THE BEST CASE OF HEAPSORT

Robert Sedgewick
Russel Schaffer

CS-TR-293-90

November 1990

The Best Case of Heapsort

Robert Sedgewick* and Russel Schaffer**
Department of Computer Science
Princeton University

Abstract: Heapsort is a fundamental algorithm whose precise performance characteristics are little understood. It is easy to show that the number of keys moved during the algorithm is $N \lg N + O(N)$ in the worst case, and it is conjectured that the average case performance is the same, though that seems very difficult to prove. In this paper, we show by construction that the *best case* for the number of moves is $\sim \frac{1}{2} N \lg N$. This destroys the conjecture that Heapsort is asymptotically flat (a possible easy road to the average-case asymptotic analysis), and also implies that a variant of Heapsort suggested by Floyd is not asymptotically optimal in the worst case.

The construction was suggested by results of an empirical study that involved generating and analyzing hundreds of billions of heaps. Other facts learned from this study about the distribution of the number of moves required by Heapsort are presented.

1. Introduction

Heapsort is a classic sorting method due to Williams [10] and Floyd [3]. It can be used to sort an array inplace in $O(N \lg N)$ steps; its primary disadvantage is that the inner loop is comparatively long, so that implementations tend to be about twice as slow as Quicksort, for example. Though the empirical evidence in support of this conclusion is rather persuasive, the algorithm has not been precisely analyzed, and we seek relevant mathematical results.

The method is based on maintaining a heap-ordered complete tree, stored in an array in level order: A heap of `N` keys in an array `a[1..N]` has `a[i]` greater than `a[2i]` and `a[2i+1]` for $1 \leq i \leq \lfloor N/2 \rfloor$, or, equivalently, `a[i]` is less than `a[i div 2]` for $2 \leq i \leq N$. Program 1 is a Pascal implementation that builds a heap and sorts the array after the heap is built, using the common procedure `siftdown` for both tasks:

```
procedure siftdown(k: integer);
  begin
  v:=a[k];
  while k<=N div 2 do
     begin
     j:=k+k;
     if j<N then if a[j]<a[j+1] then j:=j+1;
     if v>=a[j] then goto 0;
     a[k]:=a[j]; k:=j;
     end;
0: a[k]:=v
   end;
for k:=N div 2 downto 1 do siftdown(k);
repeat t:=a[1]; a[1]:=a[N]; a[N]:=t; N:=N-1; siftdown(1) until N<=1;
```

Program 1. Heapsort.

If the subtree rooted at `a[k]` is heap-ordered except possibly at the root, `siftdown` heap-orders it by exchanging the root with the larger of its two sons and moving down the tree. The array is heap-ordered in a "bottom-up" fashion by using `siftdown`, proceeding backwards through the array. Then, the array is sorted by extracting the largest element: exchanging it with the element in the last position, reducing the size of the heap by one, and

* Supported in part by the National Science Foundation and in part by the Institute for Defense Analyses, Princeton, NJ.

** Supported by a National Science Foundation Graduate Fellowship.

using `siftdown` to repair the damage. We call this process "sorting down" the heap. Further information on the implementation and operation of Heapsort may be found in [6] or [9].

Despite its prominence as a fundamental method for sorting and for implementing priority queues, comparatively little is known about the performance characteristics of Heapsort. Though the algorithm is simply stated and implemented, derivation of a precise mathematical description of its performance seems to be difficult. It is the only sorting algorithm in [6] for which Knuth is unable to give a precise formula for either the minimum, maximum or the average running time. This paper describes some approaches to progress on this problem.

It is easy to establish that in the worst case, `siftdown` needs to travel to the bottom of the heap on each call, so that the number of data moves made during the algorithm is not greater than

$$\sum_{N \geq i \geq 1} \lfloor \lg(N/i) \rfloor + \sum_{1 \leq i \leq N} \lfloor \lg i \rfloor = N \lg N + O(N),$$

but little precise information about the performance of the algorithm is available beyond this. For example, the number of times the statement `j:=j+1` is executed, in the worst case, is not known ([6], Ex. 5.2.3-30).

The distribution of the number of data moves, assuming all permutations to be equally likely as input, seems to be extremely flat: for example, experiments involving generating random heaps of 2^{19} elements typically give a sample standard deviation of less than 15. On the basis of such experiments, it is reasonable to make the conjecture that the distribution for this quantity is asymptotically flat: that all heaps require $N \lg N + O(N)$ moves.

Our main result is a rather intricate construction that shows this conjecture to be false because the *best case* is $\sim \frac{1}{2} N \lg N$. Thus, the possibility that the coefficient of $N \lg N$ in the average case is less than 1 is left open. Note carefully that each data move involves two comparisons, so this is no violation of the $\lg N!$ lower bound for all comparison-based sorting methods.

Other facts that were learned about the distribution of the number of moves during the development of this result are also presented.

2. Counting Heaps

Let $f(N) \equiv \{\text{the number of heaps of } N \text{ distinct elements}\}$. Since the elements are distinct, we can assume without loss of generality that they are from a permutation. Since the root must be N and there is no restriction on the subtrees, we must have

$$f(N) = \binom{N-1}{S_1} f(S_1) f(S_2),$$

where $S_1 + S_2 = N - 1$ are the sizes of the subtrees of the root. Dividing by $N!$ gives

$$\frac{f(N)}{N!} = \frac{1}{N} \frac{f(S_1)}{S_1!} \frac{f(S_2)}{S_2!}$$

which telescopes to give the result

$$f(N) = \frac{N!}{\prod_{1 \leq k \leq N} \{\text{size of the subtree rooted at } k\}}.$$

This formula, derived in a different way, is given in Knuth [6]. The same argument works for any heap-ordered tree: the number of ways to label any tree with the integers 1 through N such that every node is larger than its two sons is $N!$ divided by the product of all the subtree sizes.

For example, the number of heaps of 13 elements is

$$\frac{13!}{13 \cdot 7 \cdot 5 \cdot 3 \cdot 3 \cdot 3} = 506880$$

Table 1 gives the exact value of $f(N)$ for small N.

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$f(N)$	1	1	2	3	8	20	80	210	896	3360	19200	79200	506880	2745600	21964800

Table 1. Distinct Heap Counts

Two facts are evident from Table 1 and the above discussion and should be carefully noted. First, the independence of the subheaps can be used to prove that the bottom-up heap construction procedure "preserves randomness": if each of the N! permutations is equally likely before the construction process, then each of the $f(N)$ heaps is equally likely after. This makes it possible for Knuth to derive accurate formulas for the average-case performance of the heap construction process. Unfortunately, the second fact to be noted is that the sortdown process does not preserve randomness (far from it): simple numeration says that the sorting procedure cannot preserve randomness because successive elements in the table do not divide. If each of the 3 heaps of 4 elements are equally likely before the first sorting step, how could each of the 2 heaps of three elements elements be equally likely afterwards?

It is a straightforward calculation to continue from the above formula to derive an asymptotic expression:

Lemma. *The number of different heaps which can be formed from N distinct elements is*

$$4(N+1)!\left(\frac{e^{\alpha}}{4}\right)^{N+1}\left(1+O\left(\frac{1}{N}\right)\right),$$

where $\alpha = \sum_{k\geq 1}\frac{1}{2^k}\ln\left(\frac{2^k}{2^k-1}\right) \approx .440539+$.

Proof: Omitted.

Using Stirling's approximation, this means that

$$\begin{aligned} f(N) &\approx e^{\alpha}\sqrt{2\pi}N^{3/2}\left(\frac{N}{4e^{1-\alpha}}\right)^{N} \\ &\approx 3.9N^{3/2}\left(\frac{N}{7}\right)^{N} \end{aligned}$$

For example, for $N = 15$, this approximation gives 2×10^7, in agreement with the table above, and it says that there are more than 7×10^{22} heaps of size 31.

3. Generating Heaps and Exact Results for Small N

Given a heap of size $N-1$, it is convenient to consider working backwards to generate all heaps of size N that yield that heap after one `siftdown` operation. There are exactly `a[N div 2]` such heaps, which can be generated by, for each element less than or equal to `a[N div 2]`, performing the "pulldown" operation given in Program 2. This procedure can be used as the basis for an efficient program to generate all heaps: for each heap of size $N-1$, generate `a[N div 2]` heaps of size N by applying `pulldown` appropriately.

```
procedure pulldown(k: integer);
  begin
  a[N]:=a[k];
  while k<>1 do
    begin j:=k div 2; a[k]:=a[j]; k:=j end
  a[1]:=N;
  end;
```

Program 2. Pulling a new element down into a heap.

Figure 1 shows how the heaps of size 5 are generated. In this "tree of heaps", the `pulldown` procedure can be used to move down, and the `siftdown` procedure to move up, so only the heap and a small amount of state

information about which `pulldowns` have been done need be kept during the generation procedure. This "bottom-up" generation method seems more convenient than a "top-down" method corresponding to the counting formula in the previous section. (It is interesting to contemplate whether there might be some combinatorial identity implied by that counting method and this generation method.) The full distribution of the number of moves required to sort-down the more than twenty-five million heaps of 15 elements or fewer, computed using this method, is given in the Appendix.

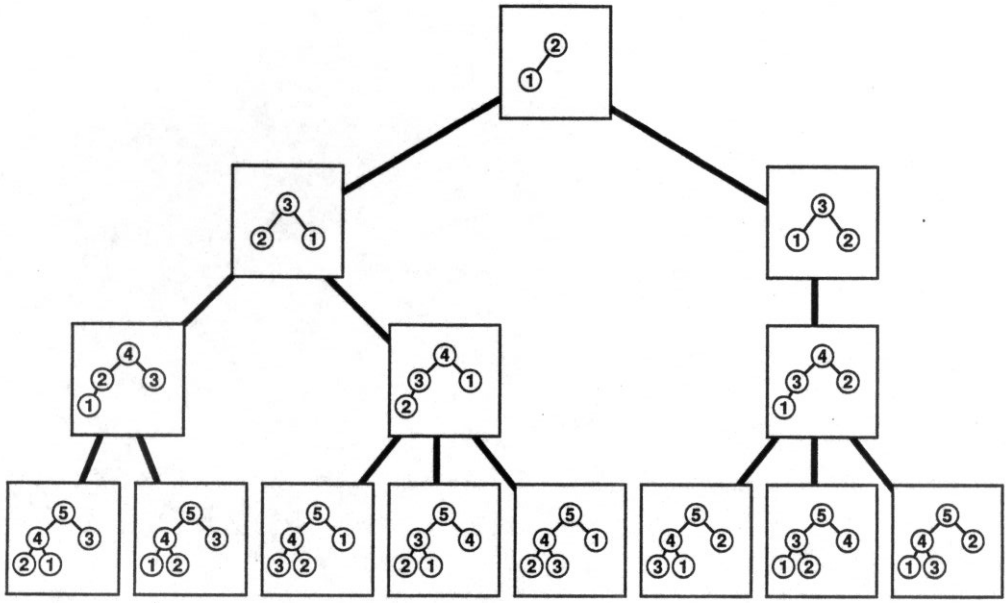

Figure 1. Generating Heaps.

In the present context, our interest is in the fact that this heap generation procedure naturally converts to a backtracking procedure to search for the best case. We simply rearrange the order of doing `pulldown` so that the best is done first, and maintain a cutoff to not generate heaps which cannot beat the best generated so far. This substantially reduces the number of heaps to be examined, though not as much as one might hope: for $N = 25$ hundreds of billions of heaps still pass the cutoff. Table 2 shows the cost of (number of moves required to sort down) the best heaps for $N < 25$. On the one hand, 25 is a dangerously small number from which to draw conclusions; on the other hand, the fact that this function grows only by 2 as N increases by 1 from 16 to 25 (except for 19) lends credence to the conjecture that the coefficient of $N \lg N$ in the best case is not 1, but 1/2.

N	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Best Case	1	2	3	4	6	7	9	11	12	14	16	17	20	22	24	26	29	31	33	35	37	39	$\leq$41

Table 2. Best-Case Costs

Furthermore, a careful examination of how a known best case heap operates gives some intuition on how to construct larger best-case heaps. Figure 2 shows the sortdown process for a best heap of size 24: A low cost seems to be achieved by alternating short paths with long paths, for an average cost of about half the heap height. In the next section we show how to generalize this to construct heaps of N nodes for which the average cost of a `siftdown` is $\frac{1}{2} \lg N$ for any large N.

4. Tight Asymptotic Bounds on the Best Case

Heap construction is linear for an array which is already a heap, so to study the best case of Heapsort, we need only consider the cost of sorting-down a heap. If equal keys are allowed, the best case is clearly linear [2]: consider

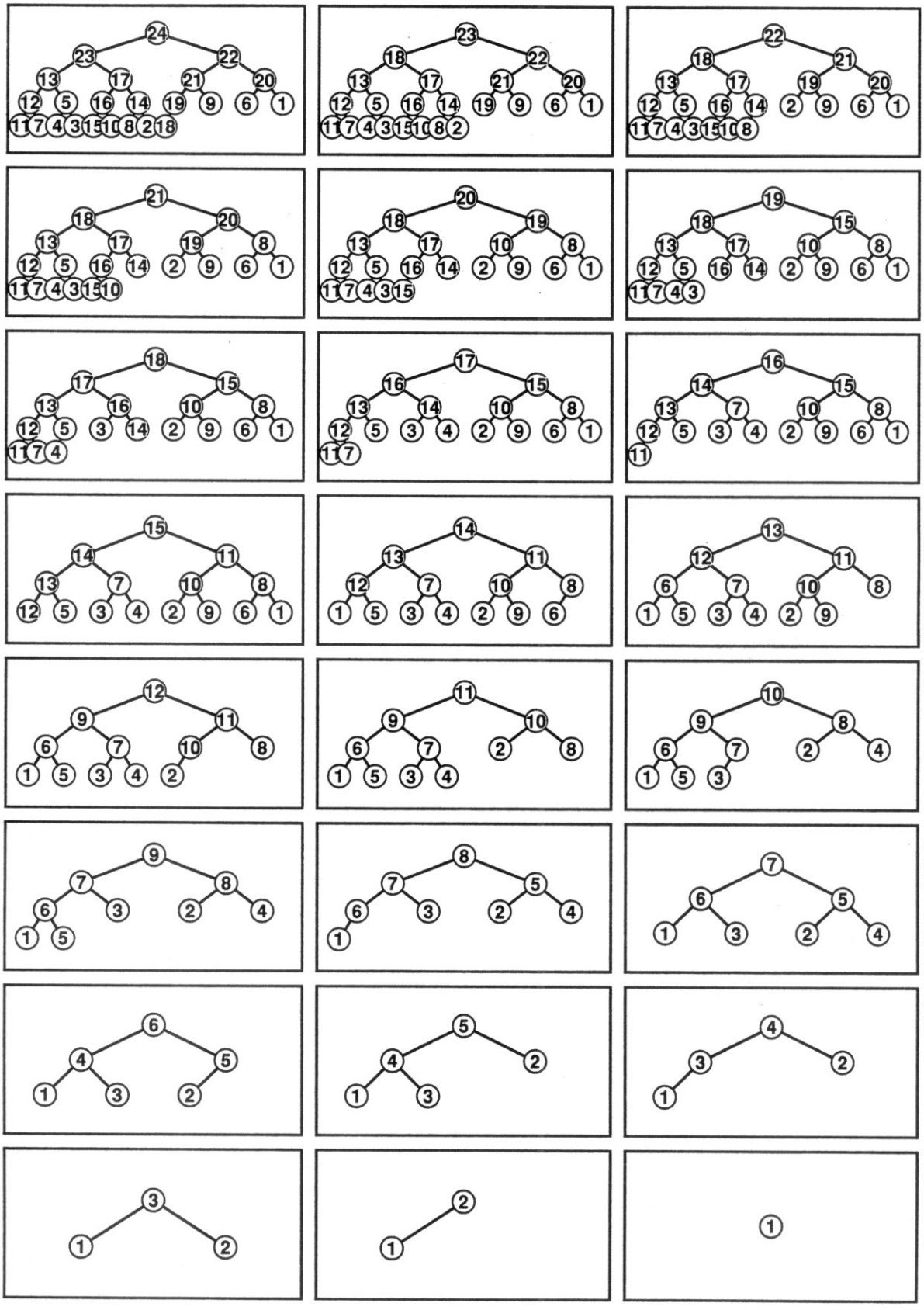

Figure 2. A best-case heap.

the case of a heap with all keys equal. For distinct keys (or equivalently, a permutation) the situation is far less clear. Do all heaps require asymptotically the same number of moves as the worst case for sorting down? If not, are there heaps which can be sorted down in linear time? In this section, we settle these questions by showing that the best case of the number of moves taken by Heapsort is $\sim \frac{1}{2} N \lg N$.

First, we consider the lower bound. This is developed by observing what must happen to the largest keys as the heap is sorted down. We have:

Theorem 1. *Heapsort requires at least $\frac{1}{2} N \lg N + \Omega(N)$ data moves for any heap composed of distinct keys.*

Proof: Consider the cost of sorting-down a large heap of size $N = 2^n - 1$. Specifically, we count the number of moves made during the first half (2^{n-1}) of the sort-down operations. Note that the 2^{n-1} largest of the keys in the original heap must be displaced by these sort-downs, and that they form a heap-ordered subtree within the original heap.

Now, the cost of sorting-down is at least the internal path length of this subtree, *except* that nodes at level n are not counted. This cost is minimized by a subtree that has the root plus either its left or right subtree (complete). Such a subtree has 2^{n-2} nodes at level n, with the average level of the other 2^{n-2} nodes about $n-2$. This gives an average cost for the first 2^{n-1} siftdown operations of at least $n/2 + \Omega(1)$.

The same argument holds each time the size of the heap is halved, and can easily be extended to handle all values of N. ∎

We develop a matching upper bound by constructing a heap which has the "alternating" flavor of the heap of size 24 shown in the previous section. The construction is more easily seen by working backwards (using `pulldown`): the goal is to build a heap in this way using only $\sim \frac{1}{2} N \lg N$ moves. We do so by pulling some large elements down to the bottom of the heap, then using those large elements to pull down an equal number of elements from near the top of the heap, then iterating the process.

Theorem 2. *The best case of Heapsort requires $\frac{1}{2} N \lg N + O(N \log \log N)$ moves.*

Proof: This is a long, technical proof, so we begin by sketching its general plan. We build a heap of size $N = 2^n - 1$ by starting with a large heap with certain properties and using the pulldown process to finish the construction. For a large heap with somewhat fewer than $2^n - 1$ elements, and a parameter k whose value will be chosen appropriately later, consider two consecutive subheaps A and B of size 2^k at the bottom of the heap with the property that all the elements in the second can be "pulled down" by the elements at the bottom of the first. (For example, if all the elements in A were larger than each of the elements in B, this property would be satisfied.) Now, 2^{k+1} pulldowns can be performed in such a heap at an average cost of $\frac{1}{2} n$, as follows: First, pull down each element in B by the elements in A (average cost: $n-1$). Now note carefully that this makes all but $n-k$ of the elements of B larger than every element in the original heap. In particular, the elements of B can be used to pulldown the elements in the 2^k subtree on the other side of the root from B (average cost: $k-1$). This analysis ignores what happens to the nodes along the path from the root to A and B. If k is chosen to be $\Theta(\log n) = \Theta(\log \log N)$ these costs are absorbed, and the average cost of the 2^{k+1} pulldowns is $\frac{1}{2} n$.

To iterate this construction, we must start with a sufficiently large heap for which the bottom heaps of size 2^k are ordered as required above. Then it suffices to show that the pulldown operations described above preserve the ordering property. They do not do so exactly, but they leave a symmetric property on the next level that suffices to keep the process going.

To complete and give precision to the proof sketch just given, we consider the heap structure shown in Figure 3. Let k be the smallest even integer not less than $\lg n$, and let α be a variable initialized to 0. The construction process maintains a heap of height $n - k + 2\alpha$, beginning with height $n-k$ and finishing with height n and $\alpha = k/2$. At the top, we define C and D to be the height $2(k+\alpha)+1$ subheaps rooted at `a[3]` and `a[2]` respectively. In the figure, the roots of these subheaps are labelled, and the subheaps explicitly drawn, for $k + \alpha = 2$. Now, let $p = 2^{n-3k-1}$. Across the bottom, for $1 \le i \le p$ we define A_i to be the subheap rooted at `a[`$2^{n-3k} + 2i - 2$`]` and B_i to be the subheap rooted at `a[`$2^{n-3k} + 2i - 1$`]`. Since the roots of the $2p$ subheaps remain fixed, they grow as we add nodes to the bottom of the heap.

We begin by placing the smallest $2^{(n-k)} - 1$ keys into the initial structure given in Figure 3, then we complete the construction by pulling keys down into the heap in $k/2$ passes. Each pass increases the variable α by one; this increases the height of the heap by 2.

To describe relationships among sets of keys in the heap structure, we adopt the following notational conventions: First, For sets S and T, take $S > T$ to mean that every key of S is greater than any key of T. Second, given

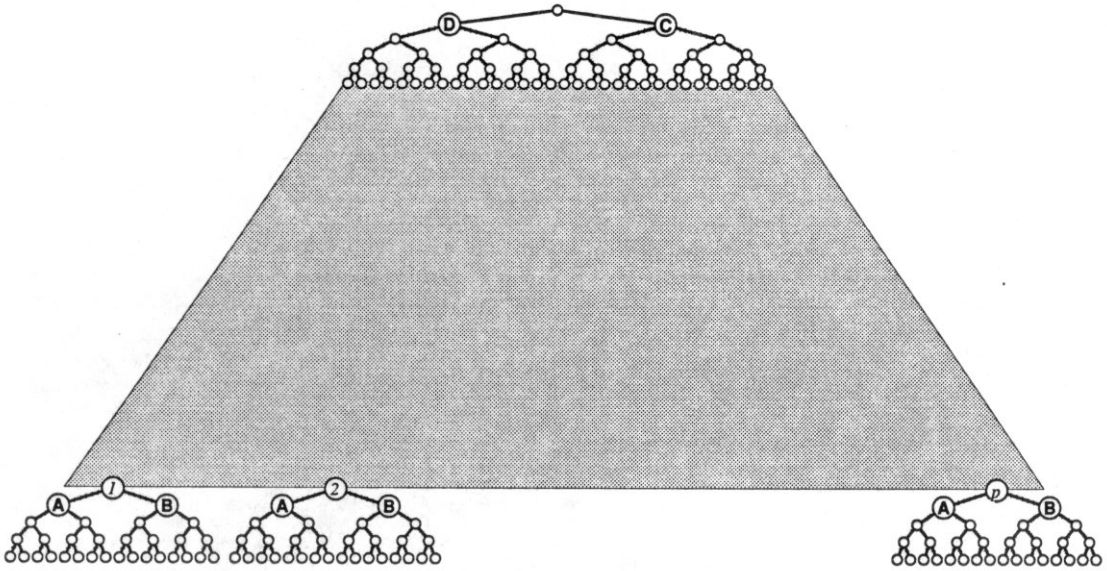

Figure 3. Initial Structure for Building the Best-Case Heap ($k + \alpha = 2$)

a subheap H and a positive integer x, we define $L(H, x)$ to be all but the smallest x keys of H and $S(H, x)$ to be all but the largest x keys of H.

Now, define $t = 2n\alpha$. The following "pass invariants" are required to hold for all i, initially and at the end of each pass:

(i) $L(A_i, t) > S(B_i, t)$.
(ii) $L(C, 2^{2(k+\alpha)} - 1 + t + n) > A_i$ for i, $1 < i \leq p/2$.
(iii) $L(C, 2^{2(k+\alpha)} - 1 + t + n) > B_i$ for i, $1 < i \leq p/2$.

Each pass is broken down into two phases, the steps of which we now give in detail. For each step, we give at the right an upper bound on the number of moves required to execute it once:

	Cost Bound
Phase AB:	
(i) for i from 1 to $p/2$ do	
(a) Pull down the $t + 1$ smallest keys in A_i.	$n(2n\alpha + 1)$
(b) Pull down all but the largest t keys in B_i.	$n2^{2(k+\alpha)}$
(c) Pull down each of the smallest $t + n$ keys in B_i.	$n(2n\alpha + n)$
(d) Pull down the largest $2^{2(k+\alpha)} - t - n$ keys in C.	$(2(k + \alpha) + 1)2^{2(k+\alpha)}$
(ii) for i from $p/2 + 1$ to p do as in (i), replacing C with D.	
Phase BA:	
(i) Pull down the smallest key in A_1, $2^{2(k+\alpha)+1}$ times.	$n2^{2(k+\alpha)+1}$
(ii) for i from 1 to $p/2 - 1$ do	
(a) Pull down the smallest $t + n + 1$ keys in B_i.	$n(2n\alpha + n + 1)$
(b) Pull down all but the largest $t + n$ keys in A_{i+1}.	$n2^{2(k+\alpha)+1}$
(c) Pull down each of the smallest $t + 2n$ keys in A_{i+1}.	$n(2n\alpha + 2n)$
(d) Pull down the largest $2^{2(k+\alpha)+1} - t - 2n$ keys in C.	$(2(k + \alpha) + 1)2^{2(k+\alpha)+1}$
(iii) for i from $p/2$ to $p - 1$ do as in (ii), replacing C with D.	
(iv)	
(a) Pull down the smallest $t + n + 1$ keys in B_p.	$n(2n\alpha + n + 1)$
(b) Pull down the smallest $2^{2(k+\alpha)+1} - t - n - 1$ keys of A_1.	$n2^{2(k+\alpha)+1}$
(v) increment α	

Before tallying the number of key moves required by the above construction we must check that every step in the algorithm can be executed. This requires establishing that all three parts of the pass invariant can be made to hold initially and at the end of each pass. First we show that all three parts of the invariant can be made to hold initially. Then, we show that Phase AB followed by Phase BA preserves the invariant: if it holds after α passes have been completed, then it will hold after pass $(\alpha+1)$ has been completed.

Initially Establishing the Invariant:

> It is clear that one can place keys into the structure of Figure 3 so that the invariant is satisfied. We move through the structure, assigning the keys in increasing order, as follows: Fill in the A_i and B_i moving from right to left, preserving heap order by moving in reverse level order from bottom to top within each, then complete by proceeding in reverse level order.

Correctness Proof for Phase AB:

(i) The purpose of this loop is to add keys to the bottom of the left half of the heap. During the loop we maintain the invariant that $L(C, 2^{2(k+\alpha)} - 1 + t + n) > A_j$ and $L(C, 2^{2(k+\alpha)} - 1 + t + n) > B_j$ for $j > i$. This invariant holds at the beginning of the loop by (ii) and (iii) of the pass invariant. As we begin the i'th iteration of the loop we know that $L(A_i, t) > S(B_i, t)$; this was implied by (i) of the pass invariant at the beginning of the loop, and A_i and B_i have not been touched since then.

- (a) We begin level $2(k+\alpha)+1$ of A_i by pulling down the $t+1$ smallest keys in A_i. We note that any key that enters A_i in this step is greater than every key of B_i since at one point it had the root of B_i as its right child. Also, any key of A_i that is smaller than some key of B_i ends up at level $2(k+\alpha)+1$. It follows that every key at level $2(k+\alpha)$ of A_i is greater than any key of $S(B_i, t)$.
- (b) Since every key of $S(B_i, t)$ is smaller than any key at level $2(k+\alpha)$of A_i, we can pull down $2^{2(k+\alpha)} - 1 - t$ keys of B_i to complete level $2(k+\alpha)+1$ of A_i.
- (c) In step (b) only t keys of B_i failed to be replaced; by pulling down the smallest $t+n$ keys in B_i, we can pull to the bottom level these t keys plus any keys that had been along the path from the root of the heap to the root of B_i at the beginning of step (b). This fills the first $t+n$ positions of level $2(k+\alpha)+1$ of B_i and ensures that all keys at or above level $2(k+\alpha)$ of B_i are new to the heap since the beginning of step (a).
- (d) Since all keys at or above level $2(k+\alpha)$ of B_i are new to the heap since the beginning of step (a) and since C has not been touched since before step (a), the largest $2^{2(k+\alpha)} - t - n$ keys in C can be pulled down to complete level $2(k+\alpha)+1$ of B_i. Since the largest $2^{2(k+\alpha)} - t - n$ keys in C are replaced with keys that are new to the heap they remain greater than any key contained in A_j or B_j for $j > i$; this satisfies the loop invariant.

(ii) This loop serves the same purpose as (i) except that it adds keys to the bottom of the right side of the heap. Note that in step (i), every key of D was replaced by a key that is new to the heap since the beginning of this phase; since none of the A_i or B_i for $i > p/2$ have been touched since the beginning of the phase we know that $L(D, 2^{2(k+\alpha)} - 1 + t + n) > A_i$ and $L(D, 2^{2(k+\alpha)} - 1 + t + n) > B_i$ for all $i > p/2$. We can thus maintain the invariant that, during this loop, $L(D, 2^{2(k+\alpha)} - 1 + t + n) > A_j$ and $L(D, 2^{2(k+\alpha)} - 1 + t + n) > B_j$ for all $j > i$.

Fix i, $1 < i \leq p/2$. At the end of Phase AB, A_i contains the keys it contained at the beginning of the phase and all but t of the keys that B_i contained at the beginning of the phase. It follows that with the exception of fewer than $t+n$ keys, every key in A_i at the end of the phase was in A_i or B_i at the beginning of the phase.

Now consider the origin of the keys in B_{i-1} at the end of Phase AB. As observed above, at the end of step (c) of the $i-1$'st iteration of loop (i), all but at most $t+n$ keys of B_{i-1} are new to the heap since the beginning of the phase; by the previous paragraph, these are greater than any key in $S(A_i, t+n)$ at the end of Phase AB. Also by the invariant maintained during loop (i), the keys in C that are pulled down into B_{i-1} in step (d) are greater than any key in $S(A_i, t+n)$. We conclude that at the end of Phase AB, $L(B_{i-1}, t+n) > S(A_i, t+n)$.

The same reasoning shows that at the end of Phase AB, $L(B_{i-1}, t+n) > S(A_i, t+n)$ for $i > p/2$. Furthermore, every key pulled down from D into B_p is new to the heap since the beginning of the phase and is thus greater than any key in $S(A_1, t+n)$; we conclude that $L(B_p, t+n) > S(A_1, t+n)$.

Finally, note that every key of C was replaced during step (ii) by a key that had not been in the heap prior to step (ii); at the same time, no key of A_i or B_i for $i \leq p/2$ was touched during step (ii). We thus have the following situation as we begin Phase BA:

(i) $L(B_{i-1}, t+n) > S(A_i, t+n)$ for $i > 1$.
(ii) $L(B_p, t+n) > S(A_1, t+n)$.
(iii) $L(C, t+2n-1) > A_i$ for all $i \leq p/2$.
(iv) $L(C, t+2n-1) > B_i$ for all $i \leq p/2$.

Correctness Proof for Phase BA:

(i) Pulling down the smallest key in A_1 $2^{2(k+\alpha)+1}$ times completes level $2(k+\alpha)+2$ of A_1.
(ii) This loop extends level $2(k+\alpha)+2$ across all but $B_{p/2}$ on the left half of the heap. As before, we know at the beginning of the i'th iteration of the loop that $L(B_i, t+n) > S(A_{i+1}, t+n)$.
  (a) When we begin this step there have been at least $2^{2(k+\alpha)+1}$ pulldowns of keys of A_i since the beginning of Phase BA. Every key along the path from the root of the heap to the root of B_i has thus been replaced with a key that is new to the heap since the beginning of Phase BA. This implies that every key to enter B_i in this step is new to the heap since the beginning of Phase BA. Since A_{i+1} has not been touched since Phase AB and since this step pulls the $t+n+1$ smallest keys of B_i down to level $2(k+\alpha)+2$, it follows that every key at level $2(k+\alpha)+1$ of B_i is larger than any key of $S(A_{i+1}, t+n)$.
  (b) This step completes level $2(k+\alpha)+2$ of B_i.
  (c) This step ensures that every key at or above level $2(k+\alpha)+1$ of A_{i+1} is new to the heap since the beginning of Phase BA.
  (d) This step completes level $2(k+\alpha)+2$ of A_{i+1}. As in Phase AB, the largest $2^{2(k+\alpha)+1} - t - 2n$ keys in C are replaced with keys that are new to the heap so they remain greater than any key in A_{j+1} or B_j for $j > i$.
(iii) This loop acts in the same way as (ii), extending level $2(k+\alpha)+2$ to all but the bottom of B_p.
(iv) This step fills in level $2(k+\alpha)+2$ of B_p.
  (a) Pulling down the smallest $t+n+1$ keys in B_p has the same effect as step (ii)a. above; every key at level $2(k+\alpha)+1$ of B_p is now greater than all but at most $t+n$ keys that were in A_1 prior to beginning Phase BA.
  (b) We complete level $2(k+\alpha)+2$ of B_p by pulling down the smallest $2^{2(k+\alpha)+1} - 1 - t - n$ keys of A_1. With the exception of the keys that were between the root of the heap and the root of A_1 at the beginning of the phase and the smallest $t+n$ keys not pulled down in this step, a total of fewer than $t+2n$ keys, every key in A_1 is new to the heap since the beginning of the phase.

We can now show that for $i > 1$ and α unincremented, $L(A_i, t+2n) > S(B_i, t+2n)$ holds by the same reasoning used to establish the analogous result for the B_{i-1} and A_i at the end of Phase AB; as before, the keys in $S(B_i, t+2n)$ were in B_i and A_{i+1} at the beginning of the phase while those in $L(A_i, t+2n)$ are either new to the heap or come from C or D whose keys are larger than those in $S(B_i, t+2n)$. That every key of $L(A_1, t+2n)$ is new to the heap during Phase BA follows from the comment at the end of (iv)b. above. When we increment α to begin a new pass, we find that (i) of the pass invariant continues to hold.

Parts (ii) and (iii) of the pass invariant continue to hold by the reasoning used before, that since the beginning of Phase BA, every key of C has been changed while for i, $1 < i \leq p/2$, A_i and B_i have remained untouched.

We now compute an upper bound on the number of key moves required to complete the construction. We proceed by summing the bounds given earlier on the number of moves performed in individual steps. The following is an upper bound on the work performed in the (c) steps of both loops in both phases. It is also an upper bound on the work performed in the (a) steps of BA(iv) and of both loops in both phases:

$$2^{n-3k} \sum_{0 \leq \alpha < k/2} n^2(2\alpha + 2) = O(n^2 2^{n-3k} k^2)$$

$$= O\left(\frac{n^2 2^n \log^2 n}{n^3}\right)$$
$$= O(N)$$

The total work performed in the (b) steps of BA(iv) and of both loops in both phases is bounded above by:

$$2^{n-3k-1} \sum_{0 \le \alpha < k/2} n2^{2(k+\alpha)} + 2^{n-3k-1} \sum_{0 \le \alpha < k/2} n2^{2(k+\alpha)+1}$$
$$= 2^{n-3k-1} n 2^{2k} \left(\sum_{0 \le \alpha < k/2} 2^{2\alpha} + \sum_{0 \le \alpha < k/2} 2^{2\alpha+1} \right)$$
$$= n2^{n-k-1} \sum_{0 \le \alpha < k} 2^{\alpha}$$
$$= n2^{n-1} - n2^{n-k-1}$$
$$= \frac{1}{2} N \lg N + O(N)$$

The total work performed in the (d) steps of both loops in both phases is bounded by:

$$2^{n-3k} \sum_{0 \le \alpha < k/2} (2(k+\alpha)+1) 2^{2(k+\alpha)+1} < 2^{n-3k} 3k 2^{2k} \sum_{0 \le \alpha < k/2} 2^{2\alpha+1}$$
$$< 3k 2^{n-k} 2^k$$
$$= O(N \log \log N)$$

The total work performed in BA(i) is bounded above by:

$$\sum_{0 \le \alpha < k/2} n2^{2(k+\alpha)+1} < \sum_{0 \le \alpha < k/2} n2^{3k} < kn2^{3k} = O(N)$$

So far we have counted the costs of all phases of the construction, or, equivalently, the cost of sorting down the final heap to the initial heap of height $n-k$ where the construction started (with $\alpha = 0$). To this we must add the cost of sorting down that initial structure. As noted in Section 1, this number is bounded above by

$$(n-k)2^{n-k} + O(2^{n-k}) \le \frac{n2^n}{2^k} + O(N) \le \frac{n2^n}{n} + O(N) = O(N)$$

We conclude that total number of moves required to sort down the constructed heap is $\frac{1}{2} N \lg N + O(N \log \log N)$ as desired. ∎

5. Concluding Remarks

As mentioned above, if only comparisons are counted, Heapsort seems to be relatively inefficient because, during the `siftdown` operation, two comparisons are used at each step, one to determine the larger of the two sons of the current node, the other to determine whether the current node is larger than both its sons (so the loop should be exited). Floyd (see [6]) suggested that comparisons could be saved by eliminating the latter type (so the loop terminates at some point of the bottom of the heap), then moving *up* the heap, using a procedure like `pulldown`, until the proper place for the element being sifted down is found. Most of the time, only a few steps back up are required.

Note that the best-case results for Heapsort of the previous section translate into *worst-case* results for the variant just described. Thus, Floyd's method requires $\sim \frac{3}{2} N \lg N$ comparisons in the worst case, not the optimal $N \lg N + O(N)$ that might have been hoped for. If extra storage is available, Gonnet and Munro [5] have pointed out that $\sim N \lg N$ can be achieved simply by relaxing the constraint that the heap be a complete tree, and sorting down by removing the root to an auxiliary array, then promoting the larger of the two sons down the tree, marking the spot reached on the bottom as vacant. But the inplace feature is one of Heapsort's main virtues.

The major open problem is still the analysis of the average case. Is the coefficient of $N \lg N$ in the expression for the average number of moves required to sortdown a random heap of N elements 1, or not? We do have the following result:

Lemma. *In a random heap, the average cost of the first "sort-down" is between* $\lceil \lg N \rceil$ *and* $\lceil \lg N \rceil - 1$.

Proof: Consider the correspondence between each heap of size $N-1$ and heaps of size N which "sort-down" to it implied by `pulldown`. Given a heap A with $N-1$ nodes, find the highest element less than `a[N div 2]`. This element is at the root of a complete subheap of elements all less than `a[N div 2]`: the average level of these elements is between the bottom and one up from the bottom. The result follows from iterating this process until all nodes less than `a[N div 2]` have been considered. ∎

(This quantity was also studied by Doberkat [1].) This proof does not work beyond one step of the Heapsort algorithm, because after one sort-down we no longer have the property that all heaps are equally likely.

The average case is difficult because Heapsort does not "preserve randomness". One reason for studying the algorithm so closely is the possibility that there exists a variant which does preserve randomness (for example some variant of pairing heaps [4]) which would perhaps not only submit to analysis but also have some better performance characteristics. Full understanding of Heapsort might lead to the discovery of such a variant.

The results of this paper destroy the attractive conjecture that Heapsort's running time is asymptotically flat, but leave open the door for the development of a variant of the algorithm that uses half as many comparisons as the standard algorithm. Floyd's variant would seem to be a likely candidate for this, but the results of this paper also suggest that average-case results are needed to prove that heaps can be used as the basis for an asymptotically optimal sort.

References

1. E.-E. DOBERKAT. "Deleting the root of a heap," *Acta Informatica* **17**, 3 (1982).
2. H. ERKIÖ. "On Heapsort and its dependence on input data," Technical Report No. A-1979-1, Dept. of Computer Science, University of Helsinki, Finland.
3. R. W. FLOYD. "Treesort 3: Algorithm 245," *Comm. of the ACM* **7**, 12 (1964).
4. M. FREDMAN, R. SEDGEWICK, D. SLEATOR, and R. TARJAN. "Pairing Heaps: a new form of self-organizing priority queue," *Algorithmica* **1**, 1 (1986).
5. G. H. GONNET AND I. MUNRO. "Heaps on heaps," *SIAM J. on Computing* **15**, 4(1986).
6. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass. (1973).
7. D. E. KNUTH AND A. SCHÖNHAGE. "The expected linearity of a simple equivalence algorithm," *Theoretical Computer Science* **6**, (1978).
8. T. PORTER AND I. SIMON. "Random Insertion into a Priority Queue Structure," *IEEE Trans. on Software Engineering* **SE-1**, 3 (1975).
9. R. SEDGEWICK. *Algorithms*, Addison-Wesley, 1983.
10. J. W. J. WILLIAMS. "Algorithm 232: HEAPSORT," *Comm. of the ACM* **7**, (1964).

Appendix

The full distribution of sorting-down costs for Heapsort, computed using the method described at the beginning of section 3, is given in Table 3. Though few simply expressed relationships among these numbers are evident, the underlying distribution seems to be rather stable. We may infer that an exact formula to describe the average-case for all N are likely to be unavailable or too complicated to be useful, but that approximate formulae may be within reach.

	5	6	7	8	9	10	11	12	13	14	15
2	2										
3	4	3									
4	2	8	5								
5		7	21								
6		2	31	6							
7			19	46	2						
8			4	86	32						
9				59	163	8					
10				13	314	97					
11					270	462	46				
12					102	975	465	5			
13					13	1051	1988	128			
14						594	4426	1142	56		
15						159	5676	5142	799		
16						14	4322	13336	5312	171	
17							1866	21139	21664	2481	10
18							387	20865	58776	16843	858
19							24	12552	107700	71465	11868
20								4230	132629	209573	82046
21								639	107582	436963	360027
22								22	54769	645556	1111286
23									15665	662679	2516164
24									1898	455735	4214038
25									30	195602	5165140
26										44837	4525046
27										3610	2719507
28										85	1035933
29											209319
30											13193
31											365

Table 3. Full Distribution of Sorting-Down Costs