Knuth D.E.
The Art of Computer Programming Vol. 1.
Addison-Wesley, Reading Mass., 1968.

K   [17] : Knuth D.E.
The Art of Computer Programming Vol. 3.
Addison Wesley, Reading Mass., 1973.

K   [18] : Knuth D.E.
Selected Topics in Computer Science.
Lecture Notes Series, Institute of Mathematics, U. of Oslo, 1973.

MS   [19] : Malcolm M.A. and Simpson R.B.
Local versus Global Strategies for Adaptative Quadrature.
ACM Trans. Math. Soft. 1, 2, 1975, p. 129-146.

PPS   [20] : Paterson M., Pippenger N., Schönhage A.
Finding the Median.
U. of Warwick Theory of Comp. Rep. n⁰ 6, 1975.

PS   [21] : Porter T. and Simon I.
Random Insertion into a Priority Queue Structure.
Rep. Stan-Cs-74-460, Dpmt. of Comp. Sci. Stanford U. 1974.

P   [22] : Prim R.C.
Shortest Connection Networks and Some Generalizations.
Bell. Sys. Tech. J. 36, 6, 1957, p. 1389-1401.

VKZ   [23] : Van Emde Boas P., Kaas R. and Zijlstra E.
Design and Implementation of an Efficient Priority Queue.
Mathematisch Centrum Report ZW 60/75, Amsterdam 1975.

VD   [24] : Vaucher J.G. and Duval P.
A Comparison of Simulation Event List Algorithms.
Comm. ACM 4, 18, 1975, p. 223-230.

V   [25] : Vuillemin J.
Structures de Données.
Notes de cours de l'école d'été CEA-EDF-IRIA 1975.

W   [26] : Williams J.W.J.
                Algorithm 232.
                Comm. ACM 7, 6, 1964, p. 347-348.

W   [27]: Wyman F.P.
                Improved Event-Scanning Mechanisms for Discrete Event Simulation.
                Comm. ACM 18, 6, 1975, p. 350-353.