Description et analyse d'une representation performante des files de priorite

Jean Francon, Jean Vuillemin, Gérard Viennot

IRIA
laboria

Institut de Recherche
d'Informatique
et d'Automatique

Domaine de Voluceau
Rocquencourt
B. P. 105 78150 - Le Chesnay
France
Tél.: 954 90 20

laboratoire de recherche
en informatique
et automatique

DESCRIPTION ET ANALYSE D'UNE REPRÉSENTATION PERFORMANTE DES FILES DE PRIORITÉ

Jean FRANÇON
Gérard VIENNOT
Jean VUILLEMIN

Rapport de Recherche N° 287

Avril 1978

9p

DESCRIPTION ET ANALYSE D'UNE REPRÉSENTATION PERFORMANTE DES FILES DE PRIORITÉS

Jean Françon*, Gérard Viennot, Jean Vuillemin*****

Résumé :

Nous étudions diverses structures de données et algorithmes permettant de représenter des **files de priorité.** Une analyse détaillée montre que l'une d'entre elles, **la pagode,** permet une implantation **très efficace en moyenne,** et est directement utilisable pour les nombreuses applications des files de priorité. Elle permet de traiter une suite arbitraire de n opérations choisies parmi MIN, INSERE, UNION, OTE, OTEMIN en temps O(n log n) ; une suite de n opérations MIN, INSERE et OTE est traitée en temps moyen linéaire O(n) ; une suite de n adjonctions INSERE est traitée en O(n) opérations dans le cas le pire.

Abstract :

New data structures and algorithms for manipulating priority queues are presented. A detailled analysis shows that one of the proposed structures, the pagoda, is a very efficient implementation of priority queues, in the average case sense. It handles a sequence of n primitive operations chosen from MIN, INSERT, UNION, EXTRACT and EXTRACTMIN in O(n log n) average time ; a sequence of n primitives MIN, INSERT and EXTRACT is treated in linear average time O(n) ; n successive INSERT are treated in linear time O(n) in the worst case.

* Centre de Calcul du C.N.R.S. B.P. 20 CR, 67037 Strasbourg.

** Informatique théorique et programmation. LA 248 C.N.R.S. - 2, place Jussieu 75221 Paris Cédex 05.

***Laboratoire de Recherche en Informatique, bât. 490 - Université Paris-Sud, 91405 Orsay.

Abstract :

We present a new data-structure for representing priority queues, *the pagoda*. A detailled analysis shows that the pagoda provides a very efficient implementation of priority queues, where our measure of efficiency is the *average run time* of the various algorithms. It handles an arbitrary sequence of n primitive operations chosen from MIN, INSERT, UNION, EXTRACT and EXTRACTMIN in time O(n log n). The constant factors affecting these asymptotic run time are small enough to make the pagoda competitive with any other priority queue, including structures which cannot handle UNION or EXTRACT.

The given algorithms process an arbitrary sequence of n operations MIN, INSERT and EXTRACT in *linear average time O(n)*, and a sequence of n INSERT in linear worst case time O(n).

1. Introduction

1.1. Priority queues are one of the fundamental data-structures of Computer Science. Several non trivial organizations for representing priority queues have been proposed, and attempts have been made to inventory some of their numerous applications. A clear account of the state of the art on priority queues is given by Knuth 73 and Brown 77.

A *priority queue* is a set Q of keys ; each key k∈Q has an associated priority p(k) which is an arbitrary integer p(k)∈Z.

Typical applications of priority queues require to perform an arbitrary sequence of primitive operations chosen among :

1) MIN : r←min(Q) ; this primitive returns in register r the location of the key in Q having least priority.
2) INSERT : Q←Q+k ; adds key k to queue Q.
3) EXTRACT : Q←Q\k ; extracts key k from queue Q ; this operation is meaningfull only if k∈Q.
4) EXTRACTMIN : Q←Q\min ; extracts from the non-empty queue Q the key having least priority.
5) UNION : Q←Q+Q' ; all the elements of Q' are added to Q and Q' is destroyed.

Some applications do not require UNION and implementations where this primitive cannot be coded in a natural way are refered to as non-mergeable priority queues.

To avoid cumbersome notations, we identify a priority queue Q with the set of priorities of its keys. Strictly speaking, this is a set with repetitions since priorities need not be all distincts ; it is convenient to ignore this technicality and assume distinct priorities.

1.2. A detailled *à la Knuth* analysis and comparison between the various priority queues implementation has been carried out by Brown 77 who finds :

1) Sorted linear lists are the best implementation of small priority queues, say of size less than 20.
2) Heaps (Williams 64, Knuth 73) are the best implementation of non-mergeable priority queues.
3) Binomial queues (Vuillemin 76) present the most efficient known implementation of (mergeable) priority queues.

1. Centre de Calcul du CNRS, B.P. 20 CR, 67037 Strasbourg.

2. Informatique Théorique et Programmation, LA 248 CNRS 2, place Jussieu, 75221 Paris Cedex 05.

Brown's conclusion are not affected by the present paper, if one considers *worst case running time* as one's measure of efficiency. If one consider *average run time*, which, for many applications is a more realistic measure of efficiency, the pagoda structure described in this paper becomes very important :

1) Sorted linear lists remain the best choice for priority queues of size less than 10.

2) Pagodas provide the most efficient average time implementation for priority queues ; for non-mergeable priority queues, performances are close to those of heaps, with advantages and disadvantages for both structures : heaps use less memory but require contiguous storage allocation ; heaps have a guaranteed O(log n) worst case for insertion versus O(n) worst case for pagodas ; on the other hand, the worst case for n successive insertions is O(n log n) in a heap and O(n) in a pagoda. Pagodas allow to merge priority queues, an operation which is not possible with heaps. In both pagodas and heaps, one can EXTRACT an arbitrary key, designated by its address in the structure, in average a time O(1). Both structures thus allow *to handle an arbitrary sequence of MIN, INSERT and EXTRACT in average linear time.*

1.3. Pagodas where discovered in the course of a more ambitious research program : study the combinatorial objects counted by the factorial n! and Catalan $\frac{1}{n+1}\binom{2n}{n}$ numbers, their machine representations and associated coding and decoding algorithms. This program stems from *combinatorial geometry*, which attemps to construct *explicit bijections* between combinatorial structures enumerated by the same numbers, and study these bijections in order to geometrically explain remarkable identities and their properties. There are intimate links between combinatorial geometry and computer science since combinatorial structures can be used for the computer representations of the mathematical object enumerated, while the coding and decoding associated algorithms provide the seeked bijection.

The part of this research pertaining to priority queues is described in Françon et al.79, where a much more comprehensive presentation of the results given here can be found, with applications to binary search trees (see Françon 76).

1.4. In section 2 of this paper, we introduce the combinatorial tools pertaining to permutations and binary tournaments ; these combinatorial results are a guide to the overall strategy of the priority queue algorithms (bottom-up merge), the low-level machine coding (order of comparisons in the insertion) and the analysis of the algorithms produced.

In section 3, we describe the data structure used and the five algorithms representing the priority queue primitives. To substanciate the claim made in 1.2, a carefull machine coding of INSERT yields $\overline{T}_I(n)=18{,}5$ for the average insertion time and $\overline{T}_E(n)=13{,}8\ \log_2 n+8$ for the average EXTRACTMIN. Equally carefull implementations for heaps yield $\overline{T}_I(n)=17$ and $\overline{T}_E(n)=10\ \log_2 n+1$ for the average insert and extractmin (these are experimental run times since no exact analysis is known of the average behaviour of heaps).

We also discuss theoretical arguments indicating the *optimality* of repeated insertions in a pagoda, both from the average and worst case point of view.

2. Priority queues as ordered sequences and binary tournaments.

2.1. The most obvious representation of priority queues is probably a sequence $s=(p_1,p_2,\ldots,p_n)$ of the priorities, kept in their order of arrival.

Union is then concatenation of the associated sequences ; insertion, deletion and deletemin are obvious enough. The only problem here is to find the minimum, which may require O(n) operation.

Nethertheless if we consider the n! possible orders of arrival of the p_i's to be equally likely, and define a random sequence as associated to a random order of the p_i's, we can state, in an obvious manner :

Proposition 1 :

If $s=(s_1,\ldots,s_n)$ and $r=(r_1,\ldots,r_m)$ are independant random sequences of priorities, then

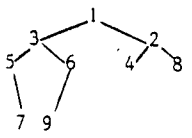
- $s.r=(s_1,\ldots,s_n,r_1,\ldots,r_m)$,
- $s.r_1$,
- $s\backslash si$ for $1\leq i\leq n$,
- $s\backslash \min(s)$

are random sequences of priorities.

2.2. To each sequence $s=(s_1,\ldots,s_n)$ of priorities, we associate a binary tree $\delta(s)$ by the following rules : let $m=\min(s)$ and write $s=g\ m\ d$; the binary tree $\delta(s)$ has m for root, $\delta(g)$ for left subtree and $\delta(d)$ for right subtree ; by convention $\delta(\Lambda)=\emptyset$.

For example,

$\delta(573961428) =$



Such binary trees have the characteristic property that labels (priorities) increase as we go down any path in the tree ; various names have been proposed for such trees : heaps, increasing trees, ... and we choose to call them *binary tournaments*.

Conversely, to any binary tournament T, we can associate the sequence $\pi(T)$ of its priorities taken in symetric order. *It is easily seen that operations* δ *and* π *are respective inverses.* (See also Foata - Schutzenberger 71, Burge 72, Françon 76, Viennot 76).

We shall use binary tournaments to represent priority queues, and perform all operations as if we were dealing with ordered sequences of priorities, although we are really manipulating trees. It follows from Proposition 1 that :

Proposition 2 :

If S and R are independant random binary tournaments,
- $\delta(\pi(S).\pi(R))$,
- $\delta(\pi(S).r_1)$
- $\delta(\pi(S)\backslash s_i)$ for $1\leq i\leq n$,
- $\delta(\pi(S)\backslash \underline{\min})$

are random binary tournaments.

This straightforward result is the key to the analysis that follow.

Working with binary tournaments has the obvious advantage of permitting to find the minimum in time O(1). The formula $\delta(\pi(S).\pi(R))$ on the other hand does not provide an efficient implementation of the union procedure. In order to design such an efficient implementation, we need to study more properties of δ. Let $s=(s_1,\ldots,s_n)$ be a sequence of n priorities, also called *word* of length n, and $x=s_i$ be a priority of s, also called *letter* of s. We can factor in a unique way s as

$$s=u\lambda(x)\ x\ \rho(x)v$$

where all letters of $\lambda(x)$ and $\rho(x)$ are $>x$, the first letter of v and last letter of u being $<x$ (if it exists).

This is called the *x factorisation of s* by Foata-Strehl 74. For example, the 2-factorisation of (573961428) is (573961) (4) (2) (8) ( ) and its 3-factorisation is ( ) (57) (3) (96) (1428).

Once this concept is understood, it is easily seen that :

Proposition 3 :

If $u\lambda(x)x\ \rho(x)v$ is the x-factorisation of s, the subtree of root x in $\delta(s)$ has $\delta(\lambda(x))$ for left-subtree and $\delta(\rho(x))$ for right subtree.

If s is a word of length n, the letter $x\in s$ is a *left to right minima* (resp. right to left) if the x factorisation $u\lambda(x)x\ \rho(x)v$ has $\lambda(x)=\Lambda$ the empty word (resp. $\rho(x)=\Lambda$). We write $x\in lrm(s)$ and $x\in rlm(s)$ respectively.

If T is a binary tournament, its *right branch rb(T)* is the increasing sequence of priorities found on the path starting at the root of T and repeatedly going to the right subtree ; left branch lb(T) is defined in a symetric manner : the *bottom* of rb(T) is the node having no right son.

Proposition 4 :

Application δ is a bijection mapping lrm(s) onto $lb(\delta(s))$ and rlm(s) onto $rb(\delta(s))$. The average value of lb(T) and rb(T) over all n! binary tournaments T is :

$$\text{ave } lb(T)=\text{ave } rb(T)=H_n=1+\frac{1}{2}+\ldots+\frac{1}{n}.$$

The average value of rb(T) follows from the well-known average value of rlm(s) ; its distribution is given by Stirling numbers of the first kind, noted $\left[{n \atop k}\right]$ by Knuth 69.

Proposition 3 characterizes the subtree of root x in $\delta(s)$; in order to characterize the path Ch(1,x) from the root to x in $\delta(s)$, we note $rlm(x)=rlm(s_1s_2..x)$ and $lrm(x)=lrm(x...s_n)$.

A letter $y<x$ is in rlm(x) iff we can write $s=uyvxw$ with all the letters in v greater than y. This is equivalent to say $x\in\rho(y)$ relatively to the y-factorisation of s. By proposition 3, this is telling us that x belongs to the right subtree of y in $\delta(s)$. Arguing symetrically for lrm(x), we conclude :

Proposition 5 :

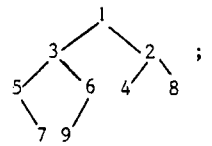
The path Ch(1,x) from the root to x in the binary tournament $\delta(s)$ consists of

$$Ch(1,x)=lrm(x)\cup rlm(x).$$

Furthermore, any $y\in Ch(1,x)$, $y\neq x$ has a left son in Ch(1,x) iff $y\in lrm(x)$.

We illustrate propositions 3, 4 and 5 on the example belows :

$s=(573961428)$; $\delta(s)=$



;

$lrm(s)=(531)=lb(\delta(s))$, $rlm(s)=(821)=rb(\delta(s))$;
6-factorisation of $s=(573)\ (9)\ 6\ (1428)$: the subtree of root 6 in $\delta(s)$ is $\delta(96)$.
$lrm(6)=(61)$, $rlm(6)=(63)$, $Ch(1,6)=(136)$.

Proposition 5 has an interesting consequence :

Proposition 6 :

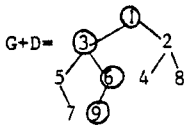
The union $\delta(rs)$ of two binary tournaments $\delta(r)$ and $\delta(s)$ can be constructed by considering the sorted sequence $(x_1\ldots x_k)=rb(r)\cup lb(s)$ obtained by merging rb(r) and lb(s). The union $\delta(rs)$ is the binary tourna-

ment of root x_1 and for $1 \le i < k$, if $x_i \in r$ (resp. $x_i \in s$) its right son (resp. left son) is x_{i+1} and its left (resp. right) subtree is the left (resp. right) subtree it posesses in $\delta(r)$ (resp. $\delta(s)$).

Thus union of binary tournaments G and D reduces to a *merge* of the two sorted sequences rb(G) and lb(D).

For example if G=  and D=  then

lb(D)∪rb(G)=(1 3 6 9) where we underline the elements in D and the union is :

G+D=  .

Here comes the interesting remark : *there are two natural ways to merge rb(G) and lb(D), bottom up i.e from large to small values, or top down i.e from small to large values.*

In order to face this design decision, let us consider the somewhat unatural bottom up solution and analize the number of comparisons performed in the merge of binary tournaments G+D. This number U(G,D) of comparisons is equal to $|rb(G)|+|lb(D)|-|B(G,D)|$ where B(G,D) is the set of priorities in rb(G) which are smaller than all properties in lb(D), together with the priorities in lb(D) smaller than rb(G) (these two cases are of course exclusive).

The left and right branches of the union F=G+D are made of the left branch of G, the right branch of D and B(G,D). It follows that we can express :

Proposition 7 :

The number of comparisons U(G,D) in the union of G and D is given by :

$$U(G,D)=|rb(G)|+|lb(G)|+|rb(D)|+|lb(D)|-|rb(G+D)| \\ -|lb(G+D)|.$$

From this proposition, we can deduce the number of comparisons IBU(G) required for the bottom-up insertion of an element k in G :

$$IBU(G)=2+|rb(G)|+|lb(G)|-|rb(G+k)|-|lb(G+k)|.$$

As for ITD(G), the number of comparisons performed in the top-down insertion of k in G, a direct inspection shows that :

$$ITD(G)=\begin{cases} rb(G+k) & \text{if } |rb(G)|>1, \\ 1 & \text{if } |rb(G)|=1. \end{cases}$$

Combining with proposition 4 yields :

Proposition 8 :

The average numbers $\overline{IBD}(n)$ and $\overline{ITD}(n)$ of comparisons in the bottom-up and top-down insertion in a binary tournament G of size n are :

$$\overline{IBD}(n)=2-\frac{2}{n},$$

and $$\overline{ITD}(n)=H_{n+1}-\frac{1}{2}.$$

In conclusion, *bottom-up insertion requires O(1) operations versus O(log n) for the top down insert.* A complete analysis of bottom-up and top-down strategies is carried out in Françon et al 78 from which it results that both strategies yield comparably efficient algorithms for all the priority queue operations, exept for INSERT where bottom-up is definitely better. In this paper, we thus restrict ourselves to the bottom-up strategy.

In order to carry the analysis of the algorithm in section 3, we need two other results from Françon et al 78.

If G is a binary tournament, we let Branch (G) be the sum, over all subtrees S of G of $|lb(S)|+|rb(S)|-2$.

Proposition 9 :

The average value of Branch (G) over the n! tournaments of size n is

$$\text{ave Branch}(G)=2n-2H_n.$$

Proposition 10 :

The number $A_{n,k}$ of subtrees of size k among the n! binary tournaments of size n is :

1) $A_{n,n}=n!$, and

2) $A_{n,k}=2\frac{(n+1)!}{(k+1)(k+2)}$ for $0 \le k < n$.

3. The pagoda structure and the five basic algorithms.

3.1. The pagoda representation of binary tournaments.

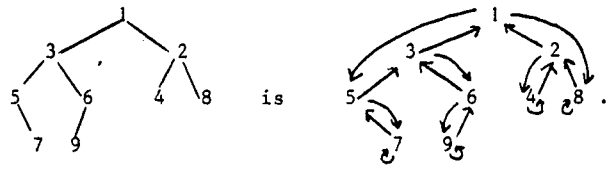
As seen in section 2, efficient processing of binary tournaments requires access to the leaves of the right and left branches. The *pagoda* is a representation of binary trees with *two pointers g and d per node*, which allows for this treatment in a uniform manner ; pointers g and d are defined by the following rules :

(i) Root : if r is the root of the tree T, d(r) points to the bottom of the right branch of T, and g(r) to the bottom of the left branch of T.

(ii) Left son : if k is a left-son in T, g(k) points to the father of k and d(k) to the bottom of the right branch of starting at k.

(iii) Right son : if k is a right son in T, d(k) points to its father and g(k) to the bottom of the left branch starting at k.

As a running example the pagoda representation of

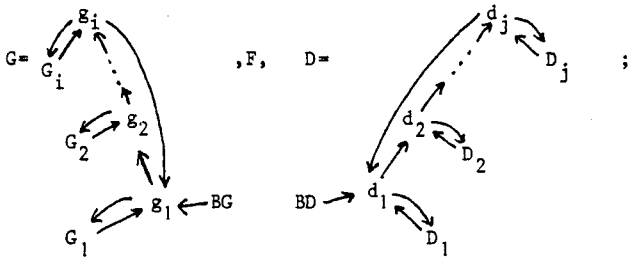


Together with these g and d links, a pointer to the root of the pagoda should be provided, in order to program MIN in the obvious manner with run time O(1).

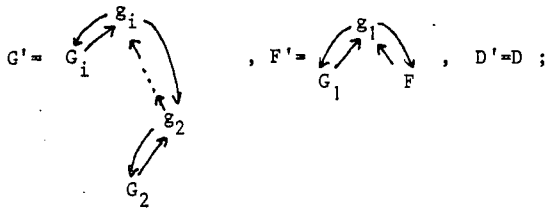
3.2. Union

It is natural to start the description of the four remaining primitives by UNION, since the other three can be reduced to it.

Let G and D be two pagodas which are to be merged into F. Initially, F=∅ and result F is constructed by a bottom-up merge of rb(G) and lb(D). If BG and BD point to the leaves of rb(G) and lb(D), the situation at a typical step of the merge is :



if $g_1 \geq d_1$, the next step is :



the case $g_1 < d_1$ is treated in a symetrical fashion. The algorithm terminates when one of the arguments, say G becomes empty ; it then remains to "glue" the other argument in order to constitute the final result  .

This algorithm can be described by the (non-optimized) code :

```
proc F←G+D
    {pagoda F,G,BG,BD}
    [F,BG,BD,d(G),g(D)]←[∅,d(G),g(D),∅,∅];
    repeat if priority(BG)≥priority(BD) then GAUCHE
                                        else DROITE
        fi ;
    until (BG=∅)∨(BD=∅) ;
    if BG=∅ then [F,g(F),g(D)]←[D,BD,g(F)]
           else [F,d(F),d(G)]←[G,BG,d(F)]
    fi
corp GAUCHE
```

Here GAUCHE stands for the code :

```
if F=∅ then [F,BG,d(BG)]←[BG,d(BG),BG]
       else [F,d(F),BG,d(BG)]←[BG,BG,d(BG),d(F)]
fi
```

and DROITE can be obtained by substituting g for d and BD for BG throughout.

The analysis of this algorithm only depends upon the number U(G,D) of comparisons performed during the merge of the two sorted sequences rb(G) and rb(D). By proposition 7 :

Proposition 11 :

The extremal and average values of the number of comparisons in the union of |G|=n and |D|=m are :

min U(G,D)=*min*(n,m),

max U(G,D)=n+m-1,

ave U(G,D)=$2(H_n+H_m-H_{n+m})$.

3.3. INSERT

By regarding a single key p as a pagoda , we can treat INSERT as the special case of union where m=1. It follows from proposition 11 that :

Proposition 12 :

The number of comparisons I(G) in the insertion in a pagoda G of size n satisfies :

min I(G)=1,

max I(G)=n-1,

ave I(G)=$2-\frac{2}{n}$.

Of course, it is more efficient to code INSERT directly ; to do so, one needs to know the *distribution of I(G)* and a complete analysis described in Françon et al 78 shows that the key to be inserted should be compared first with the leaf on rb(G), then with the root of G, then with the other keys on rb(G).

There is a big discrepancy between the worst case n-1 and the average case $2-\frac{2}{n}$ of the variation of I(G). To understand the meaning of this discrepancy, let us consider the comparisons required for performing n successive INSERT in an initially empty pagoda. A stricking consequence of proposition 7, which is directly proved by induction is :

Proposition 13 :

The number C(G) of comparisons required for constructing a pagoda G of size |G|=n by n successive insertions is

$$C(G)=2n-|lb(G)|-|rb(G)| \ ;$$

thus *min* C(G)=n-1,
max C(G)=2n-3,
ave $C(G)=2n-2H_n$.

Even though some insertions may be very costly, they have the effect of reducing rb(G) thus paving the way for efficient successive insertions. In Françon et al, we prove the optimality of the INSERT procedure for constructing binary tournaments :

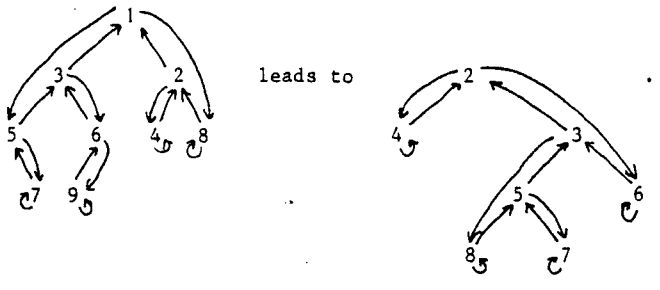
Proposition 14 :

Any on-line algorithm for constructing a binary tournament of size n can be forced to require 2n-3 comparisons in the worst case. Any off-line algorithm for constructing binary tournaments of size n requires $2n-\frac{3}{2}\log_2 n+O(1)$ comparisons on the average.

3.4. EXTRACTMIN

To extract the minimum of pagoda F = 

, we notice that d(m) and g(m) point respectively to rb(D) and lb(G) which is wrong ! We should need rb(G) and lb(D). The cure is easy : exchange G and D and concatenate them in the order D+G. A carefull inspection of the union algorithm shows that a pointer to rb(D) and lb(G) is all we need to start ; termination happens when G or D points to m. For example, the extraction of the minimum of



The EXTRACMIN algorithm thus does not preserve the initial order of elements ; however, the transformation G ↔ D being *bijective, the pagoda obtained after extraction of the minimum in a random pagoda is still random*. We can thus ignore this exchange in the analysis of the number of comparisons EM(F) required, and write EM(F)=U(G,D).

Since |G|+|D|=n-1 and proba(|G|=i|0≤i≤n)=$\frac{1}{n}$, it follows, using the elementary identity $\sum_{i\le n} H_i=(n+1)(H_{n+1}-1)$, that :

Proposition 15 :

The number of comparisons EM(F) in the extraction of the minimum of pagoda F of size |F|=n is :

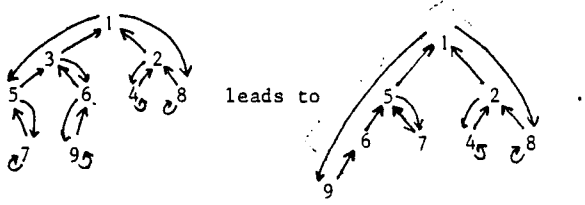
min EM(F)=0,
max EM(F)=n-2
ave $EM(F)=2(H_n-2+\frac{1}{n})$.

3.5. EXTRACT

Extraction of an arbitrary key k, designated by its address in the structure, *is possible in pagodas without extra links*. To extract k, it is sufficient to find links to the right and left branches of the subtrees G and D having k for root. Once these pointers are found, we proceed as in the extraction of the minimum by performing the union of D and G in that order. That randomness is preserved under this operation follows from the same argument as before.

One of the links to D or G can be found in g(k) or d(k) : it is d(k) iff *priority* (d(k)) > *priority* (k) ; in this case, the other link is found by following the upwards links l(k), ll(k),..., until we find a key x for which *priority* (l(x)) > *priority* (x) ; link l(x) is the required link.

For example, extraction of key 3 in the pagoda



The analysis of this algorithm depends upon E(G,k) the number of comparisons and L(G,k) the number of upwards links followed before the first turn in the extraction of k in G. Let $E(G)=\sum_{k\in G} E(G,k)$ and $L(G)=\sum_{k\in G} L(G,k)$. Upwards links are counted for each branch (left or right) in each subtree of G ; it follows that L(G)=Branch(G). This very parameters also appears directly in the analysis of E(G) : we can decompose E(G,k) as the sum of the left and right branches of the subtree of root k in G minus B(G,k), the number of comparisons "gained" in the merge. We can thus write

$E(G)=Branch(G)-\bar{B}(G)$ with $B(G)=\sum_{k\in G} B(G,k)$.

Proposition 9 gives us the average value of Branch(G). To compute ave B(G), use proposition 9 and proposition 10 to obtain :

$$\text{ave } B(G)=\frac{1}{n.n!}\sum_{1\le k\le n} A_{n,k}\left(2-\frac{2}{k}\right)$$

with $A_{n,n}=n!$ and $A_{n,k}=2\frac{(n+1)!}{(k+1)(k+2)}$ for $0\le k\le n$.

This elementary computation yields :

Proposition 16 :

The extremal and average values of the parameters L and E in the extraction of an arbitrary key in a pagoda of size n are :

$$\begin{aligned} &\textit{min } L=0,\\ &\textit{max } L=n-1,\\ &\textit{ave } L=2-\frac{2H_n}{n} \text{ ; and}\\ &\textit{min } E=0,\\ &\textit{max } E=n-2,\\ &\textit{ave } E=1-\frac{2H_n}{n}+\frac{1}{n}. \end{aligned}$$

4. References.

M.R. Brown 77 : "The analysis of a practical and nearly optimal priority queue" Stanford Ph.D. diss. STAN-CS-77-600, Stanford University, 1977.

W.H. Burge 72 : "An analysis of a tree sorting method and some properties of a set of trees" First USA-Japan Computer Conference, 1972.

D. Foata, M.P. Schutzenberger 71 : "Nombres d'Euler et permutations alternantes" University of Florida, Gainesville, Florida, nov. 71.

D. Foata, V. Strehl 74 : "Rearrangements of the symmetric group and enumerative properties of the tangent and secant numbers", Math. Z. 137, 257-264.

J. Françon 76 : "Arbres binaires de recherche : propriétés combinatoires et applications", RAIRO, Informatique Théorique, vol. 10, 12,35-50.

J. Françon, G. Viennot, J. Vuillemin : "Description et analyse d'une représentation performante des files de priorité", à paraître 1979.

C.A.R. Hoare 62 : "Quicksort", Comp. J. 5. 1962.

D. Knuth 69 : "The art of computer programming", vol. 1, Fundamental Algorithms, Addison-Wesley.

D. Knuth 73 : "The art of computer programming", vol. 3, Sorting and Searching, Addison-Wesley.

G. Viennot 76 : "Quelques algorithmes de permutations" in Journées algorithmiques, Paris, S.M.F. Astérisque n° 38-39, 275-293.

J. Vuillemin 76 : "A data structure for manipulating priority queues", Pub. math. d'Orsay, 218-76-67, Departement d'Informatique, 91405 Orsay.