JOURNAL OF ALGORITHMS **10**, 352–365 (1989)

Building Heaps Fast

C. J. H. McDIARMID*

Institute of Economics and Statistics, Oxford University, Oxford, England

AND

B. A. REED

Bell Communication Research, 435 South Street, Morristown, New Jersey 07960

Received May 26, 1987; accepted September 27, 1988

We present an algorithm to construct a heap which uses on average $(\alpha + o(1))n$ comparisons to build a heap on n elements, where $\alpha \approx 1.52$. Indeed on the overwhelming proportion of inputs our algorithm uses this many comparisons. This average complexity is better than that known for any other algorithm. We conjecture that it is optimal. Our method is a natural variant of the standard heap construction method due to Floyd. © 1989 Academic Press, Inc.

1. INTRODUCTION

Heaps [W, G, Fl] are the most studied type of priority queue. This is because they are simple, efficient, and elegant. A heap (min-heap) is a binary tree on a totally ordered set such that each node is greater than (less than) its children. Furthermore, all the leaves in a heap are on at most two adjacent levels and the leaves on the bottom level are as far to the left as possible. Heaps were originally developed by Williams [W] to be used in an in-place sorting algorithm which runs in $O(n \log n)$ time. Since that time, they have been used extensively both in practice and in the development of theoretically efficient algorithms.

The standard heap construction algorithm is due to Floyd. His algorithm uses at most $[2 + o(1)]n$ comparisons to build a heap on n elements and about $1.88n$ comparison on average [K]. We present a natural variant of

*This research was performed under a consulting agreement with Bell Communications Research.

Floyd's algorithm which uses, on average, about $1.52n$ comparisons to build a heap on n elements. Indeed on the overwhelming proportion of inputs our algorithm uses close to this number of comparisons. It has the same worst-case behavior as Floyd's algorithm.

Previously, the fastest algorithm for building heaps was due to Gonnet and Munro [GM1]. This algorithm takes $(1.625 + o(1))n$ comparisons in the worst case and can be modified slightly so that it runs in $(1\frac{65}{112} + o(1))n \approx 1.5803n$ comparisons on average ([GM2]; this corrects a figure in [GM1]). Their algorithm first builds a binomial queue (another type of priority queue: see [V]) and then converts it into a heap.

One particularly appealing property of heaps is that a heap of size n can be implicitly stored in the first n cells of an array. In an array representation of a heap, the father of the element with index j has index $\lfloor j/2 \rfloor$. In this representation, the next available leaf position corresponds to the $(n + 1)$th cell in the array. In discussing algorithms in this paper, we are assuming that they are to be implemented by an array. However, we shall not discuss the details of this implementation; instead we shall outline our algorithms informally.

2. Two Old Algorithms

The first heap construction algorithm was proposed by Williams ([W], see also, for example, [AHU]). This algorithm builds a heap by sequentially inserting elements into an initially empty heap. An element x is added to a heap by placing it in the first available leaf and then bubbling it up until it is smaller than its father. In the worst case, x bubbles up to the root of the heap and the insertion algorithm requires k comparisons where k is the depth of the new heap (the depth of a node in a heap is the number of edges in the path from the node to the root; the depth of a heap is the depth of a leaf on the bottom level of the heap). It follows easily that Williams' algorithm takes $n \log n + O(n)$ comparisons in the worst case to build a heap on n elements. The expected number of comparisons is between about $1.75n$ and $2.76n$ for sufficiently large n (see [Fr, BS]).

Floyd ([Fl], see also [AHU, K]) proposed an algorithm which makes $(2 + o(1))n$ comparisons in the worst case. It uses smaller heaps as building blocks for larger ones. For example, we build a perfect heap of depth k from two of depth $k - 1$ and a new element x as follows. First we form a binary tree with root x and left and right subtrees the two heaps. If x were larger than both its sons, the tree would be a heap. If not, we simply swap the positions of x and its larger son. After repeating this step at most k times, we obtain a heap. This "trickle-down" procedure is the core of Floyd's algorithm.

3. A New Algorithm

Our algorithm combines the ideas of Floyd and Williams. As in Floyd's algorithm, we construct a heap from two smaller heaps and an extra element, x. However, instead of putting x at the root and trickling it down, we trickle down an empty position to the bottom of the heap and then put x in this position and bubble it up. Thus, the core of our algorithm is the following procedure (see Fig. 1).

```
Merge (H: a tree which satisfies all the heap conditions except that the key $x$ at the
      root may fail to be larger than the keys of its children);
Begin {trickle-down phase}
      While $x$ is not at a leaf do: compare the keys of the two children of $x$; swap
      $x$ with the larger of these keys; end while
      {bubble-up phase}
      while $x$ is not at the root and it is larger than the key of its father do: swap $x$
      with the key of its father; end while
End
```

We call the path formed by the larger children in the trickle-down phase, the trickle-down path.

Note that out method uses five comparisons on this example while Floyd's method would require six. Since most of the elements of a heap are at the bottom, we expect that, in general, the new element will end up there. Thus, our algorithm should be faster, on average, than Floyd's.

In discussing this algorithm further, we will restrict our attention to merging perfect heaps (heaps in which all the leaves are at the same depth). Note that for any heap on n elements all but at most $\log n$ of the merges

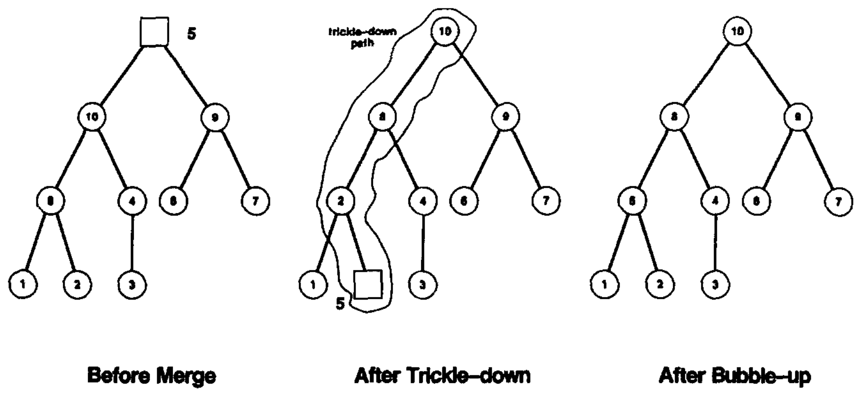

Fig. 1.

are of this type. Furthermore, each of these merges takes at most $2 \log n$ comparisons. This $O(\log^2 n)$ term can be ignored.

Let $H(k)$ be the expected time (number of comparisons made) taken to construct a k-heap (a perfect heap of depth k) using our technique, and let $M(k)$ be the expected time taken to merge two $(k-1)$-heaps into a k-heap. Clearly $H(0) = 0$ and for $k \geq 1$,

$$H(k) = 2H(k-1) + M(k). \tag{1}$$

Let $\overline{H}(k)$ be the expected time per element to construct a k-heap. That is $\overline{H}(k) = H(k)/(2^{k-1} - 1)$. By a k-merge, we mean an application of our algorithm in which we merge two $(k-1)$-heaps into a k-heap. Obviously, the trickle down phase of a k-merge takes k comparisons. The time taken by the bubble-up phase depends on the height of (the node with key) x in the new heap (the height of a node is the depth of the subheap rooted at that node). Clearly, x is at the root of the new heap if and only if it is the maximum of all the $2^{k+1} - 1$ elements in the created heap. Thus, the probability that x is at height k in the new heap is $1/(2^{k+1} - 1)$. In fact, as we show in the next section, for each $i = 0, 1, \ldots, k$ the probability that x has height i, given that it has height at most i is $1/(2^{i+1} - 1)$. Thus for each such i,

$$\begin{aligned}\text{Prob}\{x \text{ has height } i\} &= \left(\prod_{j=i+1}^{k} \frac{(2^{j+1} - 1) - 1}{2^{j+1} - 1}\right) \cdot \frac{1}{2^{i+1} - 1} \\ &= \frac{2^{k-i}}{2^{k+1} - 1}.\end{aligned}$$

Clearly, if x is at height $i \leq k - 1$, then the bubble-up phase required $i + 1$ comparisons. If x has height k, then we required k comparisons. Thus, on average, the bubble-up phase takes

$$1 + \left(\sum_{i=0}^{k-1} \frac{(2^{k-i})i}{2^{k+1} - 1}\right) + \frac{k-1}{2^{k+1} - 1} = 2 - \frac{k+2}{2^{k+1} - 1} \text{ comparisons.}$$

Now, $M(k) = (k + 2) - (k + 2)/(2^{k+1} - 1)$. Approximating recurrence (1), we find that $\overline{H}(k) \to \overline{\alpha}$ as $k \to \infty$, where $\overline{\alpha} \approx 1.649271$. (We note that Carlsson [C] has independently investigated this algorithm.)

However, the algorithm may make unnecessary comparisons. For example, if the roots of both $(k-1)$-heaps were bubbled up to the top in the last phase and we remember comparisons made then after only one further comparison, we will be able to compute the whole trickle-down path. Thus, by using sibling orders known from previous merges, we may be able to reduce the number of comparisons we make.

In the next section we show that the expected number of comparisons we save during a k-merge by doing this is $1 - k/(2^k - 1)$. Thus, the trickle-down phase of a k-merge takes $k - 1 + k/(2^k - 1)$ comparisons on average and now $M(k) = k + 1 + k/(2^k - 1) - (k + 2)/(2^{k+1} - 1)$. When we approximate recurrence (1), we find that $\overline{H}(k) \to \alpha$ as $k \to \infty$, where $\alpha \approx 1.521288$.

It turns out that all "extra" comparisons known are sibling comparisons, which can be recorded with one extra bit per node. Indeed we can make the expected number of comparisons per element arbitrarily close to α while using only a (suitably large) constant amount of extra storage.

Floyd's method can be treated in a similar though easier way. For the basic method, when we do not record sibling comparisons, the expected number of comparisons to form a perfect heap on n elements is $(\overline{\beta} + o(1))n$, where $\overline{\beta} \approx 1.881373$ (see [K]). When we do record all comparisons, $\overline{\beta}$ drops to $\beta \approx 1.791415$.

4. Analyzing Our Algorithm: Average Complexity

In this section we analyze the average time complexity of our algorithm. We begin with some definitions and basic observations. We build a k-heap on the set $X = \{x_1, x_2, \ldots, x_{2^{k+1}-1}\}$ as follows:

(i) Insert the elements of X into a complete binary tree of depth k. We shall think of the nodes of the tree as being labeled by elements of X. Initially, x_1 labels the root and for $i = 1, \ldots, 2^k - 1$, the node labeled by x_i has children labeled by x_{2i} and x_{2i+1}.

(ii) For $j = 1$ to k do: For each node y at height j, create a heap rooted at y by running our merge algorithm on the tree rooted at y.

An input to the algorithm is a permutation (i.e., linear order) π of X. We assume that all such inputs are equally likely. We are interested in counting the expected number of comparisons needed to construct a heap using our algorithm. During the execution of the algorithm, we gain some partial order information on π. Some of this information is recorded in the heap structure we create. We will also record this information by assigning each node one of three colors, red, blue, and green. Initially all leaves are colored red. During the trickle-down phase of a merge, we color all the nodes on the trickle-down path blue. Then, during the bubble-up phase, we change to green the color of all the nodes on the trickle-down path past which x bubbles up; and we color red the node finally labeled x. Note that a green node is known to be greater than its sibling. By a colored heap, we mean a heap labeled by elements of X with each node colored red, blue, or green. We now make two observations.

Observation 1. At any stage during the execution of the algorithm we have constructed a family of heaps. Each of these colored heaps corresponds to a partial order on its labels. It is easy to see that these partial orders capture all the information we have yet discovered about the input linear order π on X. Thus, the inputs which lead to the construction of this family are precisely those which extend the corresponding partial order.

Observation 2. Consider a partially ordered set $(X, <_0)$. Let C be a subset of X such that for all x, x' in C and y in $X - C$, $x <_0 y$ (resp. $x >_0 y$) if and only if $x' <_0 y$ (resp. $x' >_0 y$) (we shall call such a set a *faction*) Let Π be any set of linear extensions of $(C, <_1)$, where $<_1$ is the restriction $<_0$ to C. Then, the proportion of linear extensions of $(X, <_0)$ whose restriction to C lies in Π equals the proportion of linear extensions of $(C, <_1)$ which are in Π.

Observation 2 allows us to prove that the probability that we bubble-up to height i during a k-merge is $2^{k-i}/(2^{k+1} - 1)$ as claimed in Section 3.

LEMMA 3. *The probability that we bubble-up to height i in a k-merge given that we bubble-up no higher is* $1/(2^{i+1} - 1)$ (*for* $i = 0, 1, \dots, k$).

Proof. We can think up performing the bubble-up from the top down, for this does not change the final position of x. Let y and z be the elements on the trickle-down path at heights i and $i + 1$, respectively. To prove Lemma 3 we need only note that if x loses to z then x along with the subheap rooted at y is a faction (since y is blue; $x < z$; $y < z$). □

As we showed in Section 3, Lemma 3 implies that the bubble-up phase of a k-merge takes $2 - (k + 2)/(2^{k-1} - 1)$ comparisons on average. This completes the analysis of the simplified algorithm. We turn now to an analysis of the trickle-down phase.

For a node v at height i, let $a_j(v)$ be the ancestor of v at height j. By the *ancestor sequence* of v we mean the sequence of colors $(c_i, \dots, c_k)$, where c_j is the color of $a_j(v)$. An ancestor sequence is *valid* if every green node is followed by a red or green node. We consider only valid sequences. We shall use a_j for $a_j(v)$ when no confusion can arise.

LEMMA 4. *The probability that a node at height i in a k-heap has ancestor sequence $(c_i, \dots, c_k)$ is $\prod_{j=i}^{k} p_j$, where*

$$p_j = \begin{cases} \dfrac{1}{2^{j+1}-1} & \text{if } c_j \text{ is red and } c_{j+1} \text{ is blue or } j = k \\ \dfrac{2^{j+1}-2}{2^{j+1}-1} & \text{if } c_j \text{ is blue and } c_{j+1} \text{ is blue or } j = k \\ \dfrac{1}{2} & \text{if } c_j \text{ is green } (\text{this implies } c_{j+1} \text{ is red or green}) \\ \dfrac{1}{2}\left(\dfrac{1}{2^{j+1}-1}\right) & \text{if } c_j \text{ is red and } c_{j+1} \text{ is red or green} \\ \dfrac{1}{2}\left(\dfrac{2^{j+1}-2}{2^{j+1}-1}\right) & \text{if } c_j \text{ is blue and } c_{j+1} \text{ is red or green.} \end{cases}$$

Proof. We need only prove the lemma for leaves. The general case follows by a simple summation. We note that the theorem is true for $k = 0$ and proceed by induction on k. We shall consider a k-merge, assume the formula was correct for the two $(k-1)$-heaps and prove that it is correct for the k-heap.

Let S_v^{k-1} and S_v^k denote a leaf v's ancestor sequences in the $k-1$ and k-heaps, respectively. Consider the following generic form for a leaf's ancestor sequence in the k-heap:

$A = (c_{j+1}, \dots, c_k)$ is blue.
c_j is red.
$B = (c_{l+1}, \dots, c_{j-1})$ is green.
c_l is not green.
$C = (c_0, \dots, c_l)$ is an arbitrary valid sequence.
$(A, B, C$ may be empty.)

We call this sequence $S(j, C)$.

We want to show that

$$\begin{aligned} \text{Prob}\{S_v^k = S(j,C)\} &= \left(\prod_{m=j+1}^{k} \frac{2^{m+1}-2}{2^{m+1}-1}\right) \cdot \frac{1}{2^{j+1}-1} \\ &\quad \cdot \left(\frac{1}{2}\right)^{j-l-1} \cdot \left(\frac{1}{2} \cdot P(C)\right) \\ &= 2^{(k-j)} \cdot \frac{1}{2^{k+1}-1} \cdot \left(\frac{1}{2}\right)^{j-l-1} \cdot \left(\frac{1}{2} \cdot P(C)\right), \end{aligned}$$

where $P(C)$ is our computed probability for the sequence $\{c_0, \dots, c_l\}$ as an ancestor sequence in a l-heap (set $\frac{1}{2} \cdot P(C) = 1$ and $l = -1$ if C is empty). Now, $S(j, C)$ can arise from any of the following types of old ancestry sequences and only these types:

(i) $A' = (c_{p+1}, \dots, c_{k-1})$ is blue, c_p is red, $B' = (c_{l+1}, \dots, c_{p-1})$ is green, $C' = C$. (Call this $S'(p, C)$.)

(ii) $A' = (c_{l+1}, \dots, c_{k-1})$ is blue, $C' = C$. (Call this $S''(C)$.)

(iii) $A' = (c_{p+1}, \dots, c_{k-1})$ is blue, c_p is red, $A'' = (c_{q+1}, \dots, c_{p-1})$ is green, $A''' = (c_{j+1}, \dots, c_q)$ is blue, c_j is red, $B' = B$, $C' = C$. (Call this $S'''(p, j, C)$.)

We consider each of these three cases in turn. First however, we make two observations.

Observation 5. Let v be a leaf of one of the two $(k-1)$-heaps with ancestor sequence $(c_0, \dots, c_{k-1})$, where c_t is red and c_i is blue for each $i > t$. Then, for each such i the probability that the trickle-down path of the k-merge passes through a_j is $(\frac{1}{2})^{k-i}$.

Proof. If we fix a colored heap where v has an appropriate ancestor sequence, the result follows from Observation 2. By summing over all such colored heaps we obtain the desired result.

Observation 6. Whatever the configuration following the trickle-down phase, the probability that x bubbles up to height s is

$$\frac{2^{k-s}}{2^{k+1}-1}.$$

Proof. See Lemma 3 and its proof.

Case 1. v has an ancestor sequence of type (i) in a $(k-1)$-heap. In this case, if $p \neq j$ then x must trickle-down through a_p and then bubble-up to a_j. If $p = j$, then the trickle-down path does not necessarily pass through a_p.

Case 1.1. a_p is on the trickle-down path. Fix p and C. By the induction hypothesis:

$$\operatorname{Prob}\{S_v^{k-1} = S'(p, C)\}$$
$$= \frac{1}{2^k - 1} \cdot 2^{k-p-1} \cdot \left(\frac{1}{2}\right)^{p-l} \cdot P(C).$$

By Observation 5, given that $S_v^{k-1} = S'(p, C)$, the probability that a_p is on the trickle-down path is $(\frac{1}{2})^{k-p}$. By Observation 6, the probability that x

bubbles up to a_j given that $S_v^{k-1} = S'(p, C)$ and a_p is on the trickle-down path is $2^{k-j}/(2^{k+1} - 1)$. Summing over all p we obtain

$$\begin{aligned}P_1 &= \operatorname{Prob}\left\{ S_v^{k-1} \text{ is of type } (i) \text{ and } a_p \text{ is on the trickle-down path}\right\}\\ &= \sum_{p=l+1}^{k+1} \frac{1}{2^k - 1} \cdot 2^{k-p-1} \cdot \left(\frac{1}{2}\right)^{p-l} \cdot P(C) \cdot \left(\frac{1}{2}\right)^{k-p} \cdot \frac{2^{k-j}}{2^{k+1} - 1}\\ &= \frac{P(C)}{2^{k+1} - 1} \cdot \frac{2^{k-j}}{2^k - 1}\left(\frac{1}{2} - 2^{i-k}\right).\end{aligned}$$

Case 1.2. a_p is not on the trickle-down path. In this case, $p = j$ and the trickle-down path branches off from v's ancestors at some point above a_p. Now,

$$\operatorname{Prob}\left\{ S_v^{k-1} = S'(j, C)\right\} = \frac{1}{2^k - 1} \cdot 2^{k-j-1} \cdot \left(\frac{1}{2}\right)^{j-l} \cdot P(C).$$

Given that $S_v^{k-1} = S'(j, C)$, the probability that a_{t+1} is on the trickle-down path but a_t is not is $(\frac{1}{2})^{k-t}$ for $t \geq j$ (see Observation 5). Given that both these events occur, the probability x does not bubble-up to a_{t+1} or above is $(2^{k+1} - 2^{k-t})/(2^{k+1} - 1)$ (see Observation 6). Thus,

$$\begin{aligned}P_2 &= \operatorname{Prob}\Big\{ S_v^{k-1} = S'(j, C), S_v^k = S(j, C),\\ &\qquad\qquad a_j \text{ is not on trickle down path}\Big\}\\ &= \sum_{t=j}^{k-1} \left(\frac{1}{2}\right)^{k-t} \cdot \frac{2^{k+1} - 2^{k-t}}{2^{k+1} - 1} \cdot \frac{1}{2^k - 1} \cdot 2^{k-j-1} \cdot \left(\frac{1}{2}\right)^{j-l} \cdot P(C)\\ &= \frac{P(C)}{2^{k+1} - 1} \cdot \frac{1}{2^k - 1} \cdot \left(2^{2k-2j+l} - 2^{k-j+l} - 2^{k-2j+l-1}(k - j)\right).\end{aligned}$$

Case 2. v has an ancestor sequence of type (ii) in the $(k - 1)$-heap. In this case, C cannot be empty. Note that, by the induction hypothesis:

$$\operatorname{Prob}\left\{ S_v^{k-1} = S''(C)\right\} = P(C) \cdot 2^{k-l-1} \cdot \frac{2^{l+1} - 1}{2^k - 1}.$$

Given this, the probability that a_{l+1} is on the trickle-down path and a_l is not is $(\frac{1}{2})^{k-l}$. Given all this, the probability that x bubbles up to a_j is

$2^{k-j}/(2^{k+1}-1)$ (by Observation 6). Thus,

$$\begin{aligned} Q &= \operatorname{Prob}\{S_v^{k-1} = S''(C), S_v^k = S(j,C)\} \\ &= P(C)\cdot 2^{k-l-1}\cdot \frac{2^{l+1}-1}{2^k-1}\cdot\left(\frac{1}{2}\right)^{k-l}\cdot\frac{2^{k-j}}{2^{k+1}-1}. \end{aligned}$$

Case 3. v has an ancestor sequence of type (iii) in the $(k-1)$-heap. In this case, for fixed p, q, and C,

$$\begin{aligned} &\operatorname{Prob}\{S_v^{k-1} = S'''(p,q,C)\} \\ &= P(C)\cdot\left(\frac{1}{2}\right)^{j-l}\cdot\frac{1}{2^{j+1}-1}\cdot\prod_{m=j+1}^{q}\frac{2^{m+1}-2}{2^{m+1}-1}\cdot\left(\frac{1}{2}\right)^{p-q} \\ &\quad\cdot\frac{1}{2^{p+1}-1}\cdot\prod_{t=p+1}^{k-1}\frac{2^{t+1}-2}{2^{t+1}-1} \\ &= P(C)\cdot\left(\frac{1}{2}\right)^{j-l}\cdot\frac{1}{2^{q+1}-1}\cdot 2^{q-j}\cdot 2^{q-p}\cdot\frac{1}{2^k-1}\cdot 2^{k-p-1}. \end{aligned}$$

Given that $S_v^{k-1} = S'''(p,q,C)$, for S_v^k to be $S(j,C)$, a_p must be on the trickle-down path and we must not bubble up to a_{q+1} or above. This occurs with probability

$$\left(\frac{1}{2}\right)^{k-p}\cdot\frac{2^{k+1}-2^{k-q}}{2^{k+1}-1}.$$

Summing over all appropriate choices of p and q we obtain

$$\begin{aligned} R &= \operatorname{Prob}\{S_v^{k-1} \text{ is of type (iii) and } S_v^k = S'(j,C)\} \\ &= \sum_{p=j+1}^{k-1}\sum_{q=j}^{p-1}\left(\frac{1}{2}\right)^{k-p}\cdot\frac{2^{k+1}-2^{k-q}}{2^{k+1}-1}\cdot P(C)\cdot\left(\frac{1}{2}\right)^{j-l}\cdot\frac{1}{2^{q+1}-1} \\ &\quad\cdot 2^{q-j}\cdot 2^{q-p}\cdot\frac{1}{2^k-1}\cdot 2^{k-p-1} \\ &= \frac{P(C)}{2^{k+1}-1}\cdot\frac{1}{2^k-1}\cdot\left(2^{k+l-2j-1}(k-j-2)+2^{l-j}\right). \end{aligned}$$

Finally, we put all three cases together. We find

$$\operatorname{Prob}\{S_v^k = S'(j,C)\} = P_1 + P_2 + Q + R.$$

It is a routine but tedious matter to verify that this gives

$$\operatorname{Prob}\{S_v^k = S'(j,C)\} = \frac{P(C)}{2^{k+1}-1}\cdot 2^{k-j}\left(\frac{1}{2}\right)^{j-l}\qquad\text{as required. } \square$$

The number of comparisons made during the trickle-down phase of our algorithm depends on the height of the first (in fact: only) red node encountered on the trickle-down path. The above lemma allows us to calculate the expected height of this node.

LEMMA 7. *The probability that the red node on the trickle-down path of a k-merge is at height i is* $2^{k-i-1}/(2^k - 1)$.

Proof. Let us call a red node *exposed* if all its ancestors are blue. Let v be a node of height i in the original $(k-1)$-heap. By Lemma 4, the probability that v is a red exposed node is $2^{k-i-1}/(2^k - 1)$. By Observation 5, the probability v is on the trickle-down path given that it is an exposed red node is $1/2^{k-i}$. Since exactly one of the 2^{k-i} nodes at height i is on the trickle-down path, we obtain

$$\text{Prob}\{\text{the red node on the trickle-down path has height } i\}$$
$$= 2^{k-i} \cdot \frac{1}{2^{k-i}} \cdot \frac{2^{k-i-1}}{2^k - 1} = \frac{2^{k-i-1}}{2^k - 1}. \qquad \square$$

LEMMA 8. *The expected number of comparisons during the trickle-down phase of a k-merge is* $k - 1 + k/(2^k - 1)$.

Proof. The number of comparisons is just $\{k$-the height of the first red node on the trickle-down path$\}$. By Lemma 7 we see that the expected number of comparisons during the trickle-down phase of a k-merge is

$$\sum_{i=0}^{k-1} \frac{(k-i)2^{k-i-1}}{2^k - 1} = k - 1 + \frac{k}{2^k - 1}. \qquad \square$$

5. ANALYZING OUR ALGORITHM: DEVIATIONS FROM THE AVERAGE

In this section, we show that for the overwhelming proportion of inputs, the number of comparisons our algorithm uses is close to the average number. A similar result holds for all other variants of Floyd's algorithm. (See also [D].) More precisely, let H_k be the random number of comparisons used to build a perfect heap of depth k (with $n = 2^{k+1} - 1$ elements). Then we have,

THEOREM. *For any $t > 0$,*

$$\text{Prob}\{|H_k - E[H_k]| \geq t\} \leq 2\exp\{-t^2/34 \cdot 2^{k+1}\}.$$

Now let B_n denote the random number of comparisons used to build a heap on n elements. As before, forming imperfect subheaps leads to an $O(\log^2 n)$ correction term, which will be swallowed up in larger terms.

COROLLARY. *For any $\varepsilon > 0$, if n is sufficiently large*

$$\operatorname{Prob}\left\{|B_n - E(B_n)| > \varepsilon n\right\} < 2\exp\left(-(\varepsilon^2/35)n\right).$$

COROLLARY. *For any $\varepsilon > 0$ there is a constant c such that for all n,*

$$\operatorname{Prob}\left\{|B_n - E(B_n)| > cn^{1/2}\right\} < \varepsilon.$$

Our analysis works because the numbers of comparisons used in the different merges are "nearly independent." We may thus use the powerful martingale inequality described below.

Let $X_1, X_2, \ldots, X_t$ be random variables, and for each $i = 1, \ldots, t$ let $\mathbf{X}^{(i)}$ denote $(X_1, \ldots, X_i)$. Suppose that the random variable M is determined by $\mathbf{X}^{(t)}$, so that $E(M|\mathbf{X}^{(t)}) = M$. For each $i = 1, \ldots, t$ let

$$d_i = \left\| E(M|\mathbf{X}^{(i-1)}) - E(M|\mathbf{X}^{(i)}) \right\|.$$

Here $E(M|\mathbf{X}^{(0)})$ means just $E(M)$, and $\|\cdot\|$ is the (essential) supremum norm. Thus d_i bounds the change in the conditional expectation of M given one further piece of information, namely the value of X_i.

LEMMA. *For any $t > 0$,*

$$\operatorname{Prob}\left\{|M - E(M)| \geq t\right\} \leq 2\exp\left(-t^2 \Big/ \sum_i d_i^2\right).$$

This lemma is a special case of a martingale inequality due to Azuma (see [S]).

Proof of Theorem. Consider the ith call of merge: let X_i be the sequence of comparisons made, let N_i be the number of comparisons the naive algorithm would make, let M_i be the number of actual comparisons made, and let $F_i = N_i - M_i$ be the number of old (free) comparisons used. Further let $\mathbf{X}^{(i)}$ denote $(X_1, \ldots, X_i)$, and let $N = \Sigma_i N_i$, $M = \Sigma_i M_i$, $F = \Sigma_i F_i$.

A key observation (see Observation 6 above) is that for each $i > 1$, N_i is independent of the entire history $\mathbf{X}^{(i-1)}$. Thus

$$E(N|\mathbf{X}^{(i-1)}) - E(N|\mathbf{X}^{(i)}) = E(N_i) - N_i.$$

Suppose that the ith merge is an h-merge. Then $h + 1 \leq N_i \leq 2h$, and $h + 1 \leq E[N_i] < h + 2$. So

$$\left\| E(N|\mathbf{X}^{(i-1)}) - E(N|\mathbf{X}^{(i)}) \right\| = \left\| E(N_i) - N_i \right\| < h - 1.$$

Next we consider the free comparisons. Note that the random number of times that the trickle-down path visits the current subheap during later merges is independent of $\mathbf{X}^{(i)}$; and it has an expected value $< \sum_{j \geq 1} (\frac{1}{2})^j = 1$. So, for any possible value $\mathbf{x}^{(i-1)}$ of $\mathbf{X}^{(i-1)}$ and any two possible values x_i, x_i' of X_i,

$$\left| E\left[\sum_{j>i} F_j | \mathbf{X}^{(i-1)} + \mathbf{x}^{(i-1)}, X_i = x_i \right] - E\left[\sum_{j>i} F_j | \mathbf{X}^{(i-1)} + \mathbf{x}^{(i-1)}, X_i = x_i' \right] \right| < h.$$

So

$$\left\| E\left[\sum_{j>i} F_j | \mathbf{X}^{(i-1)} \right] - E\left[\sum_{j>i} F_j | \mathbf{X}^{(i)} \right] \right\| < h.$$

Also, of course,

$$\left\| E[F_i | \mathbf{X}^{(i-1)}] - F_i \right\| < h - 1.$$

Hence

$$\left\| E(F|\mathbf{X}^{(i-1)}) - E(F|\mathbf{X}^{(i)}) \right\| < 2h - 1.$$

The above results for N and F show that

$$\left\| E(M|\mathbf{X}^{(i-1)}) - E(M|\mathbf{X}^{(i)}) \right\| < 3h - 2.$$

Now let us suppose that the ith merge is an h_i-merge. Then

$$\begin{aligned} D &= \Sigma_i \left\| E(M|\mathbf{X}^{(i-1)}) - E(M|\mathbf{X}^{(i)}) \right\|^2 \\ &< \Sigma_i (3h_i - 2)^2 \\ &= \sum_{h=1}^{k} 2^{k-h} (3h - 2)^2 \\ &< 34 \cdot 2^k. \end{aligned}$$

The theorem now follows from the last lemma. □

6. Two Questions

As noted in the abstract, we believe that our algorithm is optimal with respect to the expected number of comparisons. Our sole reason for believing this is its simplicity. The best lower bound known on the expected number of comparisons is the straightforward information theory bound which is asymptotically about $1.3644n$ (see [GM1]). This is also the best known lower bound on the number of comparisons used in the worst case.

Question 1. Can you raise this lower bound for either the average or worst case?

The main cost of heapsort arises from a sequence of Floyd's trickle-down operations. It seems likely that using our method would reduce the overall cost of heapsort on average. For a discussion see [C].

Question 2. What is the average-time behavior of this modified heapsort?

Note that in the selection stage of heapsort we could just trickle-down and not bubble-up. The total number of comparisons needed for any input is then $n \log n + O(n)$ (which is asymptotically optimal) though we require extra storage.

References

[AHU] A. V. Aho, J. E. Hopcroft, and J. Ullman, "Data Structures and Algorithms," Addison–Wesley, Reading, MA, 1983.

[BS] B. Bollobas and I. Simon, Repeated random insertion into a priority queue structure, *J. Algorithms* **6** (1985), 466–477.

[C] S. Carlsson, Heaps, department of Computer Science, Lund University, Sweden, 1987.

[D] E. E. Doberkat, An average case analysis of Floyd's algorithm to construct heaps, *Inform. and Control* **61** (1984), 114–131.

[Fl] R. W. Floyd, Algorithm 245: TREESORT, *Comm. ACM* **7** No. 12 (1964), 701.

[Fr] A. M. Frieze, "On the Construction of Random Heaps," Queen Mary College, London, 1986.

[G] G. H. Gonnet, "Handbook of Algorithms and Data Structures," Addison–Wesley, Reading, MA, 1984.

[GM1] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM J. Comput.* **15** (1986), 964–971.

[GM2] G. H. Gonnet and J. I. Munro, private communication.

[K] D. E. Knuth, "The Art of Computer Programming, Vol. III. Sorting and Searching," Addison–Wesley, Reading, MA, 1973.

[S] W. F. Stout, "Almost Sure Convergence," Academic Press, New York/London, 1974.

[V] J. A. Vullemin, A data structure for manipulating priority queues, *Comm. ACM* **22** (1978), 309–314.

[W] J. W. J. Williams, Algorithm 232: HEAPSORT, *Comm. ACM* **7** (1964), 347–348.