THE ANALYSIS OF HEAPSORT

Russel Schaffer
Robert Sedgewick

CS-TR-330-91

January 1991

Revised January 1992

The Analysis of Heapsort

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

Abstract: Heapsort is a fundamental algorithm whose precise performance characteristics have proven difficult to analyze. It is easy to show that the number of keys moved during the algorithm when sorting a random file of N distinct elements is $N \lg N + O(N)$ in the worst case, and it has long been conjectured that the average case performance is the same. No specific results on the average case or even the best case have been found despite the algorithm's standing as a classic method that is in widespread use. In this paper, we resolve these questions by showing that the best case for the number of moves is $\sim \frac{1}{2} N \lg N$ and that the average number of moves is $\sim N \lg N$.

These results have implications for the analysis of various modified versions of the algorithm that have been suggested. In particular, they imply that a well-known variant originally due to Floyd uses an asymptotically optimal number of comparisons on average, but three-halves that in the worst case.

This essentially completes the analysis of the algorithm, though there is another quantity that contributes to the leading term of the running time that requires more intricate arguments, and we have little specific information about the distribution beyond what is implied by our asymptotic results.

1. Introduction

Heapsort is a classic sorting method due to Williams [13] and Floyd [6]. 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, precise analysis has been elusive, 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.

* Supported by a National Science Foundation Graduate Fellowship.

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

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 children 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 key: exchanging it with the key in the last position, reducing the size of the heap by one, and 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 [8] or [10].

Despite its prominence as a fundamental method for sorting and for implementing priority queues, specific analytic results about Heapsort are sparse in the literature. 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 [8] for which Knuth is unable to give a precise formula for either the minimum, maximum or the average running time. Not even the coefficient of the leading term is given by Knuth (or has been published since) for any of the important quantities that characterize the algorithm.

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.

The number of times the statement `j:=j+1` is executed also contributes to the leading term of the running time. This corresponds to the number of times we move right in the heap during `siftdown`. Not even the worst case of this quantity is given by Knuth ([8], Ex. 5.2.3-30).

But the primary quantities to be analyzed are the numbers of comparisons and data moves, and we focus on these in this paper. 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} keys 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 results in this paper show that this is not quite true, and that a similar, but slightly weaker, statement holds. First, we present a rather intricate construction that shows this conjecture to be false because the *best case* is $\sim \frac{1}{2} N \lg N$. (Note carefully that each data move involves two comparisons, so an average case of $\sim \frac{1}{2} N \lg N$ data moves would be no violation of the $\lg N!$ lower bound for all comparison-based sorting methods.) This best case bound has been derived independently by Fleischer, Sinha, and Uhrig [4], [5]. Our second and main result implies that there are not too many best case heaps because the *average* number of data moves is $\sim N \lg N$.

We also present other facts about the distribution of moves that were learned during the development of these results.

2. Counting Heaps

Let $f(N) \equiv \{\text{the number of heaps of } N \text{ distinct keys}\}$. Because we consider only heaps on distinct keys, we will always assume that an N key heap contains the integers from 1 to N. 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 [8]. 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 children is $N!$ divided by the product of all the subtree sizes.

For example, the number of heaps on 13 keys 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 should be noted from Table 1 and the above discussion. 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 on 4 keys is equally likely before the first sorting step, how could each of the 2 heaps on three keys be equally likely afterwards?

It is a straightforward calculation to continue from the above formula to derive an asymptotic expression that shows how the number of heaps grows with N:

Lemma. *The number of different heaps which can be formed from $N = 2^n - 1$ distinct keys 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: Consider a heap on $N = 2^n - 1$ keys. For each k, $1 \leq k \leq n$, there are 2^{n-k} subtrees of size $2^k - 1$ rooted at keys of the heap. The number of distinct heaps on N nodes is thus:

$$\begin{aligned}
f(N) &= \frac{N!}{\prod_{k=1}^{n} (2^k - 1)^{2^{n-k}}} \\
&= \frac{N!}{\prod_{k=1}^{n} (2^k)^{2^{n-k}}} \cdot \frac{\prod_{k=1}^{n} (2^k)^{2^{n-k}}}{\prod_{k=1}^{n} (2^k - 1)^{2^{n-k}}} = \frac{N!}{2^{\sum_{k=0}^{n-1} (n-k)2^k}} \cdot \exp\left(\sum_{k=1}^{n} 2^{n-k} \ln\left(\frac{2^k}{2^k - 1}\right)\right) \\
&= N! \cdot 2^{-2^{n+1}+n+2} \cdot \exp\left(2^n \sum_{k=1}^{n} \frac{1}{2^k} \ln\left(\frac{2^k}{2^k - 1}\right)\right) \\
&= N! \cdot \frac{4(N+1)}{4^{N+1}} \cdot e^{(N+1)\alpha}\left(1 + O\left(\frac{1}{N}\right)\right)
\end{aligned}$$

since

$$\exp\left((N+1)\sum_{k>n} \frac{1}{2^k} \ln\left(1 + \frac{1}{2^k - 1}\right)\right) < \exp\left((N+1)\sum_{k>n} \frac{1}{2^k(2^k - 1)}\right) < \exp\left((N+1)\left(\frac{1}{4}\right)^n\right) = 1 + O\left(\frac{1}{N}\right). \blacksquare$$

Using Stirling's approximation, this means that

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

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 `a[k]` less than or equal to `a[N div 2]`, performing the "pulldown" operation given in Program 2.

```
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 key down into a heap.

Figure 1 shows the result of performing `pulldown(7)` in a heap on 17 keys. Since `a[18 div 2] > a[7]`, `a[7]` can be "pulled down" to posititon `a[18]`. Replacing `a[7]` by `a[3]` and `a[3]` by `a[1]`, and assigning `a[1] := 18`, completes the expansion of the original heap to a heap on 18 keys. Performing `a[1] := a[18]`, `siftdown(1)` restores the orignal heap; in this sense, `siftdown` is the inverse of `pulldown`.

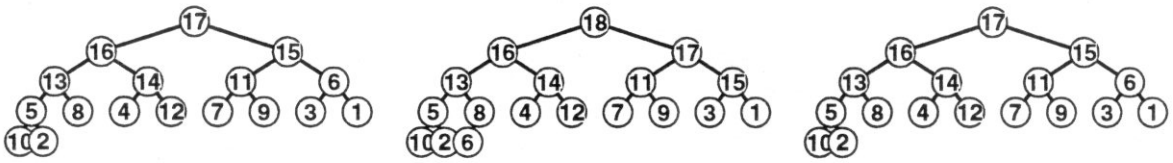

Figure 1. `pulldown(7)` followed by `a[1] := a[18], siftdown(1)`.

Sometimes when performing a `pulldown` operation it is clearer to refer to the key being "pulled down" by its value rather than its index. Figure 1 thus shows the result of pulling down 6. We might also say that 6 was pulled down by 8 since the node containing 6 becomes a child of the node containing 8.

The `pulldown` 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.

Figure 2 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.)

Each heap of size N is built by starting with the heap of size 1 and performing $N - 1$ `pulldown` operations. We associate a unique *pull-down sequence* with each heap: the sequence of the values of k used for the pulldowns. The pull-down sequences for the heaps of size 5 are given in the first line in Table 2. Each pull-down sequence has an associated *cost sequence* where each key i in the sequence is replaced by $\lfloor \lg i \rfloor$. Then the cost (number of moves) of sorting down the heap is given by summing the elements in the cost sequence. The cost sequences and costs for the heaps of size 5 (in the same order as at the bottom of Figure 1) are given in Table 2. These sequences may be viewed as the costs of constructing each heap (using the pull-down sequences given) or as the cost of sorting-down the heap (with the pull-down sequence describing where each key ends up in the sorting-down process).

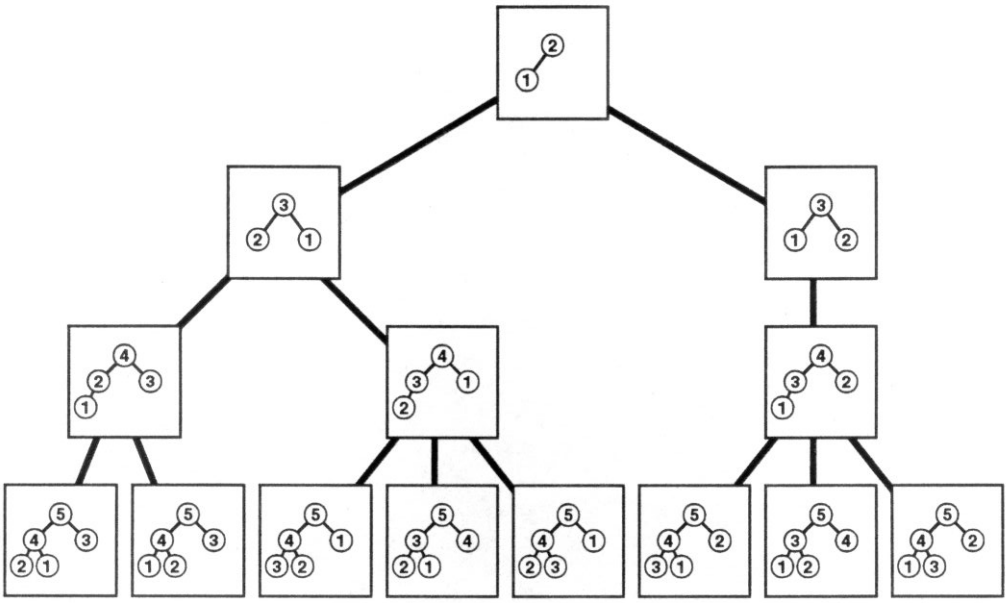

Figure 2. Generating Heaps.

Pulldown Sequence	1234	1232	1224	1223	1222	1124	1123	1122
Cost Sequence	0112	0111	0112	0111	0111	0012	0011	0011
Cost	4	3	4	3	3	3	2	2

Table 2. Cost Sequences ($N = 5$)

It is a straightforward matter to keep track of cost sequences and costs as heaps are generated. The full distribution of the number of moves required to sort-down the more than twenty-five million heaps of 15 keys or fewer, computed using this method, is given in the Appendix. The average-case costs computed from this table are given in Table 3. The average seems to drift towards the worst case, but the numbers are too small to make reasonable conjectures.

N	4	5	6	7	8	9	10	11	12	13	14	15
Average Cost	1.67	3.00	4.40	5.95	8.13	10.31	12.62	14.97	17.43	19.91	22.44	24.98

Table 3. Average Sort-Down Costs

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 4 shows the cost of (number of moves required to sort down) the best heaps for $N < 26$. On the one hand, 26 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	≤ 41

Table 4. Best-Case Costs

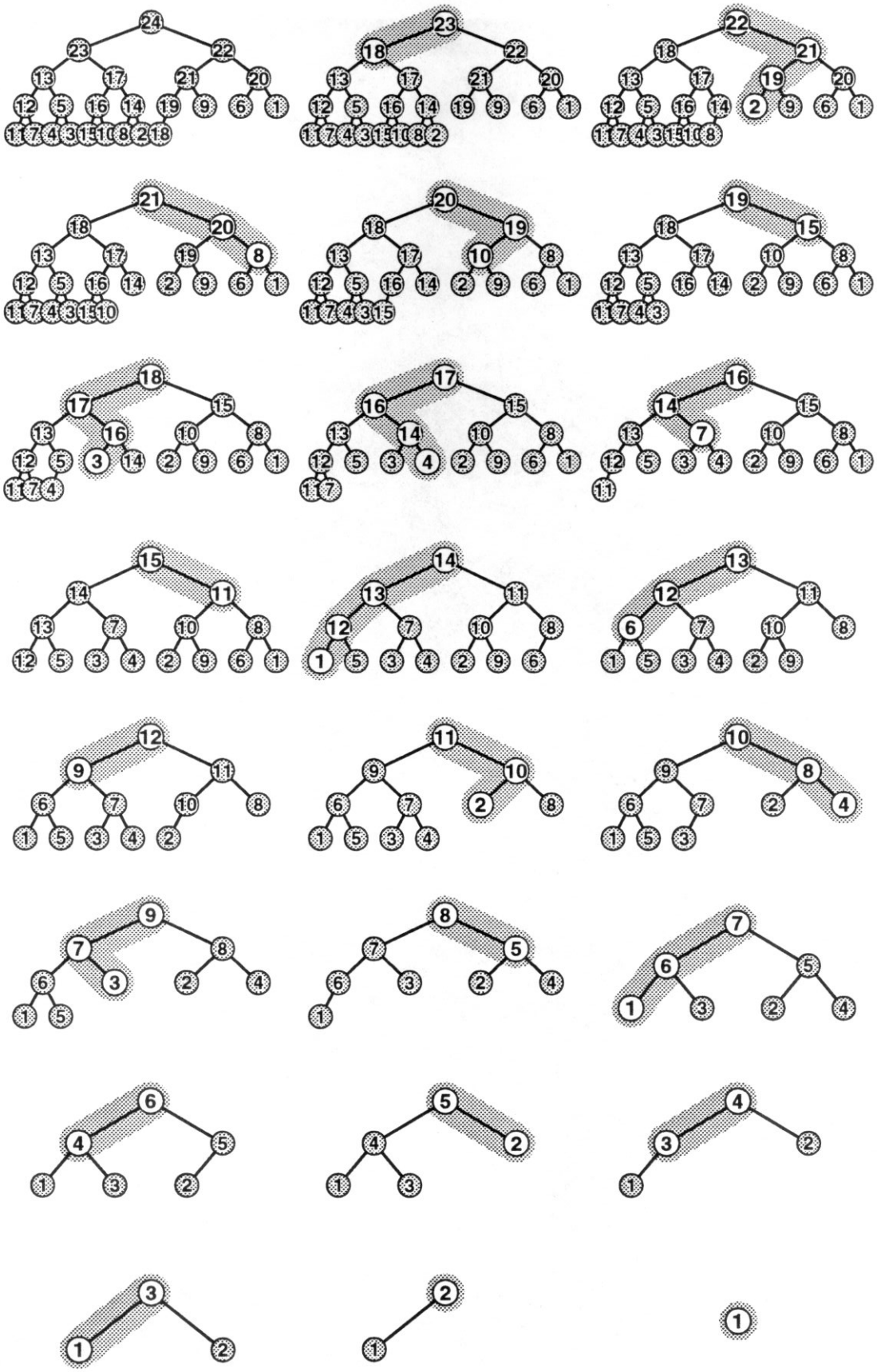

Figure 3. A best-case heap.

Furthermore, a careful examination of how a known best case heap operates gives some intuition on how to construct larger best-case heaps. Figure 3 shows the sortdown process for a best heap of size 24: the cost sequence for this heap is 001111212222123123312231 for a total cost of 37. Studying this example gives some insight into how a low cost might be achieved over several pulldowns by alternating groups of short paths with groups of long paths, for an average cost of about half the heap height. Specifically, consider the six pulldowns that grow the heap from 16 to 22. Three "long" pulldowns (of 7, 4, and 3) ensure that the keys involved in the next few pulldowns are among the largest in the heap, in particular the keys high up on the other side of the root (15, 10, and 8). 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, 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 [3]: consider 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. This argument was independently developed and presented by Wegener [11]. We have:

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

Proof: Given a heap on $N = 2^n - 1$ keys, consider the cost of the first half of the sort-down process, during which the largest 2^{n-1} keys are removed. Initally, these keys form a heap-ordered subtree of the heap. The positions in this subtree that are at the bottom level of the heap cease to exist during the first half of the sort-down process, while the higher positions are filled with smaller keys. The number of moves required for a small key to displace a large key from a high position is the distance from that position to the root. It follows that the number of data movements required in the first half of the sort-down process is at least the internal path length of the subtree composed of the largest 2^{n-1} keys, excluding those at level n.

At most 2^{n-2} large keys can be at level n, leaving at least 2^{n-2} large keys in the top $n-1$ levels. The internal path length of the subtree formed by the large keys in the top $n-1$ levels is thus at least that of the subtree formed by the top 2^{n-2} keys of the heap. The subtree formed by these top 2^{n-2} keys has an average internal path length greater than $n-3$ providing a lower bound of $(2^{n-2}(n-3) + 2^{n-2}(0))/2^{n-1} > n/2 - 2$ on the average cost of the first 2^{n-1} siftdown operations.

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

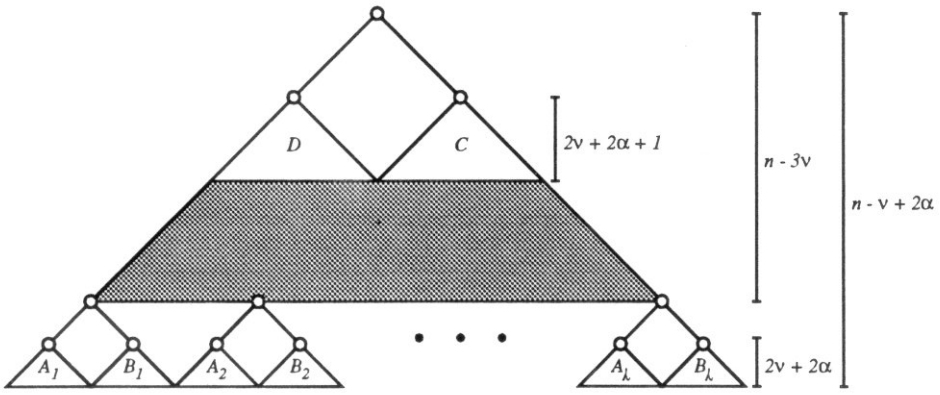

Figure 4. Structure for best case heap construction.

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 this way using only $\sim 1/2N \lg N$ moves. This construction maintains a heap of the form depicted in Figure 4, starting with a "seed" heap on $O(N/\lg N)$ keys and growing to a heap on $N = 2^n - 1$ keys. We alternately label A_i and B_i the subheaps whose roots lie in a certain fixed level of the heap being constructed. The A_i and B_i will be the source of keys to pull down from the bottom of the heap. To provide keys to pull down from the top of the heap, we mark off the top few layers of the left and right halves of the heap and call them D and C. These subheaps all grow during the construction process, and D and C each will always contain at least as many keys as any of the A_i or B_i.

Our construction is based on imposing and maintaining a set of constraints on the distribution of values in the heap under construction beyond the defining requirement that no key be larger than its parent. Specifically, we initially require that for all i, every key in A_i be larger than any key in B_i, and that every key in C be larger than any key of A_2 or B_2. Starting with any heap satisying these constraints, we can build a larger one that still satisfies the constraints using only pulldown operations, as follows. We first pull down the keys of B_1 by keys of A_1. As we perform these pulldowns, new keys flow from the root of the heap into B_1. This guarantees that all keys of B_1 are new to the heap (except those that were on the path from the root of the heap to the root of B_1); these new keys are larger than the keys in C so we pull down the keys in C by the keys of B_1. Working across the bottom of the heap, we alternate sets of pulldowns using the A's to pulldown the B's at the bottom with sets of pulldowns using the B's to pulldown C at the top (or D when B_i is on the right side of the heap). This process increases the height of the heap by 1.

Ideally, we would hope that this process, made possible by the imposed constraints, would also maintain the constraints, so that it could be iterated to form an arbitrarily large heap. Actually, the process leaves a heap with similar constraints, but with keys in B's larger than keys in A's. But we can define a similar process involving B's pulling down A's that brings us back to a heap satisfying the original constraints, but with height increased by 2. This "two-phase" process can be iterated to form an arbitrarily large heap.

The whole construction process revolves around alternating pulldowns from the bottom (A's pulling down B's or B's pulling down A's) with pulldowns from the top (B's or A's pulling down C's or D's) which gives an average pulldown length of about half the heap height. While conceptually straightforward, the full proof below is technically rather complex because of a number of special situations that arise during the process.

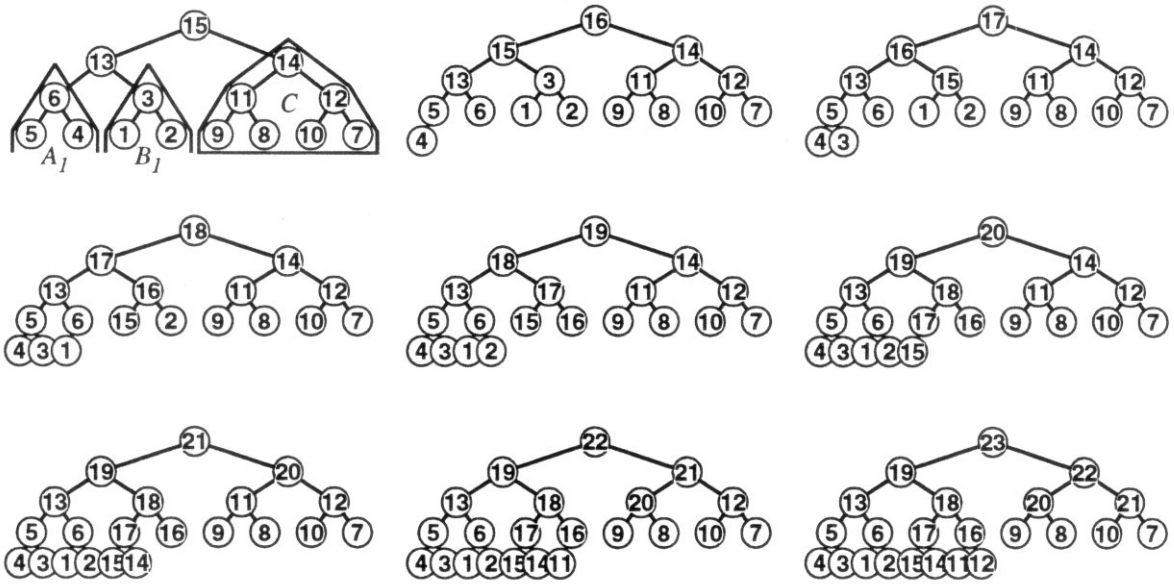

Figure 5. Numerical illustration of the construction.

Figure 5 provides a numerical illustration of the construction that is intended to illustrate some of the technical difficulties encountered when proving the correctness of the construction. This example is somewhat similar to the process already noted in Figure 3. After pulling down 4, the keys of B_1 (3, 1, and 2) are pulled down into A_1. After pulling down 15, all keys in the top two levels of B_1 have entered the heap since the first frame. The largest keys of C (14, 11, and 12) can thus be pulled down to add a new row to B_1. First, note that in the illustration, C overlaps what should be the roots of A_2 and B_2: in the actual construction, this would disturb A_2 and B_2 and cannot be permitted; the illustration is shown as it is only because a heap without overlap would be too large to draw. Second, note that there are not quite enough keys in B_1 to complete the new bottom row of A_1, so an initial pulldown of 4 from A_1 was required: such initial pulldowns must be specifically accounted for in our construction. Third, as was noted earlier, we cannot always count on the keys entering B_1 to be larger than those in C since some of them come from the path from the root of the heap to the root of B_1; any such keys should be pulled down themselves before beginning to pull down keys of C (in figure 5 we pulldown 15): in this instance this was actually not required, but if the path from the root of the heap to the root of B_1 were longer, it is easy to see how B_1 could contain keys that are smaller than those in C. Fourth, note that in the final frame of figure 5, the key 13 of A_1 is larger than a few keys of B_1, due to the initial pulldown of 4: when future passes pull down keys of some B_i by A_i, there will be a limited number of exceptions to the rule that keys of A_i are larger than those of B_i which must be accounted for in the actual construction.

Let $N = 2^n - 1$ be the number of keys in the heap to be constructed. Table 5 is a list of various notations that play a role in the construction. Though these quantities are functions of N (and each other, in some cases), we abbreviate them all with single letters for brevity. Further details on these quantities and motivation for the specific values assigned are given below: they are collected and summarized here for convenience in referring to their values and meanings. The construction starts with a heap of height $n - \nu$ and with the A_i and B_i of height 2ν. That is, A_i is the subheap rooted at $\mathtt{a}[2^{n-3\nu} + 2i - 2]$, and B_i is the subheap rooted at at $\mathtt{a}[2^{n-3\nu} + 2i - 1]$. As the construction progresses, the number of A and B heaps will always be precisely $\ell = 2^{n-3\nu-1}$ (their roots will remain fixed), and by the end of the construction, these subheaps will be of height 3ν and the main heap of height n.

Notation	Upper Bound	*Quantity Represented*
$n = \lceil \lg N \rceil$	$O(\log N)$	height of the finished heap
$\nu = 2\lceil \frac{1}{2} \lg n \rceil$	$O(\log \log N)$	height of A, B, and C subheaps (within a constant factor)
$\ell = 2^{n-3\nu-1}$	$O(N/\log N)$	number of$A$ and B subheaps (depends only on N)
$\alpha \in [0, \nu/2)$	$O(\log \log N)$	index tracking the "phases" used to build the heap
$\epsilon = 2n\alpha$	$O(\log N \log \log N)$	cumulative count of keys violating the primary invariants

Table 5. Notation for best-case construction

The heap will grow from the initial seed to the final heap in a series of $\nu/2$ passes, each consisting of two phases. The AB phase adds a level to the heap by pulling down the keys of the B_i into A_i as demonstrated in figure 5. This leaves the keys of the B_i larger than those of A_{i+1}, permitting phase BA to add a level to the heap by pulling down the keys of A_{i+1} into B_i. This restores the original condition that the keys of A_i are larger than those of B_i, so a new pass can begin.

Referring to Figure 4, note that the C and D subheaps grow in size with the A's and B's (they are one level higher at the beginning of the first part of each phase in order to accomodate the second part, after the A's and B's have already grown by one level). To ensure that C and D will never overlap the A_i and B_i, as mentioned above, we choose N large enough so that $6\nu \leq n$.

To describe relationships among sets of keys in the heap structure given in Figure 4, 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 a subheap H and a positive integer x, we define $L(H, x)$ (the large keys) to be all but the smallest x keys of H, and $S(H, x)$ (the small keys) to be all but the largest x keys of H. Thus, if we wished to say that all but one of the keys in A_1 are smaller than any key in B_1 (as is the case in the final frame of figure 5), we simply write $S(A_1, 1) < B_1$.

Theorem 2. *The best case of Heapsort requires no more than* $\frac{1}{2}N \lg N + O(N \log\log N)$ *data moves.*

Proof: Figure 6 gives the steps that build a best case heap from a seed heap of the type shown in Figure 4. Also given in Figure 6 are upper bounds on the number of data moves made by each step. This uses the notation given above, and the reader may find it convenient to refer to Figure 4. As mentioned above, take N large enough so that $6\nu \leq n$ to ensure that the structure does not degenerate.

	Cost Bound
1. Build seed heap from the integers from 1 to $2^{n-\nu} - 1$.	
2. for α from 0 to $\nu/2$ do	
Phase AB:	
(i) for i from 1 to $\ell/2$ do	
(a) Pull down the $\epsilon + 1$ smallest keys in A_i.	$n(2n\alpha + 1)$
(b) Pull down all but the largest ϵ keys in B_i.	$n2^{2(\nu+\alpha)}$
(c) Pull down each of the smallest $\epsilon + n$ keys in B_i.	$n(2n\alpha + n)$
(d) Pull down the largest $2^{2(\nu+\alpha)} - \epsilon - n$ keys in C.	$(2(\nu+\alpha)+1)2^{2(\nu+\alpha)}$
(ii) for i from $\ell/2 + 1$ to ℓ do as in (i), replacing C with D.	
Phase BA:	
(i) Pull down the smallest key in A_1, $2^{2(\nu+\alpha)+1}$ times.	$n2^{2(\nu+\alpha)+1}$
(ii) for i from 1 to $\ell/2 - 1$ do	
(a) Pull down the smallest $\epsilon + n + 1$ keys in B_i.	$n(2n\alpha + n + 1)$
(b) Pull down all but the largest $\epsilon + n$ keys in A_{i+1}.	$n2^{2(\nu+\alpha)+1}$
(c) Pull down each of the smallest $\epsilon + 2n$ keys in A_{i+1}.	$n(2n\alpha + 2n)$
(d) Pull down the largest $2^{2(\nu+\alpha)+1} - \epsilon - 2n$ keys in C.	$(2(\nu+\alpha)+1)2^{2(\nu+\alpha)+1}$
(iii) for i from $\ell/2$ to $\ell - 1$ do as in (ii), replacing C with D.	
(iv)	
(a) Pull down the smallest $\epsilon + n + 1$ keys in B_ℓ.	$n(2n\alpha + n + 1)$
(b) Pull down the smallest $2^{2(\nu+\alpha)+1} - \epsilon - n - 1$ keys of A_1.	$n2^{2(\nu+\alpha)+1}$

Figure 6. Best-case heap construction

Figure 7 gives the "pass invariants" that formalize the structure shown in Figure 4. They are required to hold in the seed heap and at the end of each pass. To prove the correctness of the construction sketched above and made explicit in Figure 6 we must prove three things. First, we must show that it is possible to create a seed heap satisfying the pass invariants with $\alpha = 0$. Second, we must show that a single pass of the construction is both permitted by and preserves the pass invariants. Third, we must sum the cost bounds given in Figure 6 to show that the number of data moves is $\sim \frac{1}{2}N \lg N$ as claimed. Together these parts form a proof by induction that the construction we have described is possible and satisfies the required bound on the number of data moves. The most complex part of the proof is the correctness—many readers may wish to skip to the section on summing the cost bounds below.

(i) $L(A_i, \epsilon) > S(B_i, \epsilon)$ for all i, $1 \leq i \leq \ell$,
(ii) $L(C, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > A_2$,
(iii) $L(C, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > B_2$.

Figure 7. Pass invariants for best-case construction

I. Constructing the Seed Heap:

Start with $2^{n-\nu} - 1$ keys — the integers from 1 to $2^{n-\nu} - 1$. Fill the A_i with the smallest $\ell(2^{2\nu} - 1)$ keys and arrange the keys in each A_i in heap order. Fill the B_i with the next block of $\ell(2^{2\nu} - 1)$ keys, again in heap order. Since the keys in the B_i are larger than those in the A_i, pass invariant (i) is satisfied. The largest $2\ell - 1$ keys remain and should be placed in heap order in the top $n - 3\nu$ levels of the heap. Since C is contained in these top levels, it follows that pass invariants (ii) and (iii) are satisfied.

II. Correctness of the Inductive Step:

In this section we show that if we start with a heap that satisfies the pass invariants and execute a pass of the construction as described in Figure 6, the resulting heap also satisfies the pass invariants and contains two more levels than at the beginning of the pass. We begin by showing, step by step, that phase AB of a pass can be executed on a heap that satisfies the pass invariants.

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. At the beginning of the i'th iteration we require $L(A_i, \epsilon) > S(B_i, \epsilon)$, $L(C, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > A_{i+1}$ and $L(C, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > B_{i+1}$. These hold at the start of the first iteration by the pass invariants.

(a) We begin level $2(\nu + \alpha) + 1$ of A_i by pulling down the $\epsilon + 1$ smallest keys in A_i. Every key at level $2(\nu + \alpha)$ of A_i is now greater than any key of $S(B_i, \epsilon)$. This follows from three observations. First, we started with $L(A_i, \epsilon) > S(B_i, \epsilon)$. Second, all of the ϵ keys of A_i which were smaller than some key of $S(B_i, \epsilon)$ have been pulled to level $2(\nu + \alpha) + 1$. Third, any key that enters A_i in this step is greater than all keys of the original $L(A_i, \epsilon)$ since these were once among its descendants.

(b) Since every key of $S(B_i, \epsilon)$ is smaller than any key at level $2(\nu + \alpha)$ of A_i, we can pull down $2^{2(\nu+\alpha)} - 1 - \epsilon$ keys of B_i to complete level $2(\epsilon + \alpha) + 1$ of A_i.

(c) Pulling down the smallest $\epsilon + n$ keys in B_i ensures that all keys at or above level $2(\nu + \alpha)$ of B_i are new to the heap since the beginning of step (a). This results from pulling down the t keys originally in B_i that failed to be pulled down in step (b) as well as the fewer than n 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).

(d) Since all keys at or above level $2(\nu + \alpha)$ 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), it follows that all keys at or above level $2(\nu + \alpha)$ of B_i are larger than any key of C. We can thus pull down the largest $2^{2(\nu+\alpha)} - \epsilon - n$ keys in C to complete level $2(\nu + \alpha) + 1$ of B_i. In the process, the largest $2^{2(\nu+\alpha)} - \epsilon - n$ keys in C are replaced with keys that are new to the heap. These new keys are larger than any key in A_{i+2} or B_{i+2}. Note that A_{i+1}, A_{i+2}, B_{i+1}, and B_{i+2} have not been touched since the beginning of phase AB. This leaves $L(C, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > A_{i+2}$, $L(C, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > B_{i+2}$, and $L(A_{i+1}, \epsilon) > S(B_{i+1}, \epsilon)$ for the next ($i + 1$'st) iteration of the loop, as required.

(ii) This loop adds keys to the bottom of the right side of the heap just as (i) added them to the left. Note that in step (i), D was flushed and refilled with keys that are new to the heap since the beginning of this phase. Since none of the A_i or B_i for $i > \ell/2$ have been touched since the beginning of the phase we know that $L(A_i, \epsilon) > S(B_i, \epsilon)$, $L(D, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > A_{\ell/2+2}$, and $L(D, 2^{2(\nu+\alpha)} - 1 + \epsilon + n) > B_{\ell/2+2}$ hold. (ii) thus proceeds exactly as (i) with C replaced by D.

So far we have shown that it is possible to use Phase AB of the construction to add a level to a heap satisfying the pass invariants. We now show that at the end of Phase AB, we are left with conditions similar to the pass invariants. These conditions permit Phase BA to add another level to the heap and restore the pass invariants by pulling down keys of the A_{i+1} by keys of B_i. Fix i, $1 \leq i < \ell$. We wish to show $L(B_i, \epsilon + n) > S(A_{i+1}, \epsilon + n)$.

Of the keys that are in A_{i+1} at the end of Phase AB, fewer than $\epsilon + n$ were not in A_{i+1} or B_{i+1} at the beginning of the phase. This follows from the observation that A_{i+1} contains all $2^{2(\nu+\alpha)} - 1$ keys that it contained at the beginning of the phase and all but ϵ of the keys that B_{i+1} contained at the beginning of the phase.

Now consider the origins of the keys in B_i. $2^{2(\nu+\alpha)} - \epsilon - n$ keys were pulled down from C (or D); by the inequalities that held prior to the i'th iteration of loop (i) (or the $i - \ell/2$'th iteration of loop (ii)), all of these keys are larger than any that were in A_{i+1} or B_{i+1} at the beginning of the phase. Even larger are the $2^{2(\nu+\alpha)} - 1$ keys that were in the top $2(\nu + \alpha)$ levels of B_i before we began pulling down the keys from C (or D). Together these constitute at least $2^{2(\nu+\alpha)+1} - \epsilon - n - 1$ keys in B_i that are larger than any key that was in A_{i+1} or B_{i+1} at the beginning of the phase. It follows that $L(B_i, \epsilon + n) > S(A_{i+1}, \epsilon + n)$. Similar reasoning shows that $L(B_\ell, \epsilon + n) > S(A_1, \epsilon + n)$.

Finally, note that A_3 and B_2 were undisturbed during step (ii) while C was flushed and refilled with keys that had not been in the heap prior to Phase AB; this leaves the keys of C larger than those in A_3 and B_2. We thus have the following situation, analogous to the pass invariants, as we begin Phase BA:

(i) $L(B_i, \epsilon + n) > S(A_{i+1}, \epsilon + n)$ for $i < \ell$.
(ii) $L(B_\ell, \epsilon + n) > S(A_1, \epsilon + n)$.
(iii) $L(C, \epsilon + 2n - 1) > A_3$.
(iv) $L(C, \epsilon + 2n - 1) > B_2$.

We now show, step by step, that these conditions suffice to ensure that Phase BA can add a level to the heap.

Correctness Proof for Phase BA:

(i) We add level $2(\nu + \alpha) + 2$ to A_1 by pulling down a key from within A_1 since there is no B_0 into which to pull down the keys of A_1.

(ii) This loop adds level $2(\nu + \alpha) + 2$ to all A_i and B_i on the left half of the heap with the sole exception of $B_{\ell/2}$. The steps of this loop serve the same purposes, and can be executed for the same reasons, as the corresponding steps of loop (i) of Phase AB. By the reasoning used before, the inequalities $L(B_i, \epsilon + n) > S(A_{i+1}, \epsilon + n)$, $L(C, \epsilon + 2n - 1) > A_{i+2}$ and $L(C, \epsilon + 2n - 1) > B_{i+1}$ hold at the beginning of the i'th iteration of this loop.

(iii) This loop acts in the same way as (ii), adding level $2(\nu + \alpha) + 2$ to all A_i and B_i on the right side of the heap with the exception of B_ℓ.

(iv) This step serves the dual purpose of filling in level $2(\nu + \alpha) + 2$ of B_ℓ and of purging the small keys from A_1 that would have been removed in step (i) if there had been a place to put them.

- (a) Pulling down the smallest $\epsilon + n + 1$ keys in B_ℓ has the same effect as Phase BA, step (ii)a.; every key at level $2(\nu + \alpha) + 1$ of B_ℓ is now greater than all but at most $\epsilon + n$ keys that were in A_1 prior to beginning Phase BA.
- (b) We complete level $2(\nu + \alpha) + 2$ of B_ℓ by pulling down the smallest $2^{2(\nu+\alpha)+1} - 1 - \epsilon - n$ keys of A_1.

We have now shown that a complete pass of the construction adds two levels to a heap satisfying the pass invariants given in Figure 7. It remains to show that the pass invariants have been restored at the end of the pass so that the process may be iterated.

First, for $i > 1$ and α unincremented, $L(A_i, \epsilon + 2n) > S(B_i, \epsilon + 2n)$ holds at the end of Phase BA by the same reasoning used to establish the analogous result for B_{i-1} and A_i at the end of Phase AB; as before, the keys in $S(B_i, \epsilon + 2n)$ were in B_i and A_{i+1} at the beginning of the phase while those in $L(A_i, \epsilon + 2n)$ are either new to the heap or come from C or D, whose keys are larger than those in $S(B_i, \epsilon + 2n)$.

Now consider the special case of $i = 1$. As above, the keys in $S(B_1, \epsilon + 2n)$ at the end of Phase BA were in B_1 and A_2 at the beginning of the phase. In A_1 on the other hand, with the exception of the keys that were between the root of the heap and the root of A_1 at the beginning of Phase BA and the smallest $\epsilon + n$ keys not pulled down in step (iv), a total of fewer than $\epsilon + 2n$ keys, every key in A_1 is new to the heap since the beginning of the phase. It follows that $L(A_1, \epsilon + 2n) > S(B_1, \epsilon + 2n)$, so part (i) of the pass invariants holds for all i.

Parts (ii) and (iii) of the pass invariants continue hold by the reasoning used before, that since the beginning of Phase BA, every key of C has been changed while A_2 and B_2 have remained untouched. Since the pass invariants hold when we increment α, we can iterate the construction a total of $\nu/2$ times to create a heap on N nodes.

III. Summing the Cost Bounds:

We now compute an upper bound on the number of key moves required to complete the construction. We proceed by summing the bounds given in Figure 6 on the number of moves performed in individual steps.

First, consider all the pulldowns that are included for the purpose of putting large keys in a path to the root (those involving about ϵ keys). This includes steps AB$(i)(a)$, AB$(i)(c)$, AB$(ii)(a)$, AB$(ii)(c)$, BA$(ii)(a)$, BA$(ii)(c)$, BA$(iii)(a)$, BA$(iii)(c)$, BA$(iv)(a)$. The cost bound for each of these steps during pass α is less than $n^2(2\alpha + 2)$ so the total work for all of these steps during the whole construction process is bounded above by

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

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

Most of the work in the construction process involves using the A_i subheaps to pull down the B_i subheaps in Phase AB and vice-versa in Phase BA. This includes steps AB(*i*)(*b*), AB(*ii*)(*b*), BA(*ii*)(*b*), BA(*iii*)(*b*), and BA(*iv*)(*b*). The total work for all of these steps is bounded above by:

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

Roughly, this acounts for about half of the pulldowns during the construction process, down at the bottom of the heap.

The other half of the pulldowns are those involving C and D, high up in the heap. The total work performed in steps AB(*i*)(*d*), AB(*ii*)(*d*), BA(*ii*)(*d*) and BA(*iii*)(*d*) is bounded by

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

and the total work performed in BA(*i*)( ) is bounded above by:

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

So far we have counted the costs of the $\nu/2$ construction passes, or equivalently, the cost of sorting down the final heap to the seed heap of height $n - \nu$. To this we must add the cost of sorting down the seed heap. As noted in Section 1, this number is bounded above by

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

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. The Average Case

Is the coefficient of $N \lg N$ in the expression for the average number of moves required to sort down a random heap of N keys 1, or not? The following result indicates that perhaps it is:

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

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 key less than `a[N div 2]`. This key

is at the root of a complete subheap of keys all less than `a[N div 2]`: the average level of these keys 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 [2].) 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. Carlsson has analyzed the average case of Heapsort under the assumption that succeeding sortdowns are not expected to be any worse than the first one [1]. This leads to stronger results than those presented below, but rests upon an unproved assumption. We obtain rigorous results by observing that almost all heaps have the property that nearly all keys sort down to the bottom. We can thus prove our main result by a surprisingly short counting argument:

Theorem 3. *The average number of data moves required to Heapsort a random permutation of N distinct keys is $\sim N \lg N$.*

Proof: The cost of constructing the heap is $\Theta(N)$ and we will complete the proof by establishing that at least $N \lg N - N \lg \lg N - 4N$ moves are required for the sort-down process.

We derive an upper bound on the number of heaps that use a small number of moves, using the cost and pull-down sequences defined in Section 3. Consider a cost sequence $L = l_1 l_2 \ldots l_N$. The total number of distinct corresponding pull-down sequences is bounded above by

$$2^{l_1} \cdot 2^{l_2} \cdot \ldots \cdot 2^{l_N} = 2^{\sum_i l_i}.$$

Recall that $\sum_i l_i$ is precisely the sort-down cost for all heaps built using any of these pull-down sequences. Since the root cannot be pulled down, there are at most $\lg N$ values possible for each l_i, so there are fewer than $(\lg N)^N$ possible values for L. Thus the number of distinct pulldown sequences that require exactly M data movements is bounded above by $(\lg N)^N 2^M$. It follows that there are at most $(\lg N)^N 2^M$ heaps requiring fewer than M data movements to sort.

Set $M = N(\lg N - \lg \lg N - 4)$. From the preceding paragraph, the number of heaps that require fewer than M data movements to sort is bounded above by $(N/16)^N$. The number of data movements required by heapsort in the average case is thus bounded below by:

$$\frac{(f(N) - (N/16)^N)M}{f(N)} = M - N\frac{(N/16)^N}{f(N)}(\lg N - \lg \lg N - 4)$$

where $f(N)$ is the number of heaps on N keys. The theorem will be proved if we can show that $f(N)$ exceeds $(N/16)^N$ by an exponential factor. The Lemma in section 2 implies this for N of the form $2^n - 1$, and the proof given there can be modified to give the result for general N. Alternatively, Munro has pointed out that such a bound results from the observation that at most $2N$ key comparisons are made in building a heap from a permutation using the bottom-up process in Program 1, from which it follows that $f(N) > N!/2^{2N} > (N/4e)^N$, which is exponentially greater than $(N/16)^N$, so the theorem is proved. If N is of the form $2^n - 1$, the constant 4 in this proof can be replaced by, in the notation of section 2, any constant larger than $\lg(4e^{1-\alpha}) \approx 2.80713-$ ∎

6. 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 chidren of the current node, the other to determine whether the current node is larger than both its children (so the loop should be exited). Floyd (see [8]) 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 key being sifted down is found. Most of the time, only a few steps back up are required.

The average-case results of the previous section show that Floyd's method is asymptotically optimal on the average, but the best-case results for Heapsort of section 4 translate into *worst-case* results for this variant. 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. Gonnet and Munro [7] have shown that $\sim N \lg N$ can be achieved, at the expense of a more

intricate algorithm, by moving back up the heap with a careful binary search instead of the linear search of Floyd's method. Further results in this direction have been obtained by Wegener [12].

It is possible to develop a construction similar to the one given in Section 4 to count the number of executions of the statement `j:=j+1`. The coefficient of $N \lg N$ for this quantity turns out to be $1/4$ in the best case and $3/4$ in the worst case [9], and these bounds are achieved in heaps having asymptotically the best and worst case numbers of data moves. It also can be shown that this quantity is $\sim \frac{1}{2} N \lg N$ in the average case, using an argument similar to the proof of Theorem 3[9]. This question is not of as much practical interest as knowing the number of data moves because this cost is typically on the order of one-tenth the total cost, but these results allow computation of the coeffient of the leading term of the running time of Heapsort in the best, worst, and average cases for typical implementations, thus solving open problems left by Knuth.

Heapsort is prototypical of algorithms which are designed to achieve good worst-case performance but for which one would also like to know average-case performance. Few such algorithms "preserve randomness" and are thus very difficult to analyze using standard probabilistic techniques. But the simplicity of the counting argument given in this paper for the average case of Heapsort suggests that it is worthwhile to consider applying such techniques to the problem of determining average-case performance of other important algorithms with near-optimal worst-case performance.

Acknowledgements

We thank Ian Munro for his helpful comments and observations.

References

1. S. CARLSSON. "Average-Case Results on Heapsort," *BIT* **27**, (1987).
2. E.-E. DOBERKAT. "Deleting the root of a heap," *Acta Informatica* **17**, 3 (1982).
3. H. ERKIÖ. "On Heapsort and its dependence on input data," Technical Report No. A-1979-1, Dept. of Computer Science, University of Helsinki, Finland.
4. R. FLEISCHER. "A Tight Lower Bound for the Worst Case of Bottom-Up-Heapsort," Technical Report No. MPI-I-91-104, Max-Planck-Institut für Informatik, W-6600 Saarbrücken, Germany.
5. R. FLEISCHER, B. SINHA, AND C. UHRIG. "A Lower Bound for the Worst Case of Bottom-Up-Heapsort," Technical Report No. A23/90, University of Saarbrücken, Germany.
6. R. W. FLOYD. "Treesort 3: Algorithm 245," *Comm. of the ACM* **7**, 12 (1964).
7. G. H. GONNET AND I. MUNRO. "Heaps on heaps," *SIAM J. on Computing* **15**, 4(1986).
8. D. E. KNUTH. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass. (1973).
9. R. SCHAFFER. Manuscript in preparation.
10. R. SEDGEWICK. *Algorithms 2nd edition*, Addison-Wesley, 1988.
11. I. WEGENER. "Bottom-Up-Heapsort, a New Variant of Heap Sort Beating on Average Quick Sort (if n not very small)," *MFCS '90*, Lecture Notes in Computer Science **452**, Springer Verlag.
12. I. WEGENER. "The Worst Case Complexity of McDiarmid and Reed's Variant of Bottom-Up-Heapsort is Less Than $n \log n + 1.1n$," *STACS '91*, Springer Verlag.
13. 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 5. Though few simply expressed relationships among these numbers are evident, the underlying distribution seems to be rather stable.

	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 5. Full Distribution of Sorting-Down Costs