AN ALGORITHMIC AND COMPLEXITY ANALYSIS OF THE HEAP AS A DATA STRUCTURE

Gaston H. Gonnet
and
Lawrence D. Rogers*

Research Report CS-75-20

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

July 1975

(*) Work by this author was supported in part by Canadian National Research Council grant A8984.

Abstract

A heap is defined as a class of data structures used to represent a partially ordered set of data from which we can repeatedly and efficiently extract the maximum (or minimum) element. The definition of the class covers heaps of integer as well as real branch factor. We define and describe the basic operations on a heap and evaluate the running times of various algorithms to perform these operations. The complexity of a heap is defined in a way that proves to be a useful tool for evaluation of running times. We give, in detail, an average case analysis of these algorithms as well as considering the worst case. A relation between probabilistic models of heaps is given and various comparisons of the running times are presented for the most common heap structures. In addition we consider some less common heap structures and describe one heap which resembles the tournament selection process.

Key words and phrases: Data structures, heap, trees, priority queues, analysis of algorithms, complexity analysis, maximum-minimum extraction.

CR Categories: 4.34, 5.25, 5.30, 5.5.

1. - Introduction.

The heap first became widely known as a data structure used in the Heapsort (Williams, 8). Since then it has been applied to a variety of algorithms and applications. Aho (1) and others have used heaps in graph theory, in algorithms for finding the shortest path through a graph and the minimum cost spanning tree of a graph. Malcolm and Simpson (7) have applied it to adaptive quadrature routines and Gentleman (4) has used it in a large sparse matrix elimination application. Heaps have also found use in sparse polynomial multiplication, (Johnson, 5) in both sort and high order merge routines and in generating prime numbers.

In the above applications a heap has generally been defined as a special binary tree with an order relationship between father and son nodes. In this paper we will define a heap to cover a wider range of data structures that includes the binary heap, some ordered trees and, to some extent, the ideas about nth degree selecting structures as described by Friend (3).

The concepts that link these structures together are:

a) the ability to access the maximum or minimum element

directly,

b) an efficient (better than ordered list) method of constructing, extracting and adding elements to the structure,

c) an implementation which allows efficient use of storage and indices, rather than pointers.

The basic operations that we will consider over this data structure are:

a) adding a new element to the heap,

b) extracting the maximum (or minimum) element,

c) constructing a heap from a set of elements.

Note that a heap can be constructed by repeatedly adding elements, but other alternatives have persuaded us to consider the construction algorithm as basic.

This structure is a compromise, in orders of magnitude of average running time, between the unordered list (O(1) to update and O(N) to retrieve maximum or minimum) and the ordered list (O(N) to update, O(1) to retrieve and O(N*lg(N)) to construct). In this paper we will present a model of the complexity of a heap and probabilistic models of the number of comparisons and moves required for the basic operations on a heap, using various algorithms. The estimated number of comparisons and moves are then related

to the complexity function. This relationship shows that the heap is $O(N)$ to construct and $O(\lg N)$ to update (add or extract).

The results are obtained from a probabilistic model of the data. We also show a correspondence between the commonly used models for analyzing this kind of algorithms and the probabilistic model.

2. - Definitions, Notation and Basic Results.

A heap is a data structure defined over a set of nodes such that:

a) there is a unique node called the root;

b) each node i , other than the root, is related to a unique father node j by a function f and an order relationship between i and j .

The order relation is usually either >= or <=. For the remainder of this paper we will assume that the father node is >= to the son, however the results are unchanged if another relation is chosen.

The above definition implies that a heap is a tree defined by the father function f and the order relation. The N nodes of the heap may be numbered breadth first and sequentially starting with the root, numbered 1, and the father function viewed as a function over this numbering.

We now define a uniform heap as one in which the father function is defined by

$$f(i) = \lfloor (i-2)/b \rfloor + 1 \quad (\lfloor x \rfloor \text{ is the floor function})$$

Where b is a constant branch factor for the heap.

This function f has the properties:

$$1 \leq f(i) < i \quad \text{for } i=2,3,\ldots,N,$$

$$f(i) \leq f(j) \quad \text{for } i<j.$$

These properties are important for the implementation of uniform heaps.

If b is an integer, each node will have the same number of sons, except for the leaves and the last non-leaf node. Figure I shows some uniform heaps and their branch factors.

We define the <u>son function</u> <u>s(i)</u> as the inverse of the father function, that is a set function defined by

$$s(i) = \langle j \mid i=f(j) \rangle.$$

For uniform heaps this becomes

$$s(i) = \langle \lceil b*(i-1)+2 \rceil, \ldots, \lceil b*i+1 \rceil \rangle$$

The most common heap is a binary heap. This is a uniform heap with b=2. The father function is then

$$f(i) = \lfloor i/2 \rfloor$$

and the son function is

$$s(i) = \langle 2*i,\ 2*i+1 \rangle .$$

Figure I(a) shows a binary heap. In the extreme case with branch factor b=1, the heap becomes a linear ordered list.

We will call a heap a perfect heap when all the leaves are at the same distance (number of edges to the root) from the root. In figure I, a and c are perfect heaps.

Graphic examples of uniform heaps with branch factors

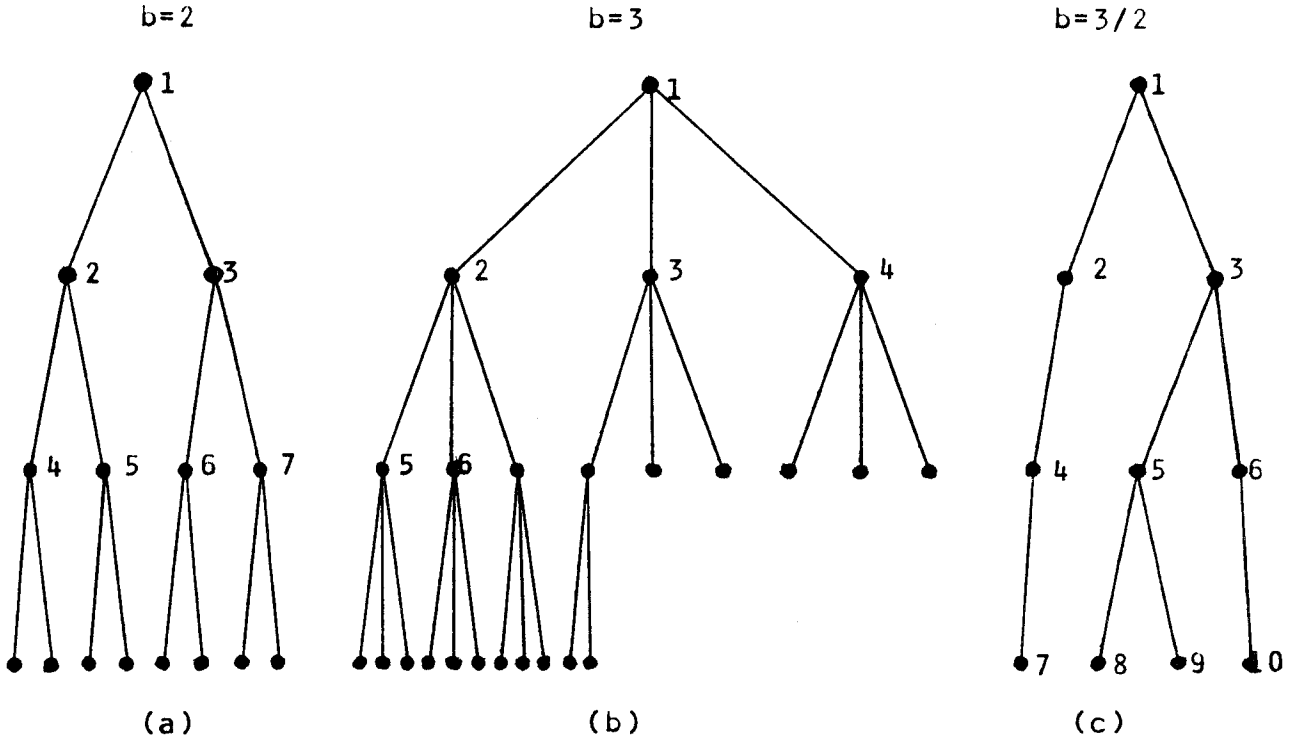


Figure I

2.1 - Notation.

Our general notation will be:

1) f(i) is a function that defines the father of node R(i).

2) s(i) is a set function that defines the sons of node R(i).

3) R(i) represents the ith node in the heap.

4) K(i) is the key value, upon which the order relation is defined, of the ith node. From the definition of the order relation on the heap we have K(f(i)) >= K(i) for every node R(i) that is not the root. This ensures that K(root) is the maximum element of the set.

5) N is the total number of nodes in the heap.

6) b is the branch factor for uniform heaps.

7) n is the height (or depth) of the heap, the maximum distance (number of edges) from the root to a leaf.

8) n(i) is the height of the subheap rooted in node i ; n=n(1).

9) T(i) is the set of nodes of the subheap rooted in i , T(i) = <i> U s(x∈T(i)).

10) t(i) is the number of nodes in the subheap rooted in node i : t(i)=|T(i)|.

11) lg(x) is the logarithm in base 2 of x.

2.2 - Storage Structure.

The fact that the nodes of a uniform heap can be numbered sequentially and the father function specified in terms of these numbers, leads to some important implementation features. We can store the nodes of the heap in sequential storage such that R(1), R(2),... are stored adjacent to each other. This allows nodes to be added and deleted from one end of the storage area and the storage management is as simple as running a stack. Further the nodes can be indexed by father and son functions and hence no pointers are needed.

To describe the storage structure and the algorithms, we will use a pseudo PASCAL language (Wirth, 9). This is motivated by the need of variables, as sets and records, that are not provided by other common languages.

From the above observations we see that the simplest representation of a uniform heap is an array of records R, each having a scalar type key K and further data fields. In PASCAL this is declared as

```
R: array (1..N) of
        record K: <scalar type>;
               data: <any type> end.
```

2.3 - Number of possible heaps.

Several questions arise from the data structure; the first of them is simply how many different heaps, G(N), can be generated from any permutation of the numbers 1,2,...,N.

The answer is quite simple:

$$G(N) = N!/(\prod_{i=1}^{N} t(i))$$

This is easily proved by induction on the subheaps.

2.4 - Complexity of a heap.

The complexity of a heap, H(N), will be defined as an information theoretic value that is the difference between the uncertainty of a random ordered vector and the maximum uncertainty of a heap. This occurs when all possible heaps are equally probable.

Defined in this manner the complexity is a measure of the information gained in constructing a heap from a ramdomly ordered vector. A heap with higher complexity means that it contains more information.

$$H(N) = lg(N!)-lg(G(N)) = \sum_{i=1}^{N} lg(t(i)) \quad \text{(in bits)}.$$

For a perfect heap with integer branch factor this becomes

$$H(N) = \sum_{K=0}^{n} b^{**}k * lg((b^{**}(n+1-k) - 1)/(b-1)),$$

which transforms into

$$H(N) = (N-1)*((2b-1)/(b-1)*lg(b)-lg(b-1)-\beta(b)*(b-1)*b)/b - n/(b-1) -b*\beta(b) + O(b^{**}(-n-1)),$$

where $\beta(b)$ is

$$\beta(b) = \sum_{j=1}^{\infty} 1/((b^{**}(2(j+1)) - b^{**}(j+1)) * j).$$

This shows that the asymptotic behavior of the complexity of a heap as $N \rightarrow \infty$ is linear with respect to N.

This is true only for perfect heaps with integer branch factor, but actual calculation of real values shows that,

for any uniform heap, this formula gives very good approximations.

Some coefficients of (N-1), in the complexity formula, are:

b=2 --> 1.3644365...

b=3 --> .94517688...

b=4 --> .75183245...

b=5 --> .63505844...

2.5 - Lower bounds on comparisons.

The complexity, as defined, is a measure of information gained in the process of constructing a heap from a randomly ordered vector. The algorithms we are going to study are based on comparisons between keys. Since each comparison (using only two possible results) may give at most one bit of information, the complexity is a lower bound of the average number of comparisons needed.

Another obvious minimum of the number of comparisons is N-1, since we obtain the maximum record out of a set of size N.

2.6 - Height of uniform heaps.

For uniform heaps, we may relate the height n to the branch factor b and the number of elements N using parameters a and c :

$$n = \lfloor \log_b((N-c)/a) \rfloor .$$

For integer branch factor

$$a = 1/(b-1) \quad \text{and} \quad -1/(b-1) < c < (b-2)/(b-1).$$

Any c in the specified range will satisfy the equation.

We were not able to find closed forms for a and c as functions of b for non-integer b. Moreover a is discontinuous at infinitely many rational points.

Some values found with a linear programming technique are:

b	a(b)	c(b)
1.5	2.4334	-2.0 to -1.7
1.8	1.5117	-1.2 to -1.1
2.2	1.0354	-.94 to -.40
2.5	0.7646	-.66 to .04

3. - Operations on the data structure.

In this section we will describe the algorithms and running time requirements to perform the basic operations on a heap. More emphasis will be given to the construction algorithms since the extraction and addition are similar to them.

3.1 - Creation.

There are two basic algorithms for heap construction, the Williams (8) method and the Floyd (2) method. In both cases we will assume that the records are stored in an array of size N in which we want to construct the heap.

We state below in PASCAL algorithm W, similar to Williams' method, and algorithm F, similar to Floyd's method.

Algorithm W.

```
for i := 2 to N do
        begin
        aux := R(i);
        j := i;
        while j>1 and K(f(j)) <= Kaux do
                begin
                R(j) := R(f(j));
                j := f(j)
                end;
        if i≠j then  R(j) := aux
        end;
```

We assume that there is a record aux that has a similar definition to R; in particular, it contains a key field Kaux.

The logical expression in "while j>1 and..." should be evaluated from left to right to avoid trying to access R(f(1)), where f(1) is not defined.

Slightly different versions are obtained by stating K(0)=+infinite, and removing the j>1 test in the while statement, or suppressing the "if i≠j" test depending on the efficiency of the "if" with respect to the "assign".

This algorithm creates a heap by repeatedly inserting one element into the tree. The insertion is done in the first free position of the array. This new element is ordered with respect to the elements on the path from the new leaf to the root by a "linear insertion".

Notice that in algorithm W, only the father function is used.

Algorithm F.

```
for i := f(N) downto 1 do
        begin
        aux := R(i);
        j := i;
        while s(j) ≠ null do
                begin
                find m so: K(m) = max(K(x∈s(j)));
                R(j) := R(m);
                j := m
                end;
        while j > i and K(f(j)) < Kaux do
                begin
                R(j) := R(f(j));
                j := f(j)
                end;
        R(j) := aux
        end;
```

The notation:

```
find m so: K(m) = max(K(x∈s(i)))
```

implies that an m is found such that K(m) is the maximum value of the set K(j) where j belongs to the set s(i). This

function will be implemented differently depending on the form of s(i) and the actual value of N. As in algorithm W, the while conditions should be evaluated from left to right.

Note that this procedure uses both functions f and s . This algorithm is slightly different from the original algorithms given by Floyd (2) and Knuth (6). In terms of the number of comparisons this version is more efficient, as we show later.

This algorithm employs an outer loop and two inner loops. The outer loop performs the following function:

```
for i := f(N) downto 1
    make a heap of the nodes presently rooted at
node i.
```

We begin at node f(N), since any node numbered greater than f(N) is a leaf and therefore is already a heap. The node i is not currently in the heap, but each son of i is the root of a subheap. The operation makes a heap of the new node and its subheaps; hence, when the operation is completed for i=1 the heap is constructed.

The operation "make a heap of the nodes presently rooted at node i " must make one heap from the new element at the root node and the heaps rooted at its sons. This is done by extracting the root node and, in the first loop, promoting repeatedly the maximum of the sons to the currently empty position. The second loop then demotes nodes, from the father of the empty position to the empty position, until one greater than the new node is found. The new node then goes in the empty position.

3.2 - Running time analysis of Algorithm W.

Typical running time analyses of algorithms are concerned with the worst case behaviour. The maximum number of node comparisons required to construct a uniform heap using algorithm W is the number of times through the outer loop, N-1, multiplied by the maximum number of times through the inner loop, n, the height of the heap. Hence it is seen that in the worst case we might perform (N-1)n node comparisons.

The worst case analysis is somewhat pessimistic and overestimates the number of comparisons required. For ex-

ample, for a binary heap with 120 nodes the worst case analysis shows we will perform no more than 714 node comparisons. The average analysis shows that we would expect no more than 250 comparisons, a significant difference even in the case of a small heap. Moreover we will find that the average behaviour is O(N) instead of O(N*n).

We will now consider and average case analysis of these two algorithms.

Algorithm W does not generate equally probable heaps from a random permutation, in fact a random permutation of (1,2,3) will give (3,2,1) with probability 1/3 and (3,1,2) with probability 2/3.

We will use random independent variables uniformly distributed between 0 and 1 to model the keys of the nodes in tne heap. Since the algorithms' work is based on comparisons between keys, only the partial ordering of them is relevant. In this case we assume that all partial orderings of the keys are equally probable.

For any function f(i), to evaluate the number of moves and comparisons needed to construct a heap with algorithm W,

we define:

1) X is a random variable U(0,1).

2) X(i,N) is the random variable located in position i in a heap of size N.

3) F(y,i,N) = Prob(X(i,N) <= y) the probability distribution function of the random variable located in position i .

4) Q(i,N) = Prob(X >= X(I,N)).

5) X(i,N) <= X(f(i),N), is the order relation in the heap.

Since X and X(i,N) are independent, we find that

$$Q(i,N) = \int_0^1 F(y,i,N)dy.$$

To calculate the expected value of the number of comparisons when inserting the N+1st element, we will follow the steps of algorithm W. Let i be a node of the heap that belongs to the path from node 1 to N+1; i.e. i=f(...f(N+1)...). If the new element is greater than X(i,N), it will be inserted above in the branch, and hence (unless i=1) it will be compared to the father of X(i,N) namely X(f(i),N). So the new element is compared with prob Q(i,N) with X(f(i),N); (with prob 1 with X(f(N+1),N)) so summing these terms we have

$$E(Cmpr) = 1 + Q(f(N+1),N) + Q(f(f(N+1)),N) + \ldots + Q(1,N) - Q(1,N).$$

Here the $-Q(1,N)$ term is included to simplify the notation.

Similary the number of moves (note that the element is initially in position N+1 of the array) is

$$E(Moves) = 1 + 2*Q(f(N+1),N) + Q(f(f(N+1)),N) + \ldots + Q(1,N).$$

In this case the extra move with Prob $Q(f(N+1),N)$ is the one caused by the statement "if i≠j then R(j) := aux;".

The variance of both quantities can be readily evaluated too.

When the heap grows by one element the only functions that change are the ones that describe distributions of elements that may be modified by the insertion. Conversely we have

$$F(y,i,N+1) = F(y,i,N) \quad \text{iff} \quad i \neq f(f(..f(N+1)..)).$$

If $i = f(f(..f(N+1)..)) \neq 1$, that is i is in the path from the root to node N+1, from the description of the

algorithm we find that

$$X(i,N+1) = \min(\max(X,X(i,N)),\ X(f(i),N))).$$

The distribution function of the variable in such a node is

$$\begin{aligned} F(y,i,N+1) &= \text{Prob}(\min(\max(X,X(i,N)),\ X(f(i),N))<y) \\ &= \text{Prob}(\max(X,X(i,N))<y) + \text{Prob}(X(f(i),N)<y) \\ &\quad - \text{Prob}(\max(X,X(i,N))<y \text{ and } X(f(i),N)<y) \\ &= \text{Prob}(X<y \text{ and } X(i,N)<y) + F(y,f(i),N) - \\ &\quad \text{Prob}(X<y \text{ and } X(i,N)<y \text{ and } X(f(i),N)<y). \end{aligned}$$

Since $X(i,N) <= X(f(i),N)$, we have

$$\begin{aligned} F(y,i,N+1) &= y*F(y,i,N) + F(y,f(i),N) - \text{Prob}(X<y \text{ and } \\ &\quad X(f(i),N)<y) \\ &= y*F(y,i,N) + (1-y)*F(y,f(i),N). \end{aligned}$$

If we analyze the boundary situations, (i.e. $F(y,1,N+1)$ and $F(y,N+1,N+1)$) we find that the same recursive relation holds, provided that we make the natural definitions

$$F(y,f(1),N) = 0, \text{ and } F(y,N+1,N) = 1.$$

Note that the operations involved in the construction of the distribution functions imply that they are polynomials of degree at most N. This allows us to calculate in $O(N*N*n)$ operations the expected number of comparisons and

moves.

The expected values generated from the above shows very close linear relationship to the complexity of a heap. A least squares approximation for 0<N<121 yields:

$$b=2 \quad E(Cmpr) = 1.0608 + 1.6399*H(N) - 1.6480*\ln(H(N))$$

$$b=3 \quad E(Cmpr) = -.3275 + 1.8209*H(N) - 0.7720*\ln(H(N))$$

$$b=4 \quad E(Cmpr) = -.3542 + 2.0530*H(N) - 0.9011*\ln(H(N))$$

The root mean square deviations (RMSD) of each fit were 0.24, 0.21 and 0.30, respectively and the corresponding coefficients of variation were 0.20%, 0.22% and 0.34%.

It may be interesting to explore non integer b in the uniform heaps. Since we can measure the complexity obtained with a particular b and the expected number of comparisons required, we may select the b that maximizes the ratio H(N)/E(Cmpr), which we will call the <u>efficiency</u> of a heap.

Some special values are:

b	N	E(Cmpr)	H(N)	H(N)/E(Cmpr)
1.4	110	326.93	205.95	0.6299
1.6	110	278.66	177.82	0.6381
1.8	110	244.61	155.98	0.6377
2.0	110	229.50	144.22	0.6284
2.2	110	209.63	129.22	0.6164

The maximum efficiency for algorithm W is located near b=1.72 where the efficiency is approximately 0.643. This should be interpreted as maximizing the usefullness of each comparison. Note that the efficiency, as defined, depends on N since E(Cmpr) is not exactly linear on H(N). For different values of N we may have slight changes in the efficiencies and where the maximum is located.

3.3 - Equivalence with other models.

A common model for analyzing these types of algorithms is to consider as an input, a random permutation of 1,2,...,N. The problem is then reduced to one of counting certain configurations.

Let P(i,j,N) be the probability that integer j appears in position i in a heap of size N, resulting from the construction from a random permutation of 1,2,...,N.

Let D(j,y,N) be the distribution function of the jth ordered element from a set of N, U(0,1), random independent variables.

We know that:

$$D(j,y,N) = j\binom{N}{j}\int_0^Y x^{j-1}(1-x)^{N-j}\,dx = Iy(j,N-j+1)$$ (Incomplete Beta function).

Then if we know P(i,j,N), the marginal distribution of the element located in node i is:

$$F(y,i,N) = P(i,*,N) \cdot D(*,y,N)$$

(The "*" notation indicates that we expand to a vector or matrix in that dimension).

Expanding for all nodes in the heap we obtain:

$$F(y,*,N) = P(*,*,N) \cdot D(*,y,N)$$

Moreover the components functions of D and F are polynmials in y of degree at most N. Let Y be the column vector

(y,y**2,...,y**N). Then we have

$$F(y,*,N) = F(N).Y = P(*,*,N).D(N).Y = P(N).D(N).Y.$$

In our notation F(N) is the matrix of coefficients of the polynomials that describe the distribution functions of each node in the heap. The element (i,j) is the coefficient of y**j in F(y,i,N). The (i,j) element of D(N) is the coefficient of y**j in Iy(i,N-i+1) (The incomplete Beta function for positive integers is a polynomial).

Since the above equation should be equal for any value of y, it implies that

$$F(N) = P(N) \cdot D(N).$$

Since D(N) is known for any N this shows how to obtain the distribution function from the frequency counts.

Conversely,

$$P(N) = F(N) \cdot D^{-1}(N)$$

The function $D^{-1}(N)$ exists and its elements d_{ij} satisfy the recursive relation:

$$d_{ij} = d^*_{ij} / \binom{N}{i};$$
$$d^*_{ij} = d^*_{i,j-1} + d^*_{i-1,j-1} \; ; \quad d^*_{11} = 1; \; d^* = 0 \text{ otherwise.}$$

This completes the equivalence between the distribution function model and frequency count model.

3.4 - Running time analysis of Algorithm F.

The worst case analysis for algorithm F occurs when we construct a heap from a heap. In this case we have at most

$$\sum_{i=1}^{N} \lceil b \rceil * n(i) = O(N*n) \text{ comparisons.}$$

This worst case, unlike algorithm W, occurs only for pathological heaps. For uniform heaps, the worst case is O(N).

The average running time analysis of a simpler version for the case b=2 can be found in Knuth (6) Ch 5.2.3. In our case we first note that for any b the resulting heaps of a random permutation of keys, are equally frequent. Knuth (6) demonstrates this theorem for binary heaps (Theorem H 5.2.3) and the extension to general heaps is immediate. This fact will ease the analysis.

The operation "find m so K(m) = max(K(x∈s(j)))" takes |s(j)|-1 comparisons. The probability of any of the sons of node j being the maximum is proportional to the weight of its subheap; i.e.,

Prob(m being the maximum son of j) = t(m)/(t(j)-1),

Thus the expected number of comparisons in a complete "sift up" operation (promoting the maximum element within a subheap all necessary levels) is

$$SFU(j) = |s(j)|-1 + \sum_{k \in s(j)} SFU(k)*t(k)/(t(j)-1)$$

if |s(j)| = 0 then SFU(j) = 0.

The second loop contains only one comparison in each step, so to count them we observe that the top element will land in any position of the subheap with equal probability (because all resulting heaps are equally frequent).

Let U(j) be the number of comparisons to promote an element from any of the leaves of node j up to node j. We can see that

$$U(j) = 1 + \sum_{k \in s(j)} U(k)*t(k)/(t(j)-1).$$

with the boundary condition U(j)=0 if |s(j)|=0. Since the element may land in any of the nodes of the subheap, (and except for node j we need one more comparison), the expected number of comparisons is

$$SFD(j) = \Big(\sum_{k \in T(j)} (U(k)+1) \Big) - 1) \; / \; t(j).$$

The total number of comparisons to construct the heap is then

$$E(Cmpr) = \sum_{k=1}^{N} SFU(k) + SFD(k).$$

For a perfect heap these formulas are readily simplified:

$$SFU(j) = n(j)*(b-1),$$

$$U(j) = n(j),$$

$$SFD(j) = (b**(n(j)+2)-b**2-b*n(j)+n(j)) \; / \; ((b**(n(j)+1)-1) * (b-1)) = A(n(j)),$$

$$E(Cmpr) = (b**(n+1) - (n+1)*b +n) \; / \; (b-1) + \sum_{k=0}^{N} A(k)*b**(n-k).$$

As in algorithm W, an excellent fit was found in relating the complexity and the estimated number of comparisons. In this case the formulas for a few values of b are,

b=2 E(Cmpr) = -0.26448 + 1.20876*H(N) + 0.20162*n,

b=3 E(Cmpr) = -0.07784 + 1.45213*H(N) + 0.14858*n,

b=4 E(Cmpr) = -0.03996 + 1.68392*H(N) + 0.12157*n,

b=5 E(Cmpr) = -0.02484 + 1.90236*H(N) + 0.10364*n.

The RMSD of each fit was 0.08, 0.06, 0.12 and 0.03 respectively.

To compare against the original algorithm, (Floyd, 2) let SF(j) be the number of comparisons needed for a "sift up" operation from node j . (The original algorithm differs from algorithm F in the "linear insertion" operation of the top element. This was done, instead, by comparing it at each step against the maximum selected).

In each iteration comparing the maximum to the top element of the subheap requires |s(j)| comparisons. If the operation is not finished, then we continue to count the comparisons in the selected subheap. Therefore the expected number of comparisons is

$$SF(j) = |s(j)| + \text{Prob(having to continue siftup)} * SF(k) * t(k) / (t(j)-1)$$

$$SF(j) = |s(j)| + \sum_{k \in s(j)} SF(k)*t(k)/t(j).$$

For perfect heaps, if h=n(j) we have,

$$SF(j) = b*(h*b**(h+2)-h*b**(h+1)-b**h+1) / ((b**(h+1)-1)*(b-1)).$$

Now the difference (term by term) between the original and the proposed algorithm is

$$SF(j)-SFU(j)-SFD(j) = b*((h*b-b-h-1)*b**h+b*h+b-h+1) / ((b**(h+1)-1)*(b-1)).$$

For b>=1 and h>=1 this is nonnegative (this can be proved by induction on h, noting that the difference is zero for h=1).

For perfect heaps the version presented is better in terms of number of comparisons. Notice that the overhead needed by the double loop is not considered.

Some least square approximations on the original method yield:

b=2 E(Cmpr) = 0.84638 + 1.37886*H(N) - 0.60211*n,

b=3 E(Cmpr) = 0.23421 + 1.55313*H(N) - 0.26230*n,

b=4 E(Cmpr) = 0.11032 + 1.75551*H(N) - 0.15961*n.

The RMSD of each fit was 0.21, 0.10 and 0.16 respectively.

In either case, this construction algorithm is, on the average, better than algorithm W in terms of comparisons. However this difference is small, so the details of implementation may be important.

3.5 - Extraction of the maximum element.

The extraction itself is trivial, the maximum element is located in R(root). Our main effort will be in reconstructing the heap with the elements left. The basic operation of algorithm F is to create a new heap from a node and the subheaps rooted at its sons. We now use this operation to form a heap from R(N) and the sons of the now empty root.

To analyze the number of operations taken in the reconstruction of the heap we should know the probability distribution of the X(i,N). These are known only when we extract the first element, but each extraction changes the probability distribution of the heap, so a complete analysis

upon succesive extractions becomes very difficult.

This process is equivalent to the body of algorithm F, changing only the top element of the subheap, that now comes from the last leaf (node N).

Under the assumption that all heaps are equally probable, we may analyze, roughly, the algorithms for reconstruction derived from algorithm F and the original one. In the first one, we find that the first loop does exactly SFU(1) comparisons. (SFU(k) is the number of comparisons needed for a complete "sift up" operation from node k during construction. Here the same arguments apply.) The second loop, for a random variable, will execute U(1) comparisons. (U(k) is the number of comparisons needed to insert a random element in any path of a subheap rooted at node k .) The variable introduced is not random, but comes from a leaf, and is smaller than at least one variable. Because of this, U(1) is an upper bound of the expected number of comparisons.

For the reconstruction algorithm based on the original algorithm we have the same situation, in this case SF(1) becomes a lower bound of the average number of comparisons. (SF(k) is the number of comparisons of a complete "sift up"

operation from node k for the original algorithm.) The quantity SF(1)-SFU(1)-U(1), which is the difference in the running time between both methods, is greater in the extraction-reconstruction phase than during the creation phase. We have already seen that this difference was always positive.

For uniform heaps, b*n is an upper bound of the average number of comparisons.

3.6 - Insertion of a new element.

This process is accomplished by the body of algorithm W.

The time required for each insertion has already been determined during the analysis of algorithm W.

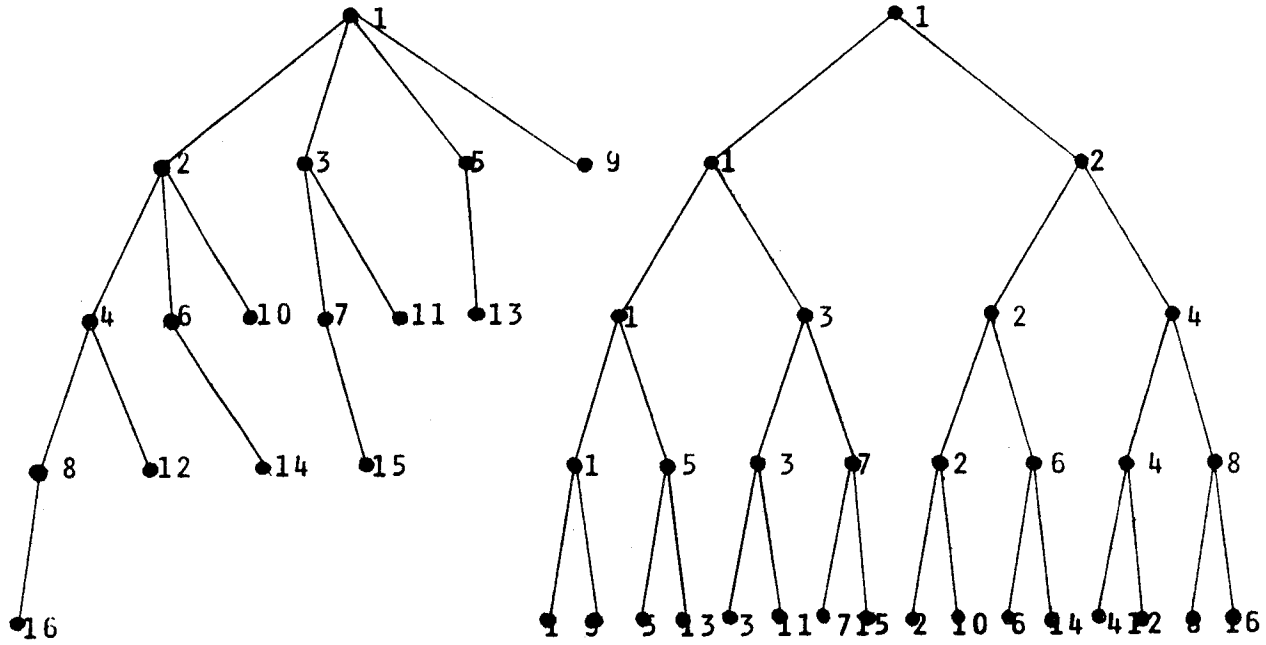
4. - Non uniform heap.

We have so far restricted ourselves to uniform heaps, but other father functions give rise to interesting non-uniform heaps.

A rather natural heap arises from

$$f(i) = i - 2^{**}\lfloor \lg(i-1) \rfloor .$$

The resulting heap resembles the tournament selection process.



heap of size 16 equivalent tournament

Figure II

If we observe the tournament, we find that node 1 played against 9,5,3 and 2 and beat them all. In the heap, node 1 is the father of nodes 9,5,3 and 2. Each node in the heap bears the same relation to each node in the tournament.

Note that f(i) in this case is not monotonic. Therefore the algorithms presented earlier cannot be applied directly. Let N be a power of 2 for simplicity. The construction of the heap in this case is easily accomplished by the repetition of the following steps: divide the elements into two equal parts; compare each element of one half with the corresponding element in the other half and interchange them so the first half element is greater than the second half; iterate with the first half.

Its main drawback is that the extraction takes (n*n+n-2)/4 comparisons on the average, which is O(lg(N)**2).

To demonstrate this result we observe that when N=2**n, the recursion formula for the number of comparisons of a heap is

$$B(2^{**}n) = n-1 + \sum_{k=1}^{n-1} B(2^{**}k) * 2^{**}(k-n),$$

with B(1)=B(2)=0.

The above follows, in the usual way, from the description of the algorithm.

5. - Conclusions.

The above algorithms are not intended to demonstrate the optimum coding techniques. Instead they are written in such a way that they are easy to translate to any other language in which they are going to be implemented. In many cases, without modifying the behaviour of the algorithm, the language may provide some shortcuts for the implementation.

The class of data structures we have defined is considerably broader than those usually considered under the name heap. The results apply to this class of data structure and we suggest that in some applications it may be worth considering heaps other than binary heaps.

Our main concern in the running time evaluation is with the number of comparisons. In terms of actual implementations the number of comparisons may account for only a small percentage of the total running time, but in most cases a comparison or a number of grouped comparisons is directly related to the number of times a given loop is repeated. This leads us to the important conclusion that, in these cases, the total running time is some linear function of the number of comparisons.

The numerical approximations show that the running time requirements of the algorithms are closely approximated by a linear function of the complexity. This is an useful tool in comparing algorithms or evaluating them absolutely.

In the authors' experience, the computation of the probability distribution functions as polynomials is a powerful tool in solving these kinds of problems. We feel that this technique may be usefully applied to the average case analysis of other algorithms.

The results derived for Algorithm W make use of the U(0,1) distribution of the keys in the heap. Similar results should be obtained for any other distributions used, as long as the variables are independent. That is, that any particular ordering is equally probable.) It may be possible to find a distribution function that simplifies the calculation and avoids the need for polynomials.

6. - References.

(1) Aho A.V., Hopcroft J.E., and Ullman J.D. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

(2) Floyd R.W. "Algorithm 245", CACM 7, 1964, pp. 701.

(3) Friend E.H. "Sorting", JACM Vol. 3, (1956), pp. 152-154.

(4) Gentleman W.M. "Row Elimination for Solving Sparse Linear Systems and Least Squares Problems", Dundee Biennial Conference on Numerical Analysis, July 1975,

(5) Johnson S.C. "Sparse Polynomial Arithmetic", Bell Telephone Labs. Murray Hill N.J.

(6) Knuth D.E. The Art of Computer Programming, Vol. 3, Ch 5, Addison-Wesley, 1973.

(7) Malcolm M.A. Simpson R.B. "Local Versus Global Strategies for Adaptive Quadrature", ACM TOMS Vol. 1, Number 2 (June 1975).

(8) Williams J.W.J. "Algorithm 232", CACM 7, 1964, pp. 347-348.

(9) Wirth N. "The Programming Language PASCAL" Acta Informatica Vol. 1, pp. 35-63.

7. - Appendix I - Numerical results.

Table of expected value of comparisons for various sizes, comparing algorithms W, F and the original (Floyd, 2), for b=2, 3 and 3/2.

	Algorithm W			Algorithm F			Original Algorithm		
Size	b=2	b=3	b=3/2	b=2	b=3	b=3/2	b=2	b=3	b=3/2
10	13.95	11.75	14.54	12.17	11.07	12.73	12.93	11.40	13.50
20	33.55	27.91	39.53	27.83	24.32	33.39	30.23	21.51	36.55
30	53.48	43.97	65.84	43.33	37.41	54.59	47.43	39.37	60.77
40	75.27	59.72	93.32	59.86	50.35	76.95	66.14	52.93	86.04
50	96.61	77.82	122.24	76.08	64.87	97.10	84.41	68.69	110.56
60	117.69	94.72	150.13	92.08	78.71	119.26	102.33	83.40	136.48
70	139.91	111.62	180.61	108.50	92.99	141.59	121.06	98.48	162.69
80	161.91	128.88	210.00	124.84	106.27	163.34	139.56	112.82	188.17
90	183.93	145.36	238.51	141.30	119.36	185.53	158.22	126.69	214.28
100	206.02	162.28	270.27	157.71	133.42	207.58	176.82	141.62	241.37
110	227.63	179.16	301.06	173.71	146.33	229.23	194.94	155.42	267.08
120	249.63	195.44	331.22	190.13	159.17	253.26	213.47	168.97	294.17