ANALYSIS OF HEAPSORT

Russel Warren Schaffer

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

COMPUTER SCIENCE

June 1992

Acknowledgments

Professor Robert Sedgewick has been everything I could have hoped for in an advisor. He not only suggested Heapsort as a problem to study, he also supplied a point from which to begin work. His encouragement and advice have been very important to the development of this thesis; it is hard to thank him enough.

I must also thank Professor Ian Munro for taking an early interest in this work, making suggestions that led to strengthened results, and finally for reading the finished product. His presence at Princeton this year has been most fortunate.

For reading the thesis I also thank Professor Robert Tarjan.

The secretaries in the Department have been amazingly helpful during my few years here. In particular, I thank Sharon Rodgers, as well as Melissa Lawson and Ginny Hogan.

My thanks also go to my officemates Ayellet Tal, Liang-Fang Chao, and Michael Golan, and to virtual officemate Shlomo Gortler, and to other students, professors, and members of the Department who have helped to make my past four years memorable and productive.

Finally, I thank my parents who have been supportive since long before I came to Princeton.

This material is based upon work supported under a National Science Foundation Graduate Fellowship.

ANALYSIS OF HEAPSORT

Abstract

Author: Russel Warren Schaffer

Advisor: Professor Robert Sedgewick

Heapsort is a classical sorting algorithm due to Williams. Given an array to sort, Heapsort first transforms the keys of the array into a heap. The heap is then sorted by repeatedly swapping the root of the heap with the last key in the bottom row, and then sifting this new root down to an appropriate position to restore heap order.

In Williams's original Heapsort, the new root is sifted down by repeatedly comparing its two children, and swapping it with its larger child if a comparison shows the child to be larger than the key being sifted. Floyd proposed an important variant of Williams's Heapsort which unconditionally performs the swap with the larger child.

This thesis analyzes the asymptotic number of executions of each instruction for both versions of Heapsort in the average, best, and worst cases.

In the average case, when sorting a uniformly generated random heap on N distinct keys, Williams's Heapsort performs $\sim 2N \lg N$ key comparisons while Floyd's variant performs $\sim N \lg N$ key comparisons (lg is the logarithm base two). Another quantity of interest is the number of times keys are swapped with their right children. Both Williams's and Floyd's versions of Heapsort expect to perform $\sim \frac{1}{2} N \lg N$ such swaps.

Sedgewick, and independently Fleischer and Wegener, have presented arguments to demonstrate that the number of key comparisons required by Williams's Heapsort in the best case and Floyd's Heapsort in the worst case are $\sim N \lg N$ and $\sim \frac{3}{2} N \lg N$ respectively. These arguments are extended and applied in a different form to demonstrate that in the worst case, Williams's and Floyd's Heapsorts perform $\sim \frac{3}{4} N \lg N$ swaps of keys with their right children, while in the best case at most $\sim \frac{1}{4} N \lg N$ such swaps are performed. For both versions of Heapsort, it is shown that these best and worst case numbers can be found in heaps that also require the best and worst case numbers of key comparisons.

Contents

Chapter 1

Introduction

Sorting is one of the basic problems of Computer Science, and Williams's Heapsort [28] is one of the basic sorting algorithms. It is thus surprising that the analysis of Heapsort has long remained incomplete. This thesis determines, in the best, worst, and average cases, the asymptotic number of executions of Heapsort's most frequently executed instructions. These results are combined to permit calculation of the constant factors multiplying the leading terms in the total running times of Heapsort in the best, worst, and average cases. Similar bounds are also developed for an important variant of Heapsort due to Floyd ([18] Ex. 5.2.3.18).

Heapsort sorts an array $\mathtt{A}[1 \ldots N]$ of N keys by selecting the largest key in A and exchanging it with the last key in A. The second largest key is then extracted from among the remaining keys and swapped with the key in the penultimate position of A. Continuing in this manner, it is clear that eventually the array A will contain, in ascending order, the keys it contained originally.

The algorithm just described can be implemented naively, extracting each maximum key by examining all unsorted keys. However, the running time of the resulting Selection Sort is $\Omega(N^2)$. Heapsort achieves a guaranteed running time of $O(N \log N)$ by storing the unsorted keys in a heap; that is in a tree in which every node contains a key which is at least as large as the keys in its children. In the particular versions of Heapsort to

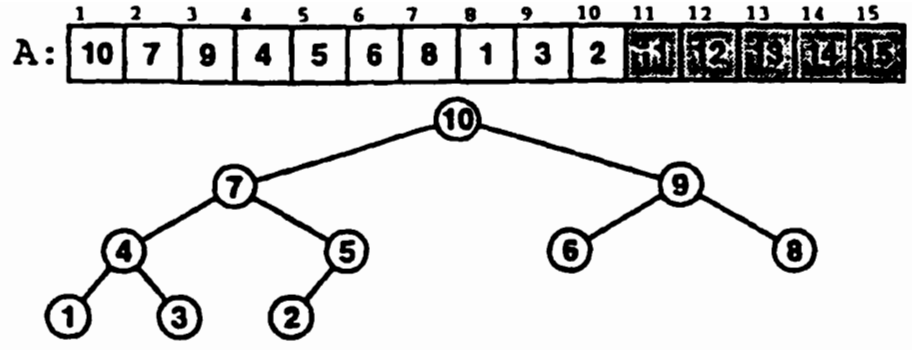

Figure 1.1: A partially Heapsorted array.

be considered here, the heap is a complete binary tree stored in implicit level order in the first positions of A. This means that a key in position i of A has as its parent the key in position $\lfloor i/2 \rfloor$. Heap order is maintained by requiring that $A[\lfloor i/2 \rfloor] \geq A[i]$ for all positions $i > 1$ containing unsorted keys.

Figure 1.1 shows a partially sorted array A containing 15 keys. The ten smallest keys are stored as a heap in the first ten positions of the array while the shaded five positions at the end of the array contain keys that have already been sorted into place. Below the array is an explicit diagram of the heap structure that is implicit in the unsorted portion of the array.

The $O(N \log N)$ bound on the running time of Heapsort is obtained by noting that the largest unsorted key is always at the root of the heap, and thus is extracted in constant time from A[1]. This largest unsorted key is moved to the position it should occupy in the sorted list by exchanging it with the last key from the bottom of the heap. Unfortunately, this key from the bottom of the heap is probably too small to belong at the top of the heap, so it violates the heap ordering on the unsorted keys. Fortunately, the heap can be restored by moving the misplaced key, call it a, from A[1] to an appropriate position in the heap. This position is found by repeatedly swapping a

```
1.   void Williams() {
2.      long i, k;
3.      key a, t;

4.      makeheap();                                   O(N)
5.      for (i = N; i > 1; i--) {                     O(N)
6.         t = A[i];                                  O(N)
7.         a = A[i];                                  O(N)
8.         k = 2;                                     O(N)
9.         while (k < i) {                            B_W(H)+O(N)
10.           if (A[k] < A[k + 1])                    B_W(H)+O(N)
11.              k++;                                 C_W(H)
12.           if (a >= A[k]) break;                   B_W(H)+O(N)
13.           A[k / 2] = A[k];                        B_W(H)
14.           k *= 2;                                 B_W(H)
15.           }
16.        k /= 2;                                    O(N)
17.        A[k] = a;                                  O(N)
18.        A[i] = t;                                  O(N)
19.        }
20.     }
```

Figure 1.2: Williams's Heapsort implemented in C.

with its larger child until a either reaches the bottom of the heap or is at least as large as both of its children. This process restores heap order on the unsorted keys since it makes a at least as large as its children (if any), while leaving all keys above a larger than their children as well. Since the heap is a complete binary tree, a travels at most $\lg N$ levels before it stops, where lg is the base two logarithm. This gives the claimed bound on the running time of the algorithm.

Figure 1.2 gives C code to implement the algorithm that has just been described. Each iteration of the outer loop selects the largest unsorted key t and places it in its final position after fixing up the heap. Note the call to makeheap at the beginning of the code. Since the keys are initially all unsorted, they must first be rearranged to form a heap. The code for makeheap will be given later. The implementation of Heapsort given in Figure 1.2 is the original version of Heapsort, first proposed by Williams in 1964 [28]; it will thus be referred to as Williams's Heapsort. This implementation follows Williams's

original presentation in using the key in A[1] as a sentinel in the event that line 10 is executed with k = i − 1.

Along with the code for Williams's Heapsort, Figure 1.2 also gives the cost of each line. Given a permutation on N keys in the array A[1...N], let H be the heap into which the permutation is transformed by makeheap. It will be proved later that makeheap can take at most $O(N)$ time, which is noted in the figure. Similar cost estimates are provided for the other lines of the algorithm. Specifically, the figure at the right of a line of code shows, to within a $O(N)$ additive error, the number of times that the line is executed when sorting the heap H. Note that this defines the quantities $B_W(H)$ and $C_W(H)$ to be the number of times lines 13 and 11 are executed when sorting H. (This notation was chosen for compatibility with Knuth [18] who uses B and C for $B_W(H)$ and $C_W(H)$ respectively. Actually, Knuth's quantities also include some of the work performed to build the heap, but since the heap is built in linear time, this difference will be ignored.) If this code were compiled to produce an assembly language implementation, many lines would be implemented by more than one instruction, and in some cases, not all of these instructions would be executed the same number of times. However, in reasonable compilations the expressions given are valid for all instructions associated with a given line.

Williams's Heapsort operates in $O(N \log N)$ time in the worst case, is relatively simple to code, and requires no extra storage beyond a constant number of variables. This is enough to make Williams's Heapsort a basic, well known sorting algorithm [1, 6, 15, 18, 25]. Indeed, these properties are sufficient for *Numerical Recipes* [22] to prefer Williams's Heapsort to Quicksort. Floyd ([18], Ex. 5.2.3.18) proposed a modification to Williams's Heapsort that results in an algorithm that is even better, both practically and theoretically. In particular, note that as key a is moved down the heap, Williams's Heapsort makes two key comparisons every time a moves down a level; first the children of a are compared, and then the larger child is compared with a. Floyd suggested making only the comparison between the children of a, then comparing the children of the larger child, and continuing until a path has been found to the bottom of

```
1.   void Floyd() {
2.      long i, k;
3.      key a, t;

4.      makeheap();                                   O(N)
5.      for (i = N; i > 1; i--) {                     O(N)
6.         t = A[1];                                  O(N)
7.         a = A[i];                                  O(N)
8.         k = 2;                                     O(N)
9.         while (k < i) {                            N lg N + O(N)
10.           if (A[k] < A[k + 1])                    N lg N + O(N)
11.              k++;                                 C_F(H)
12.           A[k / 2] = A[k];                        N lg N + O(N)
13.           k *= 2;                                 N lg N + O(N)
14.           }
15.        k /= 2;                                    O(N)
16.        while (a > A[k]) {                         B_F(H) + O(N)
17.           k /= 2;                                 B_F(H)
18.           A[k] = A[k / 2];                        B_F(H)
19.           }
20.        A[k] = a;                                  O(N)
21.        A[i] = t;                                  O(N)
22.        }
23.     }
```

Figure 1.3: Floyd's Heapsort implemented in C.

the heap along which every key is the larger of its parent's two children. In Williams's Heapsort, this is precisely the path that is followed by a as it is moved down the heap, except that a need not go all the way to the bottom. Once this path has been found, it remains only to find the correct position for a along the path by searching up from the bottom of the heap. It will be proved later that the average key a ends up near the bottom of the heap, so Floyd's variant benefits by *performing* half as many comparisons on the way down the heap at the expense of a little bit of extra work to move a back up into position. Indeed, the typical key a moves so far down the heap that this variant is asymptotically optimal in the average case with respect to the number of comparisons performed. For this reason this basic variant is the subject of most articles on Heapsort in the literature today [3, 4, 11, 12, 26, 27].

```
1.    void makeheap() {
2.       long i, k;
3.       key a;

4.       for (i = N / 2; i > 0; i--) {
5.          a = A[i];
6.          k = 2 * i;
7.          while (k < N) {
8.             if (A[k] < A[k + 1]) k++;
9.             A[k / 2] = A[k];
10.            k *= 2;
11.            }
12.         if (k == N) A[k / 2] = A[k];
13.         else k /= 2;
14.         while ((a > A[k]) && (k > i)) {
15.            k /= 2;
16.            A[k] = A[k / 2];
17.            }
18.         A[k] = a;
19.         }
20.      }
```

Figure 1.4: makeheap() implemented in C.

It was noted previously that the running time of Williams's Heapsort is $O(N \log N)$. The same bound obtains for Floyd's variant of Heapsort since the amount of work required to move a key a into position is again bounded by a multiple of the height of the heap, which is bounded by $\lg N$.

Figure 1.3 gives C code for this variant of Heapsort, which will be referred to as Floyd's Heapsort. The only differences between the code given here and the code given previously are the elimination of line 12 of Figure 1.2, which compared a to keys along the path down the heap, and the addition of lines 16–19 of Figure 1.3, by which a is moved into position once the path to the bottom has been found. As with the code for Williams's Heapsort, each line is labeled with the number of times it is executed when sorting the heap H produced by makeheap; as before $B_{\mathcal{F}}(H)$ and $C_{\mathcal{F}}(H)$ are defined by the number of executions of lines 17 and 11 respectively.

Note that the key a stops at the same position in the heap whether it is being

moved down by Williams's Heapsort or Floyd's Heapsort. This means that both versions generate the same sequence of progressively smaller intermediate heaps when sorting a list. This causes certain properties of Heapsort to be the same for both versions. In such cases, the term Heapsort will be used to refer to both versions at once.

Heapsort requires that the input array A be arranged in the form of a heap before it can begin sorting. Williams's original presentation [28] built this heap by regarding the first key as a heap of size one and augmenting it by repeated insertion of the remaining keys. Williams's heap building method will be further examined in Chapter 5. The results of the next three chapters, however, will assume use of the well known bottom-up heap building method proposed by Floyd [13]. Heapsort sorts by repeatedly swapping the largest key in the heap A[1] with the last key in the heap a = A[i], and then fixing the heap by moving a down to an appropriate position. Floyd observed that the code for fixing a heap can be generalized and used to build a heap. Whenever the subtrees rooted at A[2 * i] and A[2 * i + 1] satisfy heap order, Heapsort's method of fixing a heap can be applied to the key a = A[i], moving a down the heap, with the result that the subtree rooted at A[i] will also satisfy heap order. Now the subtrees rooted at A[i] for $i \geq \lceil N/2 \rceil$ automatically satisfy heap order since they each consist of a single node. A heap on all N keys can thus be built by successively fixing the heaps rooted at A[i] as i is decreased from $\lfloor N/2 \rfloor$ to 1. Figure 1.4 gives C code for bottom-up heap building; as i ranges from N / 2 down to 1, it fixes the heap rooted at a = A[i] by moving the key a down the heap. Note that this code finds the position for a in the manner of Floyd's Heapsort, so lines 5–18 are analogous to lines 7–20 of Figure 1.3; the only major additions are line 12 which handles the case that N is even, and lines 6 and 14 which have been modified to restrict a's travels to the subheap rooted at A[i].

The bottom-up method of heap building is well known because of its linear running time. The sum of the heights of the nodes in a complete binary tree on N keys is N minus the sum of the bits in the binary representation of N; from this it follows that Floyd's heap building routine preforms $\Theta(N)$ work overall, and fewer than $2N$ key comparisons, when constructing a heap on N keys. Another nice property of the

bottom-up heap building method is that it preserves randomness. If a permutation is chosen with uniform probability from the space of all permutations on N keys, the result of building a heap out of the elements of the permutation, using the bottom-up method, is a random heap structure, uniformly distributed over the space of all heap structures on N keys. See [18] for a proof of this fact. Because of these properties, all heaps will henceforth be assumed to have been constructed using this bottom-up method. Other heap building methods will be discussed only in Chapter 5.

It is the purpose of this work to give the asymptotic number of executions of each instruction in reasonable implementations of the two versions of Heapsort in the best, worst, and average cases. The next chapter re-examines both versions of Heapsort and gives definitions and preliminary results that are needed to support later constructions. Chapter 3 contains the main result of the thesis; assuming that the input to Heapsort is a randomly permuted list of distinct keys, the average key a is moved asymptotically all way to the bottom of the heap, and the instruction "k++," on line 11 of both versions of Heapsort, is executed asymptotically half of the time. Best and worst case asymptotics are given for both versions of Heapsort in Chapter 4. It is shown for both versions of Heapsort that the instruction "k++" on line 11 is executed at least $\frac{1}{4}N \lg N$ times and at most $\frac{3}{4}N \lg N$ times; this result is developed in the context of a construction due to Sedgewick [23, 24] which maximizes the number of comparisons performed by Floyd's Heapsort. Chapter 5 presents conclusions, open problems, and related areas of research.

To the best of the author's knowledge, all results in this thesis are new except as explicitly noted to the contrary. The presentation of these results found in [23] has been submitted for possible publication in *Journal of Algorithms*.

Chapter 2

Preliminaries

The last chapter introduced Heapsort. This chapter makes a few more introductions that are needed to support a detailed analysis. In particular, this chapter will explain how to construct heaps by viewing the sorting process in reverse. To avoid confusion, it should be noted at the outset that in this chapter heap construction will mean starting with a heap on a single key and adding progressively larger keys until a heap on N keys is reached. This is not the same as the problem of building a heap from a permutation that was discussed briefly in the previous chapter. This chapter will conclude by defining heap construction costs that correspond closely with the cost of sorting a heap.

One concept is central to the average, best, and worst case bounds of the next three chapters — the constructions required for these bounds are based on a technique developed by Sedgewick [23, 24] which views the Heapsort process in reverse. This technique requires the assumption, here and in all further chapters, that the input to Heapsort is a permutation on the integers in $[1 \ldots N]$. It is possible to make this assumption without loss of generality since it will be assumed everywhere that all input keys are distinct.

Let H_N be the set of all heaps on the distinct integers in $[1, N]$. Then given a heap $\mathtt{A}[1 \ldots N] \in H_N$, one iteration of the main loop of Heapsort removes the root of the heap and replaces it with the key from $\mathtt{A}[N]$, it then moves that key down to

```
1.    void pulldown(long k) {
2.       A[k] = A[N];
3.       while (k > 1) {
4.          A[k] = A[k / 2];
5.          k /= 2;
6.          }
7.       A[1] = N;
8.       }
```

Figure 2.1: pulldown implemented in C.

some position A[k], leaving a heap on $N-1$ keys. Following Doberkat [7, 9], this establishes a mapping $\Phi : H_N \rightarrow H_{N-1}$. The reverse process would have to take the key from some position A[k] and place it in position A[N] at the bottom, fill the resulting vacancy at A[k] by shifting down all keys above it in the heap (leaving a vacancy at the root), and assign N to the root position A[1]. Figure 2.1 gives C code for the procedure pulldown that makes this process explicit.

Figure 2.2 shows the result of calling pulldown(11) to expand a heap from 12 keys to 13 keys. This is followed by an iteration of the main loop of either version of Heapsort, with i = 13, which returns the original heap on 12 keys. In this respect, pulldown and the main loop of Heapsort are inverses of each other. It will sometimes be useful to refer to the pulldown process less formally than as a call to pulldown with a given position as its argument. In particular, it is often convenient to refer to "pulling down" a given key by its value. Figure 2.2 thus shows the result of pulling down 2. This usage is not ambiguous since it is assumed that all keys are distinct.

The pulldown procedure reverses a step of the Heapsort process by transforming a heap on $N-1$ keys to a heap on N keys. Let $\mathcal{P}(H_N)$ be the power set of H_N. Also, let $\Phi^{-1} : H_{N-1} \rightarrow \mathcal{P}(H_N)$ be the inverse of Φ, assigning to each heap on $N-1$ keys the set of heaps on N keys that are mapped to it by Φ. Then given a heap $H \in H_{N-1}$, every element of $\Phi^{-1}(H)$ can be generated from H by calling pulldown on an appropriate value of $k \leq N-1$. Note, however, that not all values of $k \leq N-1$ result in legitimate heaps. After calling pulldown it must be the case that A[N] < A[$\lfloor N/2 \rfloor$]. This is so if and only

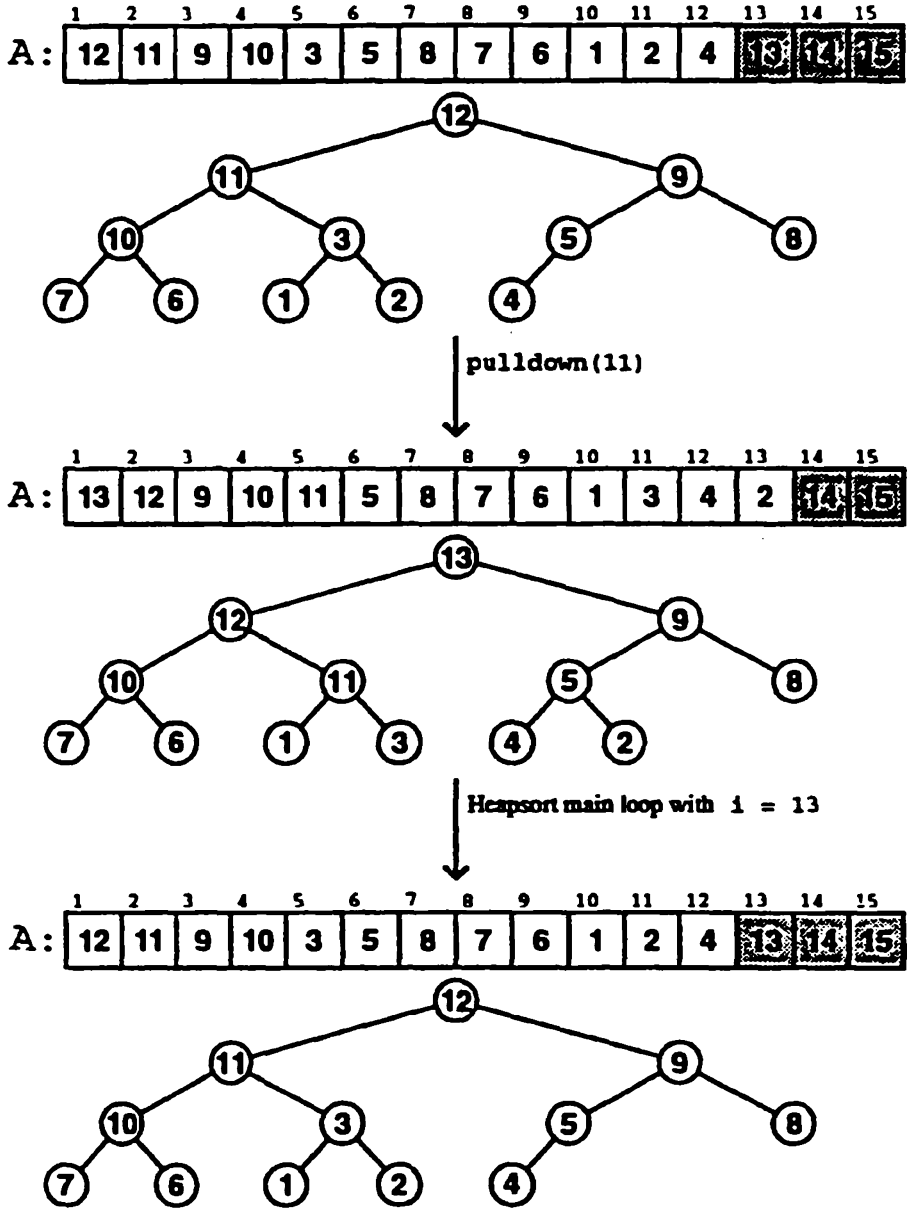

Figure 2.2: pulldown(11) followed by an iteration of Heapsort with i = 13.

if A[k] $\leq$ A[$\lfloor N/2 \rfloor$] prior to calling pulldown(k). From this it follows that there are exactly A[$\lfloor N/2 \rfloor$] elements in the image under Φ^{-1} of any heap A[$1 \ldots N-1$] $\in H_{N-1}$. To expand the 12 key heap of Figure 2.2, for example, any of 5, 6, 10, 11, or 12 would be acceptable arguments to pulldown since these positions contain keys that are less than or equal to 5 = A[$\lfloor N/2 \rfloor$].

Pulldowns are interesting because every heap can be associated with a unique sequence of pulldowns. Running Heapsort results in a sequence of progressively smaller heaps. The reverse of such a progression corresponds to a unique sequence of calls to pulldown. Sedgewick [23, 24] has used this fact to explicitly construct heaps with certain properties and to enumerate conveniently all possible heaps on a given number of keys. Sequences of pulldowns will also be crucial to the analysis of the average case of Heapsort. This motivates the following definition:

DEFINITION: A pulldown sequence $P = \{p_i\}_{i=2}^{N}$ and the heap $H(P)$ that it constructs in A[$1 \ldots N$] are defined inductively as follows:

- For $N = 1$, the empty sequence $\{p_i\}_{i=2}^{1}$ is a pulldown sequence and the heap it constructs, $H(\{p_i\}_{i=2}^{1})$, is the heap containing the single key "1".
- For $N > 1$, a sequence $\{p_i\}_{i=2}^{N}$ is a pulldown sequence if $\{p_i\}_{i=2}^{N-1}$ is a pulldown sequence, and if a valid heap $H(\{p_i\}_{i=2}^{N})$ is constructed when pulldown(p_N) is called given $H(\{p_i\}_{i=2}^{N-1})$.

The use of pulldown to construct a heap is demonstrated by Figure 2.3, which shows the heap built by the pulldown sequence:

Heap Size:	1	2	3	4	5	6	7	8	9	10	11
Pulldown Sequence:	1	2	2	2	3	2	4	3	5	7	

Pulldown sequences will prove useful because the cost of sorting a heap is closely related to the cost of constructing the heap by repeated pulldowns. In order to take advantage of this correspondence, it is necessary to measure the cost of each pulldown

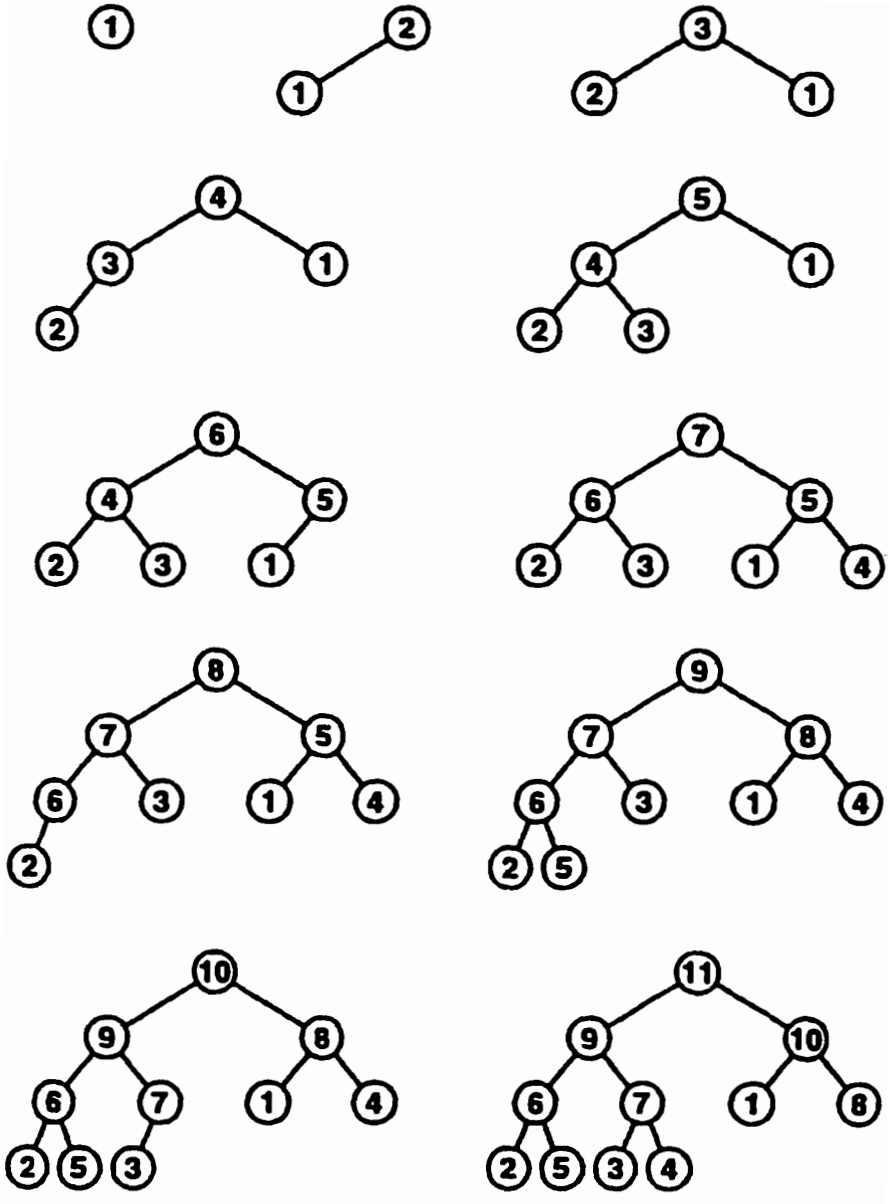

Figure 2.3: A heap constructed by a pulldown sequence.

in a pulldown sequence in a way that is directly related to the cost that will be incurred by Heapsort when reversing that pulldown. This motivates the following definitions:

DEFINITION: Given a pulldown sequence, $P = \{p_i\}_{i=2}^{N}$, define $B_W(p_i)$, the B_W cost of pulldown p_i, to be the level containing the position p_i, where the root is considered to be level 1. That is, $B_W(p_i) = \lceil \lg(p_i + 1) \rceil$. The B_W cost sequence associated with P is defined to be $B_W(P) = \{B_W(p_i)\}_{i=2}^{N}$, and the B_W cost of the heap $H(P)$ is defined to be $B_W(H(P)) = \sum_{i=2}^{N} B_W(p_i)$.

Note that when the main loop of Williams's Heapsort reverses pulldown p_i, the key a will be moved from the root at the first level down to the level of position p_i. In the process, line 13 is executed a total of $B_W(p_i) - 1$ times. It follows that for any pulldown sequence $P = \{p_i\}_{i=2}^{N}$, the B_W cost of heap $H(P)$ is equal to $B_W(H(P)) + N - 1$.

DEFINITION: Given a pulldown sequence, $P = \{p_i\}_{i=2}^{N}$, define $C_W(p_i)$, the C_W cost of pulldown p_i, to be the number of times the right child is selected along the path from the root to position p_i. That is, $c_i = bit(p_i) - 1$ where $bit(x)$ is the number of 1's in the binary representation of x. Define the C_W cost sequence associated with P to be $C_W(P) = \{C_W(p_i)\}_{i=2}^{N}$, and define the C_W cost of the heap $H(P)$ to be $C_W(H(P)) = \sum_{i=2}^{N} C_W(p_i)$.

When pulldown p_i is reversed by the main loop of Williams's Heapsort, a path has to be found from the root to position p_i. This makes a right turn, and thus executes line 11, every time a 1 is encountered in the binary representation of p_i (excluding the leading 1, of course). Sometimes line 11 is executed once more; in particular, when position p_i has two children, that is when $2p_i \leq i$, and when the right child has a larger key than the left child, line 11 is executed an additional time before executing the break statement on line 12. Because this adds at most one execution of line 11 per iteration of the main loop, it follows for any pulldown sequence $P = \{p_i\}_{i=2}^{N}$ that the C_W cost of heap $H(P)$ equals $C_W(H(P)) + O(N)$.

Some similar definitions will now be given for Floyd's Heapsort.

DEFINITION: Given a pulldown sequence, $P = \{p_i\}_{i=2}^{N}$, define $B_F(p_i)$, the B_F cost of pulldown p_i, to be a bound on the height of position p_i, namely $B_F(p_i) = \lceil \lg(i+1) \rceil - \lceil \lg(p_i+1) \rceil$. As before, the B_F cost sequence associated with P is $B_F(P) = \{B_F(p_i)\}_{i=2}^{N}$, and the B_F cost of the heap $H(P)$ is $B_F(H(P)) = \sum_{i=2}^{N} B_F(p_i)$.

Floyd's Heapsort reverses pulldown p_i by tracing a path to the bottom of the heap and then moving the key a up the path into position. The number of times line 17 is executed to move a up is the difference between the number of the level where the path hit the bottom and the level of position p_i. Since there are usually two "bottom" levels to the heap, this difference can be either $\lceil \lg(i+1) \rceil - \lceil \lg(p_i+1) \rceil$ or $\lceil \lg(i+1) \rceil - \lceil \lg(p_i+1) \rceil - 1$. This implies that for any pulldown sequence $P = \{p_i\}_{i=2}^{N}$, the B_F cost of $H(P)$ is $B_{\mathcal{F}} + O(N)$.

DEFINITION: Given a pulldown sequence, $P = \{p_i\}_{i=2}^{N}$, define $C_F(p_i)$, the C_F cost of pulldown p_i, to be the number of times line 11 is executed when an iteration of Floyd's Heapsort is performed to undo the effects of pulldown(p_i). Again, the C_F cost sequence associated with P is $C_F(P) = \{C_F(p_i)\}_{i=2}^{N}$, and the C_F cost of heap $H(P)$ is $C_F(H(P)) = \sum_{i=2}^{N} C_F(p_i)$.

This definition is very nice in the sense that it results in exact equality between the C_F cost of $H(P)$ and the value of $C_{\mathcal{F}}(H(P))$. It is somewhat less nice in that unlike the previous definitions, the C_F cost of a pulldown p_i is not a strict function of the position p_i, but depends upon the context of $p_1 \ldots p_{i-1}$. Unfortunately, this is unavoidable; once Floyd's Heapsort has moved a below position p_i, its behavior is dependent not upon p_i, but upon the structure of the heap created by the first $i-1$ pulldowns.

Note that cost sequences are defined within the context of pulldown sequences. A cost sequence C is thus always assumed to be associated with at least one pulldown sequence P. To illustrate these definitions, the table on the next page gives the cost sequences associated with the pulldown sequence that constructs the heap in Figure 2.3.

Heap Size:	1	2	3	4	5	6	7	8	9	10	11
Pulldown Sequence:		1	2	2	2	3	2	4	3	5	7
B_W Cost Sequence:		1	2	2	2	2	2	3	2	3	3
C_W Cost Sequence:		0	0	0	0	1	0	0	1	1	2
B_F Cost Sequence:		1	0	1	1	1	1	1	2	1	1
C_F Cost Sequence:		0	0	0	1	1	1	0	2	1	2

The following lemma summarizes for future reference the relations that hold on the heap costs that have just been defined.

LEMMA 2.1: Given any pulldown sequence $P = \{p_i\}_{i=2}^{N}$, the following hold:

1. $B_W(H(P)) = B_W(H(P)) - N + 1$.
2. $C_W(H(P)) = C_W(H(P)) + O(N)$.
3. $B_F(H(P)) = B_F(H(P)) + O(N)$.
4. $C_F(H(P)) = C_F(H(P))$.
5. $B_W(H(P)) + B_F(H(P)) < N \lg N$.
6. $C_W(H(P)) \leq C_F(H(P))$.
7. $B_W(H(P)) < N \lg N$.

Proof: The correctness of the first four parts of the lemma has already been demonstrated. Part 5 follows from:

$$\sum_{i=2}^{N}(B_W(p_i) + B_F(p_i)) = \sum_{i=2}^{N}(\lceil \lg(p_i + 1) \rceil + \lceil \lg(i + 1) \rceil - \lceil \lg(p_i + 1) \rceil) < N \lg N.$$

The sixth part of the lemma is a consequence of the fact that Floyd's Heapsort finds a path all the way to the bottom of the heap, while Williams's Heapsort sometimes stops short of the bottom. Part 7 follows from part 5 and the fact that $B_F(H(P))$ is always a non-negative quantity. □

The foundations needed for the main results of the next two chapters have now been laid. The next chapter will prove average case bounds for Heapsort by showing that all but an exponentially decreasing fraction of pulldown sequences build heaps whose costs are close to the expected value. The best and worst case bounds will be proved in the following chapter by specifying pulldown sequences that build heaps with low and high costs.

Chapter 3

Average Case

This chapter contains the main result of this thesis; it analyzes the asymptotic behavior of Williams's Heapsort and Floyd's Heapsort in the average case. In particular, it will be shown that given a uniformly generated random permutation on N keys in $A[1 \ldots N]$, if H is the heap that is built from that permutation by makeheap, then it is expected that $B_W(H) \sim N \lg N$, $C_W(H) \sim \frac{1}{2} N \lg N$, $B_F(H) = O(N)$, and $C_F(H) \sim \frac{1}{2} N \lg N$.

These results will be obtained in a somewhat roundabout way. In particular, recall that if H_N is the set of all heaps on the integers in $[1 \ldots N]$, then makeheap maps the same number of size N permutations to each heap in H_N. From this it follows that the average cost of sorting a uniformly generated random permutation is expected to be the same as the expected cost of makeheap plus the cost of sorting a uniformly generated random heap from H_N. The first four parts of Lemma 2.1 state that to determine the asymptotic complexity of sorting a random heap structure, it is sufficient to determine the expected B_W, C_W, B_F, and C_F costs of the heap constructed by a random pulldown sequence. It is these values that will be established in this chapter.

The average B_W cost of a random heap structure is determined by showing that only an exponentially decreasing fraction of all pulldown sequences construct heaps with costs significantly different from the claimed average cases. The value for the B_F cost of the average heap will follow immediately. The expected C_W and C_F costs of a random heap

will be established by similar methods.

In the remainder of the chapter, any pulldown sequence whose length is not explicitly mentioned will be assumed to construct a heap of size N; likewise, any cost sequence whose length is not mentioned will be assumed to be associated with a pulldown sequence that constructs a heap of size N. Since frequent reference will be made to uniformly generated random heaps, let R_N be a random variable that is generated with uniform probability from H_N, and let P_N be a uniform random variable over the space of all permutations on the integers in $[1 \ldots N]$. Munro [20] suggested the proof of the following lemma, which will be useful throughout the chapter:

LEMMA 3.1: For any N, the number of heaps $|H_N|$ is $|H_N| > N!/4^N$.

Proof: There are $N!$ permutations on the N integers in $[1 \ldots N]$. It was noted in the first chapter, when makeheap was introduced, that makeheap requires fewer than $2N$ key comparisons to build a heap on N keys. Since makeheap maps the same number of permutations to each heap, it follows that $|H_N| > N!/2^{2N}$. □

Lemma 3.1 has two main virtues. It is easily seen to be correct for arbitrary N, and it is sufficiently strong for the results of the remainder of the chapter. It is not tight, however. Gonnet and Munro [16] show that $|H_N| = N!/2^{1.3644\ldots N+O(\log N)}$. The usefulness of Lemma 3.1 derives from the way that Theorem 3.1 and Lemmas 3.4 and 3.5 will be proved. The proofs of these statements will need an estimate of $|H_N|$ in order to show that only an exponentially small fraction of the heaps in H_N have abnormal costs. Before proceeding to the proof of Theorem 3.1, however, the following two lemmas need to be presented.

LEMMA 3.2: Let a_1, a_2, and a_3 be real variables and let p, q, and r be positive real constants. Suppose that the constraints:

$$a_1 + a_2 + a_3 = p \qquad (r-1)a_1 + (r)a_2 + (r+1)a_3 = q \qquad a_i \geq 0, (i = 1,2,3)$$

are satisfied by some assignment to the a_i where at least two a_i are positive. Then $a_1^{a_1} a_2^{a_2} a_3^{a_3}$ is minimized subject to these constraints by exactly one assignment to the a_i

for which all the a_i are positive; this assignment satisfies the relation $a_1/a_2 = a_2/a_3$. ($a_i^{a_i}$ is assumed to equal 1 when $a_i = 0$.)

Proof: This lemma is needed only in the proof of Lemma 3.3. Its proof does not explicitly involve Heapsort, so it is relegated to the appendix. □

LEMMA 3.3: Given positive integers N and T, let $n = \lceil \lg(N+1) \rceil$ and $t = n - T/N$. Then for N sufficiently large, the number of B_W cost sequences C having cost equal to T is bounded by:

$$\left|\{C = \{c_i\}_{i=2}^{N} : \sum_{i=2}^{N} c_i = T\}\right| \leq \begin{cases} n^N, & \text{if } T < N(n - 2\lg n); \\ N^n 3^N (t+1)^N, & \text{if } N(n-2\lg n) \leq T \leq N(n-10). \end{cases}$$

Proof: The first half of the bound is trivial; for any B_W cost sequence C, there are at most n choices for each c_i, which implies $|\{C : \sum_{i=2}^{N} c_i = T\}| < n^N$. The proof of the second half of the bound is considerably more difficult. Note that the second half of the bound assumes $10 \leq t \leq 2\lg n$.

Consider a B_W cost sequence C; let Na_i be the number of c_i in C for which $c_i = n-i$, where i ranges from 0 to $n-1$. Note that $Na_i > 0$ for $i > 1$ since the smallest key always resides at the bottom of the heap and thus must be pulled down at least once every time a complete row is added to the heap. There are fewer than N^n ways to choose the Na_i, so an upper bound on $|\{C : \sum_{i=2}^{N} c_i = T\}|$ is given by N^n multiplied by the maximum number of B_W cost sequences C that can correspond to any fixed choice of the Na_i for which $\sum_{i=0}^{n-1} iNa_i = T$. It follows that $|\{C : \sum_{i=2}^{N} c_i = T\}|$ is bounded from above by the maximum value of:

$$N^n \binom{N}{Na_0, Na_1, \ldots, Na_{n-1}} \tag{3.1}$$

subject to $a_i > 0$ for $i > 1$ and:

$$\sum_{i=0}^{N-1} Na_i = N \tag{3.2}$$

and

$$\sum_{i=0}^{N-1} iNa_i = tN. \tag{3.3}$$

Applying Stirling's Formula to (3.1), the proof of the lemma will be complete once it has been shown for N sufficiently large, that all choices of the a_i subject to (3.2)

and (3.3) satisfy:

$$\frac{(\frac{N}{e})^N}{(\frac{Na_0}{e})^{Na_0}\cdot(\frac{Na_1}{e})^{Na_1}\cdot\ldots\cdot(\frac{Na_n}{e})^{Na_n}} = \frac{(\frac{N}{e})^N}{(\frac{N}{e})^N\left(\prod_{i=0}^{n} a_i^{a_i}\right)^N} < 3^N(t+1)^N \qquad (3.4)$$

where $a_i > 0$ for $i > 1$, and $(a_i/e)^{Na_i}$ is taken to equal 1 when $a_i = 0$. The remainder of this proof is thus devoted to minimizing:

$$\left(\prod_{i=0}^{n} a_i^{a_i}\right)^N \qquad (3.5)$$

subject to (3.2) and (3.3), where $a_i > 0$ for $i > 1$, and $a_i^{a_i}$ is taken to equal 1 when $a_i = 0$.

Expression (3.5) does not require that all of the a_i be positive. By Lemma 3.2, however, the fact that a_2 and a_3 are positive implies that there exists a minimum for (3.5) where a_1 and a_0 are positive as well. Furthermore, Lemma 3.2 implies that this minimum is attained when the a_i form an exponential sequence $a_i = ca^i$ for some positive, real a and c which are dependent upon N and T. The exact values of a and c are determined by (3.2) and (3.3); these constraints can be restated in terms of a and c as follows:

$$cN\sum_{i=0}^{n-1} a^i = N \qquad (3.6) \qquad \text{and} \qquad cN\sum_{i=0}^{n-1} ia^i = tN. \qquad (3.7)$$

Note that these equations, together with the fact that $t < n/2$, imply $a < 1$. The expression (3.5) can also be bounded in terms of a and c:

$$\left(\prod_{i=0}^{n} a_i^{a_i}\right)^N \geq \left(\prod_{i=0}^{n} (ca^i)^{ca^i}\right)^N = c^{(cN\sum_{i=0}^{n} a^i)} a^{(cN\sum_{i=0}^{n} ia^i)} = c^N a^{tN}. \qquad (3.8)$$

Inequality (3.8) should be enough to finish proof of the lemma if it is coupled with good lower bounds on a and c. Such bounds are provided by (3.9) and (3.10) which will be proved later:

$$a > \frac{t-1}{t} \qquad (3.9) \qquad \text{and} \qquad c > \frac{1}{t+1}. \qquad (3.10)$$

Inequalities (3.9) and (3.10) can be plugged into (3.8) to produce:

$$\left(\prod_{i=0}^{n} a_i^{a_i}\right)^N \geq c^N a^{tN} > \left(\frac{1}{t+1}\right)^N \left(\frac{t-1}{t}\right)^{tN} = \left[\left(\frac{1}{t+1}\right)\left(1-\frac{1}{t}\right)^t\right]^N > \left(\frac{1}{t+1}\cdot\frac{1}{3}\right)^N.$$

Since this establishes (3.4), the proof of the lemma would be complete, except that it now remains to show that (3.9) and (3.10) hold for sufficiently large N.

Dividing (3.6) and (3.7) by N and obtaining closed forms for their sums, gives the following two equations:

$$\frac{(1-a^n)}{1-a}c = 1 \qquad (3.11) \qquad \text{and} \qquad \frac{(n-1)a^{n+1} - na^n + a}{(1-a)^2}c = t. \quad (3.12)$$

Solving (3.11) for c in terms of a, and substituting the result for c in (3.12) yields:

$$t = \frac{a(1-a^n) + n(a-1)a^n}{(1-a)(1-a^n)} = \frac{a}{1-a} + n\frac{a^n}{1-a^n}. \qquad (3.13)$$

Equation (3.13) implies that $t > a/(1-a)$, from which it follows that

$$a < \frac{t}{t+1}. \qquad (3.14)$$

The fact that $t \leq 2\lg n$ together with (3.14) yields:

$$a < \frac{2\lg n}{2\lg n + 1}. \qquad (3.15)$$

Combining (3.13) and (3.15) gives:

$$t < \frac{a}{1-a} + n\frac{(\frac{2\lg n}{2\lg n+1})^n}{1-(\frac{2\lg n}{2\lg n+1})^n}.$$

But since $n(\frac{2\lg n}{2\lg n+1})^n$ approaches zero as n gets large, this implies

$$t < \frac{a}{1-a} + 1$$

for sufficiently large n. Inequality (3.9) follows by simple algebra. Now note that (3.11) implies $c > 1 - a$; together with (3.14) this proves (3.10). This concludes the proof of the lemma. □

It is time now to begin proving the main results of this chapter.

THEOREM 3.1: The expected value of $B_W(R_N)$ is $N\lg N + O(N)$.

Proof: By the discussion in the introduction, it is sufficient to show that the average over all pulldown sequences P of $B_W(H(P))$ is $N\lg N + O(N)$. By part 7 of Lemma 2.1,

no pulldown sequence P can have B_W cost more than $N \lg N$ so the remainder of the proof is devoted to proving that the average pulldown sequence has B_W cost at least $N \lg N + O(N)$.

Given a B_W cost sequence $C = \{c_i\}_{i=2}^N$, consider the pulldown sequences $P = \{p_i\}_{i=2}^N$ for which $B_W(P) = C$. Note in particular that for any i there are at most $2^{c_i - 1}$ possible choices for position p_i. This means that:

$$\left|\{P : B_W(P) = C\}\right| < \prod_{i=2}^{N} 2^{c_i} = 2^{\sum_{i=2}^{N} c_i} = 2^{B_W(H(P))}. \tag{3.16}$$

What is desired at this point is a bound on $|\{P : B_W(H(P)) \leq T\}|$ that is exponentially smaller than $|H_N|$ for some reasonably large value of T. Let $n = \lceil \lg(N+1) \rceil$. Then such a bound is easily obtained for $T = N \lg N - N \lg n - 4$; the total number of cost sequences was bounded in the proof of Lemma 3.3 at $|\{C = \{c_i\}_{i=2}^N\}| < n^N$, which with (3.16) implies $|\{P : B_W(H(P)) < T\}| < n^N 2^T = (N/16)^N$. This argument is sufficient to show that the B_W cost of R_N is expected to be $\sim \frac{1}{2} N \lg N$, but does not yield the bound given in the theorem statement. Munro [20] suggested that a better bound could follow from a more careful estimate of $|\{C = \{c_i\}_{i=2}^N : \sum_{i=2}^N c_i \leq T\}|$. The result is Lemma 3.3 and its application with (3.16) to prove the following:

$$\begin{aligned}
&\left|\{P : B_W(H(P)) \leq N(n-10)\}\right| \\
&\quad \leq \sum_{i=0}^{\lfloor N(n-2\lg n)\rfloor} n^N 2^i + \sum_{\lfloor N(n-2\lg n)\rfloor+1}^{N(n-10)} N^n 3^N \left(n - \frac{i}{N} + 1\right)^N 2^i \\
&\quad < \left(\frac{2N}{n}\right)^N + N^n 3^N \sum_{\lfloor n-2\lg n\rfloor}^{(n-10)} N(n-i+1)^N 2^{N(i+1)} \\
&\quad < \left(\frac{2N}{n}\right)^N + N^n 3^N \sum_{\lfloor n-2\lg n\rfloor}^{(n-10)} N\left((11)\left(\frac{12}{11}\right)^{n-10-i}\right)^N 2^{N(i+1)} \\
&\quad = N^{n+1} 3^N (11)^N 2^{N(n-9)} (1 + O((3/4)^N)) \\
&\quad = O\left(\left(\frac{N}{15}\right)^N\right).
\end{aligned}$$

The remainder of the proof is straightforward. The average of $B_W(H(P))$ over all

pulldown sequences $P = \{p_i\}_{i=2}^{N}$ is greater than:

$$\begin{aligned}\frac{(|H_N| - O((N/15)^N))N(n-10) + O((N/15)^N)0}{|H_N|} &= N(n-10)\left(1 - \frac{O((N/15)^N)}{|H_N|}\right) \\ &> N(n-10)\left(1 - \frac{O((N/15)^N)}{N!/4^N}\right) \\ &> N(n-10)\left(1 + O((4e/15)^N)\right)\end{aligned}$$

This completes proof of the theorem. □

COROLLARY: The expected value of $B_F(R_N)$ is $O(N)$.

Proof: This follows immediately from Theorem 3.1 and part 5 of Lemma 2.1. □

LEMMA 3.4: $C_W(R_N)$ is expected to be greater than $\frac{1}{2}N \lg N + O(N\sqrt{\log N \log\log N})$.

Proof: The proof begins by bounding the number of pulldown sequences that can be associated with a given C_W cost sequence. Let $n = \lceil \lg(N+1) \rceil$ be the number of levels in a heap on N keys. Then given k, there are at most $\sum_{i<n} \binom{i}{k} = \binom{n}{k+1}$ positions in a heap on N keys with C_W cost equal to k. This gives an immediate upper bound on the numher of pulldown sequences P with which a given C_W cost sequence $C = \{c_i\}_{i=2}^{N}$ may be associated:

$$\left|\{P : C_W(P) = C\}\right| \le \binom{n}{c_2+1} \cdot \binom{n}{c_3+1} \cdot \cdots \cdot \binom{n}{c_N+1}.$$

A pulldown sequence P constructs a heap with low C_W cost if $C_W(H(P)) < (N-1)T$ for some cutoff value $T < n/2 - 1$. It can be shown by an inductive argument that $\binom{n}{T+1}^{N-1} \ge \binom{n}{c_2+1} \cdot \binom{n}{c_3+1} \cdot \cdots \cdot \binom{n}{c_N+1}$ whenever $\sum_i (c_i + 1) \le (N-1)(T+1)$. This means that not very many pulldown sequences P can correspond to a single low cost C_W cost sequence $C = \{c_i\}_{i=2}^{N}$:

$$\left|\{P : C_W(P) = C\}\right| \le \binom{n}{T+1}^{N-1} \quad \text{whenever} \quad \sum_{i=2}^{N} c_i \le (N-1)T.$$

A bound on the number of low cost C_W cost sequences is easily obtained by bounding the total number of cost sequences. Given C_W cost sequence $C = \{c_i\}_{i=2}^{N}$, there are

fewer than n choices for each c_i, from which it follows that there are fewer than n^{N-1} choices for C. Since there aren't many cost sequences, not many pulldown sequences P can construct low cost heaps:

$$\left|\{P : C_W(H(P)) < (N-1)T\}\right| \le n^{N-1}\binom{n}{T+1}^{N-1}. \tag{3.17}$$

This bound now needs to be evaluated at an appropriate value of T. So assign $T = \lfloor n/2 - \sqrt{n \lg n} \rfloor - 1$. Then by Stirling's Formula,

$$\begin{aligned}
\ln\binom{n}{T+1} &\le n\ln n - n + \frac{\ln n}{2} \\
&\quad -\left(\frac{n}{2} - \sqrt{n\lg n} + \frac{1}{2}\right)\ln\left(\frac{n}{2} - \sqrt{n\lg n}\right) + \frac{n}{2} - \sqrt{n\lg n} \\
&\quad -\left(\frac{n}{2} + \sqrt{n\lg n} + \frac{1}{2}\right)\ln\left(\frac{n}{2} + \sqrt{n\lg n}\right) + \frac{n}{2} + \sqrt{n\lg n} + O(1) \\
&= n\ln n + \frac{\ln n}{2} - (n+1)\ln\frac{n}{2} \\
&\quad -\left(\frac{n}{2} - \sqrt{n\lg n} + \frac{1}{2}\right)\ln\left(1 - \frac{2\sqrt{n\lg n}}{n}\right) \\
&\quad -\left(\frac{n}{2} + \sqrt{n\lg n} + \frac{1}{2}\right)\ln\left(1 + \frac{2\sqrt{n\lg n}}{n}\right) + O(1) \\
&= -\frac{\ln n}{2} + (n+1)\ln 2 \\
&\quad -\left(\frac{n}{2} - \sqrt{n\lg n} + \frac{1}{2}\right)\left(-\frac{2\sqrt{n\lg n}}{n} - \frac{2n\lg n}{n^2} + O\left(\frac{\log^{3/2} n}{n^{3/2}}\right)\right) \\
&\quad -\left(\frac{n}{2} + \sqrt{n\lg n} + \frac{1}{2}\right)\left(\frac{2\sqrt{n\lg n}}{n} - \frac{2n\lg n}{n^2} + O\left(\frac{\log^{3/2} n}{n^{3/2}}\right)\right) \\
&= -\frac{\ln n}{2} + (n+1)\ln 2 - 2\ln n + O(1).
\end{aligned}$$

It follows that for N sufficiently large, $\binom{n}{T+1} \le N/n^{2.25}$. By (3.17), when N is sufficiently large, there are fewer than

$$n^{N-1}\binom{n}{T+1}^{N-1} \le n^{N-1}\left(\frac{N}{n^{2.25}}\right)^{N-1} \le \left(\frac{N}{12}\right)^{N}$$

pulldown sequences P for which $C_W(H(P)) < (N-1)T$.

This means that the average of $C_W(H(P))$ over all pulldown sequences $P = \{p_i\}_{i=2}^N$, for N sufficiently large, is greater than:

$$\frac{(|H_N| - (N/12)^N)(N-1)T + (N/12)^N 0}{|H_N|} = (N-1)T - \frac{N^N(N-1)T}{12^N|H_N|}.$$

By Lemma 3.1, $|H_N| > N!/4^N > (N/4e)^N$. This means that as N becomes large, the average of $C_W(H(P))$ over all pulldown sequences $P = \{p_i\}_{i=2}^N$ is greater than:

$$(N-1)T - \left(\frac{4e}{12}\right)^N (N-1)T = \frac{Nn}{2} + O(N\sqrt{\log N \log\log N}).\square$$

LEMMA 3.5: $C_F(R_N)$ is expected to be less than $\frac{1}{2}N\lg N + O(N\sqrt{\log N \log\log N})$.

Proof: The proof of this lemma is similar to the proof of Lemma 3.4, though there are some differences. The proof again begins by proving an upper bound on the number of pulldown sequences P that can correspond to a given B_F cost sequence. Let $\lceil \lg(N+1) \rceil$ be the number of levels in a heap on N keys. Given $k < n$, the number of paths to the bottom of the heap that select exactly k right children is bounded above by $\binom{n-1}{k}$. For each such path, there are at most n keys along the path, any of which could be pulled down at a C_F cost of k. This yields the following upper bound on the number of pulldown sequences P having corresponding C_F cost sequence $C = \{c_i\}_{i=2}^N$:

$$\begin{aligned} \left|\{P : C_F(P) = C\}\right| &\le \binom{n-1}{c_2} n \cdot \binom{n-1}{c_3} n \cdot \ldots \cdot \binom{n-1}{c_N} n \\ &< \binom{n}{c_2} \cdot \binom{n}{c_3} \cdot \ldots \cdot \binom{n}{c_N} n^N. \end{aligned}$$

A high cost heap H is one for which $C_F(H) > (N-1)T$ where $T > n/2 + 1$ is a cutoff value. It can be proved inductively that $\binom{n}{T}^{N-1} \ge \binom{n}{c_2} \cdot \binom{n}{c_3} \cdot \ldots \cdot \binom{n}{c_N}$ whenever $\sum_i c_i \ge (N-1)T$. In particular, if $T = \lceil n/2 + \sqrt{n \lg n} \rceil$, this means that for any C_F cost sequence $C = \{c_i\}_{i=2}^N$:

$$\left|\{P : C_F(P) = C\}\right| \le \left(\binom{n}{\lceil n/2 + \sqrt{n \lg n} \rceil}\right)^{N-1} n^N \quad \text{whenever} \quad \sum_{i=2}^N c_i > (N-1)T.$$

Now it was shown during the proof of Lemma 3.4 that there are at most n^{N-1} cost sequences $C = \{c_i\}_{i=2}^{N}$ and that for sufficiently large N:

$$\left(\left\lceil n/2 + \sqrt[n]{n \lg n}\right\rceil\right) = \left(\left\lfloor n/2 - \sqrt[n]{n \lg n}\right\rfloor\right) < N/n^{2.25}.$$

For sufficiently large N, this implies the following bound on the number of pulldown sequences that correspond to high cost C_F cost sequences:

$$\left|\{P : C_W(H(P)) > (N-1)T\}\right| \leq n^{N-1}\left(\frac{N}{n^{2.25}}\right)^{N-1} n^N < \left(\frac{N}{12}\right)^N.$$

This means that for N sufficiently large, the C_F cost of the heap constructed by the average pulldown sequence $P = \{p_i\}_{i=2}^{N}$ is bounded above by:

$$\begin{aligned}
\frac{(|H_N| - (N/12)^N)(N-1)T + (N/12)^N N \lg N}{|H_N|} &< (N-1)T + \frac{N^N N \lg N}{12^N |H_N|} \\
&< (N-1)T + \left(\frac{4eN}{12N}\right)^N N \lg N \\
&= \frac{Nn}{2} + O(N\sqrt{\log N \log\log N}).
\end{aligned}$$

This completes proof of the lemma. □

These two lemmas have done all of the dirty work required to determine the asymptotic values of $C_W(R_N)$ and $C_F(R_N)$. The following theorem ties up loose ends:

THEOREM 3.2: The expected values of $C_W(R_N)$ and $C_F(R_N)$ are both $\sim \frac{1}{2} N \lg N$.

Proof: This theorem follows immediately from Lemmas 3.4 and 3.5 and part 6 of Lemma 2.1. □

This chapter has determined the expected values of $B_W(R_N)$, $C_W(R_N)$, $B_F(R_N)$, and $C_F(R_N)$ by showing that all but an asymptotically negligible number of pulldown sequences will build heaps that have costs near the average. The next chapter will show how a pulldown sequence can construct some of the rare heaps with exceptional costs.

Chapter 4

Extreme Cases

This chapter gives tight bounds on the highest and lowest values taken on by $B_W(H)$, $C_W(H)$, $B_F(H)$, and $C_F(H)$ for any heap H; as in the previous chapter, these bounds are an immediate consequence of bounds that will be proved on $B_W(H)$, $C_W(H)$, $B_F(H)$, and $C_F(H)$. The first section gives upper bounds on the worst cases and lower bounds on the best cases for both versions of Heapsort. The following four sections present three similar constructions that show that the bounds of Section 4.1 are tight. The final section summarizes the chapter's results. Because the constructions of Sections 4.2–4.5 are similar, their presentations can seem repetitive. In the interests of readability, as many redundant details as possible have been eliminated. This means, however, that the constructions should be read in the order in which they are presented.

4.1 Upper and Lower Bounds

Given a pulldown sequence, one can imagine splitting it in the middle. The first part of the sequence constructs a heap that has about half as many keys as the finished heap. Performing the pulldowns specified by the second part of the sequence adds one more row to the heap created by the first part. The results of this section are based on the observation that the keys in the half-sized heap are small and thus belong near

the bottom of the finished heap. This means that a substantial fraction of the keys in the half-sized heap must be pulled down in the second half of the pulldown sequence. From this it follows that the second part of the pulldown sequence must pull down many keys from near the bottom of the heap, and many keys with large numbers of 1's in their binary representations. This leads to tight lower bounds. This idea was developed independently by Sedgewick [23, 24] and Wegener [26], who both used it to prove the following theorem:

THEOREM 4.1: For any heap H, $B_W(H)$ is at least $\frac{1}{2}N \lg N + O(N)$.

Proof: Starting with a heap on one key, any heap on N keys can be obtained by successively doubling the size of the heap, inserting single pulldowns between doubling operations when appropriate. The lemma that follows gives a lower bound on the cost of doubling a heap's size. The theorem results from summing this bound as the heap size is successively doubled. □

LEMMA 4.1: Given a heap on N keys, any pulldown sequence expanding the heap to $2N$ keys must increase the B_W cost of the heap by at least $\frac{1}{2}N \lg N - 2N$.

Proof: It will be useful to partition the keys of the expanded heap into two classes: the keys with values from 1 to N will be called the old keys while keys with values from $N+1$ to $2N$ will be called the new keys. Likewise, positions 1 to N of the expanded heap will be referred to as the old positions, while positions beyond N will be referred to as the new positions. Each of the N pulldowns required to augment the heap can be classified according to whether it pulls down a new key or an old key.

The new keys in the heap form a subtree of the heap at every stage of the pulldown process. Each pulldown adds a leaf to this subtree, increasing its size by one. Any leaf added in an old position must be the result of pulling down an old key. The B_W cost of pulling down an old key is bounded below by the level of the leaf created since the leaf is along the path from the key being pulled down to the root of the heap. This implies that the B_W cost of all pulldowns of old keys is bounded below by the internal path length of the subtree formed by new keys in old positions.

Now the subtree formed by the new keys has all its internal nodes in old positions, so in the expanded heap there are at least $\lfloor N/2 \rfloor$ new keys in old positions. The internal path length of any binary tree on $\lfloor N/2 \rfloor$ nodes is at least $\frac{1}{2}N \lg N - 2N$. This completes proof of the lemma. □

COROLLARY: For any heap H, $B_F(H)$ is at most $\frac{1}{2}N \lg N + O(N)$.

Proof: This follows immediately from Theorem 4.1 and part 5 of Lemma 2.1. □

The argument used to prove the previous theorem applies in a slightly more complicated form to the worst and best case behavior of C_W and C_F. The following theorem gives a lower bound on the value of C_W that will be shown to be tight later in this chapter.

THEOREM 4.2: For any heap H, $C_W(H)$ is at least $\frac{1}{4}N \log N + O(N\sqrt{\log N})$.

Proof: As in the proof of the previous theorem, this theorem follows from summing the lower bound given by the following lemma as the heap size is repeatedly doubled. □

LEMMA 4.2: Given a heap on N keys, any pulldown sequence expanding the heap to $2N$ keys must augment the C_W cost of the heap by at least $\frac{1}{4}N \lg N + O(N\sqrt{\log N})$.

Proof: As in the proof of Lemma 4.1, the keys in $[1, N]$ will be called the old keys while the keys in $[N+1, 2N]$ will be called the new keys. Also, positions 1 to N of the heap will be referred to as the old positions while positions numbered above N will be referred to as the new positions.

At all stages of the pulldown process, the new keys of the heap form a subtree whose root is the root of the heap. Each pulldown adds a leaf to this subtree. As was the case in the proof of Lemma 4.1, the cost of the N pulldowns is bounded below by the internal path length of the part of this subtree that occupies old positions. It was demonstrated in the proof of Lemma 4.1 that there must be at least $\lfloor N/2 \rfloor$ new keys in old positions. A lower bound on the C_W costs of the N pulldowns can thus be obtained by summing the C_W costs of the $\lfloor N/2 \rfloor$ lowest cost positions in the top $\lceil \lg(N+1) \rceil$ levels of the heap.

Now the number of positions in the top $\lceil \lg(N+1) \rceil$ levels having C_W cost equal to i is at most:

$$\sum_{j \le \lceil \lg(N+1) \rceil} \binom{j-1}{i} = \binom{\lceil \lg(N+1) \rceil}{i+1}.$$

Let k be the largest integer such that $\lfloor N/2 \rfloor$ bounds the number of positions in the top $\lceil \lg(N+1) \rceil$ levels with C_W cost at most k. That is:

$$\sum_{j \le k} \binom{\lceil \lg(N+1) \rceil}{j+1} \le \left\lfloor \frac{N}{2} \right\rfloor < \sum_{j \le k+1} \binom{\lceil \lg(N+1) \rceil}{j+1}$$

Then a lower bound on the C_W cost of the N pulldowns is given by

$$\begin{aligned}
&\sum_{j \le k} j \binom{\lceil \lg(N+1) \rceil}{j+1} \\
&= \sum_{j \le k} (j+1) \binom{\lceil \lg(N+1) \rceil}{j+1} - \sum_{j \le k} \binom{\lceil \lg(N+1) \rceil}{j+1} \\
&= \lceil \lg(N+1) \rceil \sum_{j \le k} \binom{\lceil \lg(N+1) \rceil - 1}{j} - \sum_{j \le k} \binom{\lceil \lg(N+1) \rceil}{j+1} \\
&> \frac{\lceil \lg(N+1) \rceil}{2} \left[\sum_{j \le k} \binom{\lceil \lg(N+1) \rceil}{j+1} - \binom{\lceil \lg(N+1) \rceil - 1}{k+1} \right] - \sum_{j \le k} \binom{\lceil \lg(N+1) \rceil}{j+1} \\
&= \frac{\lceil \lg(N+1) \rceil}{2} \sum_{j \le k+1} \binom{\lceil \lg(N+1) \rceil}{j+1} - \sum_{j \le k} \binom{\lceil \lg(N+1) \rceil}{j+1} \\
&\qquad - \frac{\lceil \lg(N+1) \rceil}{2} \left[\binom{\lceil \lg(N+1) \rceil}{k+2} + \binom{\lceil \lg(N+1) \rceil - 1}{k+1} \right] \\
&> \frac{\lceil \lg(N+1) \rceil}{2} \left\lfloor \frac{N}{2} \right\rfloor + O(N) + O(N\sqrt{\log N}).
\end{aligned}$$

This completes proof of the lemma. □

COROLLARY: For any heap H, $C_F(H)$ is at least $\frac{1}{4} N \log N + O(N\sqrt{\log N})$.

Proof: This follows immediately from Theorem 4.2 and part 6 of Lemma 2.1. □

THEOREM 4.3: Given a heap H, $B_W(H) - C_W(H)$ is at least $\frac{1}{4} N \log N + O(N\sqrt{\log N})$.

Proof: This theorem is proved by the same reasoning that was used to prove Theorem 4.2 since the two theorems are essentially the same. Given a pulldown sequence

$P = \{p_i\}_{i=2}^{N}$, Lemma 4.2 gives a lower bound on the number of 1 bits in the binary representations of the p_i. Since $B_W(H(P)) - C_W(H(P))$ is within $O(N)$ of the total number of 0 bits in the binary representations of the p_i, this theorem is just a symmetric version of Theorem 4.2. □

COROLLARY: For any heap H, $C_W(H)$ is at most $\frac{3}{4}N \log N + O(N\sqrt{\log N})$.

Proof: This follows immediately from Theorem 4.3 and part 7 of Lemma 2.1. □

COROLLARY: For any heap H, $C_F(H)$ is at most $\frac{3}{4}N \log N + O(N\sqrt{\log N})$.

Proof: When sorting any given heap, line 11 of Williams's Heapsort performs a subset of the comparisons performed by line 11 of Floyd's Heapsort. Floyd's Heapsort must thus find $A[k] \geq A[k+1]$ at least as many times as the $\frac{1}{4}N \log N + O(N\sqrt{\log N})$ times mandated for Williams's Heapsort by Theorem 4.3. Since Floyd's Heapsort executes line 11 $N \lg N + O(N)$ times, the corollary holds. □

The following table summarizes for easy reference the bounds presented in this section and in Lemma 2.1. The remainder of the chapter shows that, asymptotically, these bounds are tight.

	Lower Bound	Upper Bound
$B_W(H)$	$\frac{1}{2}N \lg N + O(N)$	$N \lg N$
$C_W(H)$	$\frac{1}{4}N \lg N + O(N\sqrt{\log N})$	$\frac{3}{4}N \lg N + O(N\sqrt{\log N})$
$B_F(H)$	0	$\frac{1}{2}N \lg N + O(N)$
$C_F(H)$	$\frac{1}{4}N \lg N + O(N\sqrt{\log N})$	$\frac{3}{4}N \lg N + O(N\sqrt{\log N})$

4.2 The Best Case of Williams's Heapsort

Among the bounds presented in the last section were bounds on the best cases of $B_W(H)$ and $C_W(H)$. These bounds showed that every time a heap's size is doubled by pull-downs, a minimum of about half of the small keys must be pulled down. While the large keys were assumed to be pulled down for free, restrictions on how many small keys could be pulled down from low cost positions forced some small keys to be pulled from high

cost positions. These high cost pulldowns of small keys were the sole contributors to the best case cost of Williams's Heapsort. Sedgewick [23, 24] proposed a pulldown sequence that alternates pulldowns of sets of small keys from the bottom of the heap with sets of large keys from the top of the heap. This constructs a heap on $N = 2^n - 1$ keys that has asymptotically the best possible B_W cost, and the worst possible B_F cost. Fleischer et al have independently developed similar results [11, 12]. It turns out that Sedgewick's construction also illustrates asymptotically the best possible C_W cost, and the worst possible C_F cost. This section presents Sedgewick's construction as a best case input for Williams's Heapsort. The following two sections present two variations on this construction that exhibit respectively the worst possible asymptotic behavior for Williams's Heapsort and the best possible asymptotic behavior for Floyd's Heapsort. The section following these re-examines Sedgewick's construction as a worst case input for Floyd's Heapsort.

4.2.1 Details of the Construction

For Williams's Heapsort, all pulldowns from near the top of the heap contribute little to the cost of sorting the heap; a pulldown from the top is reversed by moving a key down only a few levels, and thus has low B_W and C_W costs. Pulldowns from near the top of the heap are clearly beneficial to the best case of Williams's Heapsort. In practice, however, one would expect the keys at the top of the heap to be among the largest in the heap since they are required to be larger than all keys beneath them. To pull down such large keys, it is clear that a best case construction will have to provide even larger keys at the bottom of the heap to serve as parents in the pulldown process. The best case construction to be considered here provides these keys by pulling down a pocket of small keys at the bottom of the heap. As the small keys are pulled down, large keys take their places. Once a pocket of large keys has been created, it can provide parents to keys pulled from the top of the heap.

Figure 4.1 shows schematically how this process is made to work. The figure shows a heap on $2^\ell - 1$ keys for some ℓ which will be more completely specified later. Since

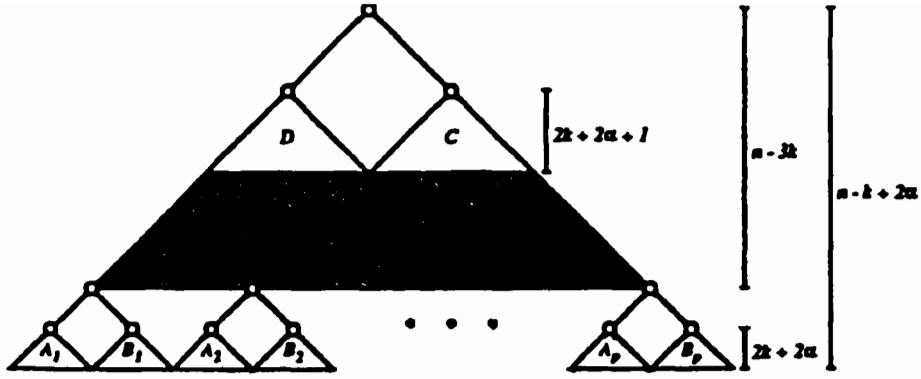

Figure 4.1: Schematic of the Williams's Heapsort best case construction.

the bottom row of the heap is full, pulldowns will add new key positions starting at the bottom left of the heap. The keys of a fixed level near the bottom of the heap are chosen as roots of small subheaps, labeled alternately A_i and B_i. The keys of each A_i are initially larger than the keys of the corresponding B_i. Given a heap of this form, the keys in B_1 can be pulled down to add a new level to A_1. These pulldowns cause a stream of new keys to flow from the root into B_1. Almost all keys entering B_1 are new to the heap since the start of this sequence of pulldowns; the only exceptions are those keys that were originally on the path from B_1 up to the root. The keys of B_1 are now large enough to pull down the largest keys in subheap C, where C is composed of enough levels below the right child of the root to supply a new level to B_1. A new level can be added to the left half of the heap by iterating this process of pulling down the keys of B_i to add a level to A_i and then pulling down the keys of C to add a new level to B_i. The same process adds a level to the right half of the heap when C is replaced by D.

A numerical example of this process will illustrate how it works, as well as some of the difficulties that must be faced when it is presented in detail. Figure 4.2 shows the heap that is constructed by the pulldown sequence $\{1,2,3,2,3,5,4,3,7,6,6,8,7,12\}$.

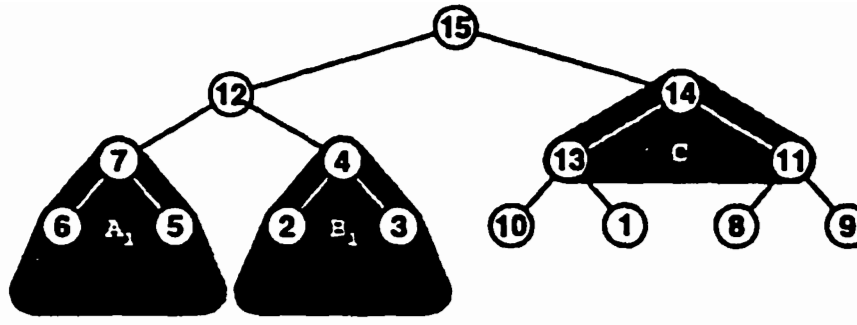

Figure 4.2: A heap like that of the schematic.

This heap illustrates the major features of the schematic Figure 4.1. Subheaps A_1 and B_1 are rooted at A[4] and A[5] respectively, and the keys of A_1 are all larger than any key in B_1. C is rooted at A[3]. Space limitations have forced the bottom level of C to overlap what should be the roots of A_2 and B_2; in the construction that will be described below, the heap must be large enough that C and D never overlap the A_i and B_i.

Figure 4.3 extends the heap of Figure 4.2 by performing pulldowns from positions $\{9, 11, 11, 10, 11, 3, 6, 7\}$. This illustrates the process of constructing a low cost heap starting from a heap like that of Figure 4.1. The first pulldown is of a key in A_1. Such a pulldown is necessary since there are not enough keys in B_1 to add a complete bottom row to A_1. The construction to be detailed below will perform such initial pulldowns from the A_i. The following three pulldowns empty B_1 of the keys that it contained in Figure 4.2. After these pulldowns have been performed, 15 is the only key in B_1 that was in the heap in Figure 4.2. Next, the smallest key of B_1 is pulled down. This pulldown is needed not only because there are not enough keys in C to add a new row to B_1, but because it is necessary to pull down all keys that were along the path from B_1 to the root of the heap in Figure 4.2 — in this illustration there is only one such key and it is larger than the keys in C, but it is easy to see that in a heap with more levels,

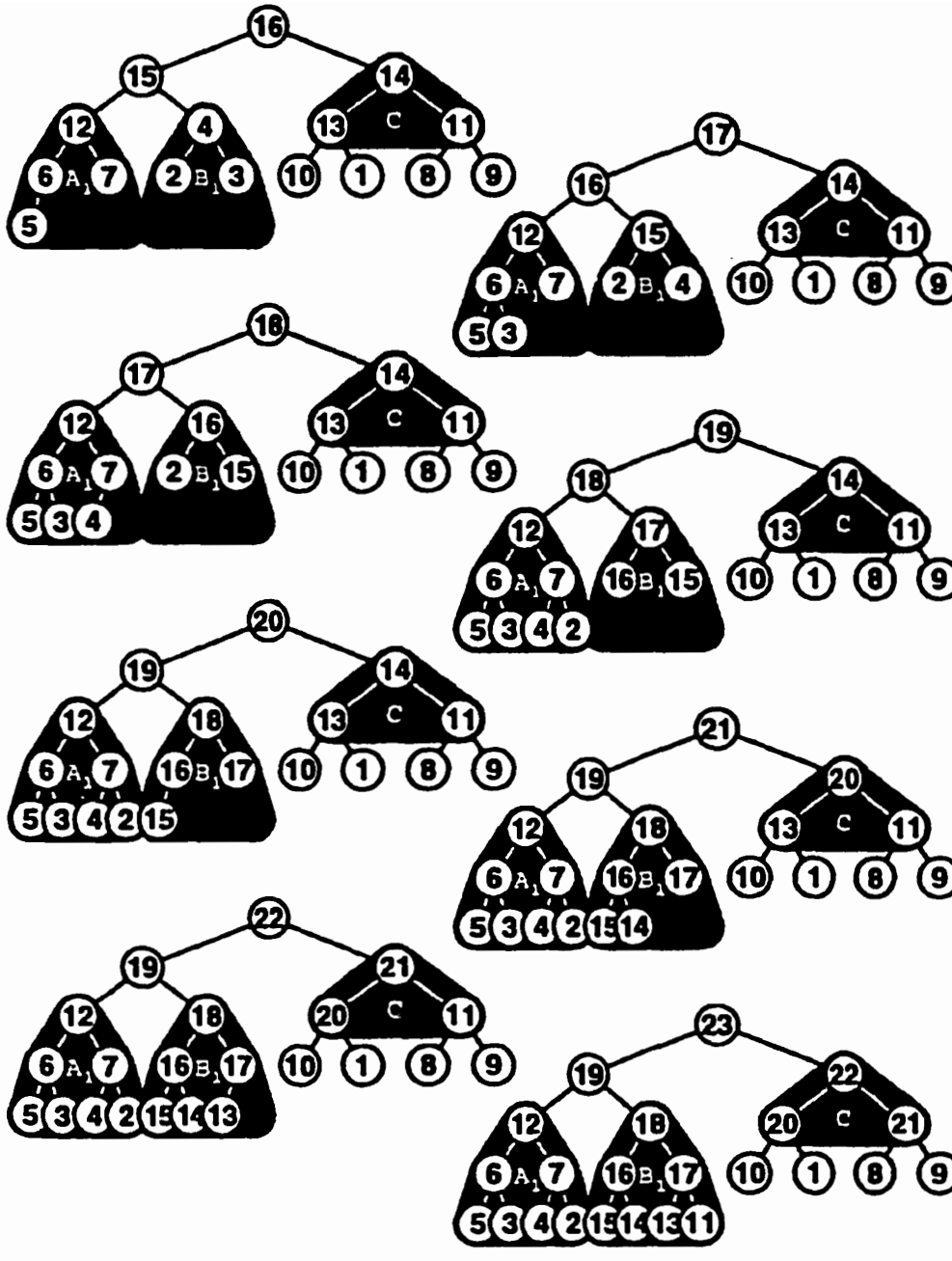

Figure 4.3: Constructing a low cost heap.

there could be many such keys and they could be smaller than the keys in C. After pulling down the key from B_1, none of the keys above the new bottom level of B_1 were in the heap in Figure 4.2. This means that all keys in the top levels of B_1 are larger than any key of C, so the last three pulldowns can empty C of the keys it contained in Figure 4.2.

The process that has been described, of pulling down the keys in B_i to add a level to A_i and then pulling down keys near the top to add a level to B_i depends upon the initial condition that the keys in A_i are larger than those in B_i, and adds just one level to the heap. Unfortunately, this condition does not continue to hold after the new level has been added. If, however, the keys of C were initially larger than those of A_2, the addition of the new level causes a complementary condition to hold, where the keys of B_i are larger than those of A_{i+1}. Ignoring, for a moment, what happens to A_1 and the last B_i, this complementary condition allows the addition of another level to the heap in a manner analogous to the above process, and at the same time restores the original relations on the A_i and B_i. The construction as a whole is thus a two part process. First, a "seed heap" is created satisfying the condition that the keys in each A_i are larger than the keys in the corresponding B_i. Second, the seed heap is expanded to the final size by adding levels in pairs.

Figure 4.3 illustrated the basic idea of alternating pulldowns from low positions with pulldowns from high positions; it also showed why a few additional pulldowns will be required when the construction is presented in detail. These additional pulldowns must be handled with care; they must not contribute to the asymptotic cost of constructing the low cost heap. Prior to pulling the keys of B_i into A_i, it is expected that $\Omega(\lg N)$ keys will have to be pulled down from A_i. In order for the cost of these pulldowns to be negligible with respect to the cost of pulling the keys of B_i into A_i, the sizes of the A_i and B_i will be taken to be $\Omega(\log^2 N)$. The costs of initial pulldowns from B_i prior to pulling the keys of C into B_i are also absorbed with this choice of size for the A_i and B_i.

The size of the seed heap must be chosen with similar care. The seed heap must

contain few enough keys that its construction cost is negligible, and yet enough keys that the A_i and B_i can each start with $\Omega(\log^2 N)$ keys. The seed heap will thus be taken to contain $\Theta(N/\log N)$ keys.

The sizes of the seed heap and of the A_i and B_i are correctly determined by the following choice of parameters: Set $k = 2\lceil \frac{1}{2}\log n \rceil$. Start with a seed heap on $2^{n-k} - 1$ keys, and fix the roots of the A_i and B_i to be the keys of level $n - 3k + 1$ where the root of the heap is level 1. Formally, this means that the root of A_i is $A[2p + 2i - 2]$ and the root of B_i is $A[2p + 2i - 1]$, where $p = 2^{n-3k-1}$ is the number of A_i (and thus also of B_i). The seed heap is then expanded to the full heap by $k/2$ passes, each of which adds two levels to the heap in two phases, named Phase AB and Phase BA. Let α be a variable that records the current pass of the construction, and initialize it to zero. Then C and D can be defined to be the $2(k + \alpha) + 1$ level subtrees rooted at $A[3]$ and $A[2]$ respectively; it is necessary that C and D each have $2(k + \alpha) + 1$ levels since they must contain enough keys to add new levels to the A_i in Phase BA. Note that the the roots of all subtrees remain fixed so that the subtrees grow as α grows. Note also that $n > 6k$ is a necessary and sufficient condition for C and D not to overlap the roots of the A_i and B_i. For an illustration of these quantities and how they determine the strucure of the heap, refer back to Figure 4.1.

Let $t = 2n\alpha + 1$. This quantitiy describes the number of keys that must be pulled down from A_i before pulling the keys of B_i into A_i (or the number of keys that must be pulled down within B_i before pulling keys from the top into B_i). These keys represent a certain amount of slop that does not satisfy requirements like the condition that the keys of A_i be larger than the keys of B_i. Since this slop accumulates, it is necessary for t to be proportional to α. Finally, let $L(H, x)$ be the largest x keys of the subtree H and $S(H, x)$ be the smallest x keys of the subtree H; let $L(H, x) > S(H, x)$ imply that the largest x keys of H are all larger than any of the smallest x keys of H. For example, $L(B_1, 6) > S(A_1, 7)$ holds for the final heap of Figure 4.3.

These definitions permit Figures 4.4 and 4.5 to describe in detail the construction of a low cost heap. The instruction "Pull down $S(H, x)$" means to pull down all keys

I. Build a seed heap on $2^{n-k}-1$ keys that satisfies the following inequalities:

1. $L(A_i, 2^{2k}-1) > S(B_i, 2^{2k}-1)$ for all i.
2. $L(C, 2^{2k}-n-1) > S(A_2, 2^{2k}-1)$.

II. Complete the heap by executing:

For $\alpha = 0$ to $k/2-1$ execute the code in Figure 4.5.

Figure 4.4: Construction for Williams's Heapsort best case.

Action Performed	Bw Upper Bound	Cw Upper Bound
Phase AB:		
(*i*) For $i = 1$ to $p/2$ do:		
a) Pull down $S(A_i, t)$	n	$bit(i-1)+3k$
b) Pull down $S(B_i, 2^{2k+2\alpha}-t)$	n	$bit(i-1)+3k$
c) Pull down $S(B_i, t+n)$	n	$bit(i-1)+3k$
d) Pull down $L(C, 2^{2k+2\alpha}-t-n)$	$3k$	$3k$
(*ii*) For $i = p/2+1$ to p do:		
a-d) As above, replacing C by D		
Phase BA:		
(*i*) Pull down $S(A_1, 1)$, $2^{2k+2\alpha+1}$ times	n	$3k$
(*ii*) For $i = 1$ to $p/2-1$ do:		
a) Pull down $S(B_i, t+n)$	n	$bit(i-1)+3k$
b) Pull down $S(A_{i+1}, 2^{2k+2\alpha+1}-t-n)$	n	$bit(i)+3k$
c) Pull down $S(A_{i+1}, t+2n)$	n	$bit(i)+3k$
d) Pull down $L(C, 2^{2k+2\alpha+1}-t-2n)$	$3k$	$3k$
(*iii*) For $i = p/2$ to $p-1$ do:		
a-d) As above, replacing C by D		
(*iv*)		
a) Pull down $S(B_p, t+n)$	n	$bit(p-1)+3k$
b) Pull down $S(A_1, 2^{2k+2\alpha+1}-t-n)$	n	$3k$

Figure 4.5: The Williams's Heapsort best case main loop, with cost bounds.

that were in $S(H,x)$ at the beginning of the instruction. It will be assumed that these pulldowns are performed on the keys in increasing order, so that keys of the A_i and B_i are pulled down from one of the bottom two levels of the heap. "Pull down $L(H,x)$" is defined similarly. The costs given with each step of the construction are upper bounds on the *average* amount of work performed by the individual pulldowns of that step.

THEOREM 4.4: Figures 4.4 and 4.5 specify a pulldown sequence P with $B_W(H(P)) = \frac{1}{2}N\lg N + O(N\log\log N)$ and $C_W(H(P)) = \frac{1}{4}N\lg N + O(N\log\log N)$.

Proof: The next subsection will prove that the pulldown sequence described in Figures 4.4 and 4.5 is actually legitimate in the sense that no key is pulled down to be a child of a key smaller than itself. The following subsection will show that the claimed cost bounds hold. □

4.2.2 Correctness of the Construction

This section presents an inductive argument that shows that Figures 4.4 and 4.5 specify a legitimate pulldown sequence. As a base case, it is shown that it is possible to create a seed heap of the type required by Figure 4.4. The conditions satisfied by the seed heap are then generalized, and it is shown that these generalized conditions both permit and are preserved by one execution of the code of Figure 4.5. From this it follows that it is possible to execute multiple iterations of Figure 4.5 as is required to construct the final heap from the seed heap.

It is very easy to see that a seed heap on $2^{n-k}-1$ keys can be constructed to satisfy the conditions of Figure 4.4. Simply divide the keys to be contained by the heap into three intervals: $[1, p(2^{2k}-1)]$, $[p(2^{2k}-1)+1, 2p(2^{2k}-1)]$, and $[2p(2^{2k}-1)+1, 2^{n-k}-1]$. The integers in $[1, p(2^{2k}-1)]$ can be assigned to fill the B_i and arranged in heap order within each B_i. Likewise, the keys in $[p(2^{2k}-1)+1, 2p(2^{2k}-1)]$ can be placed in the A_i in heap order. The remaining $2^{n-3k}-1$ keys in $[2p(2^{2k}-1)+1, 2^{n-k}-1]$ can be arranged in heap order in the positions above the A_i and B_i. Observe:

- The roots of the A_i and B_i are from the first two intervals and are thus smaller

than their parents, which come from the last interval. Since keys within a given subtree are arranged in heap order, it follows that the whole arrangement obeys heap order as well.

- Since the keys in A_i are drawn from the second interval while those in B_i are drawn from the first, it follows that $L(A_i, 2^{2k}-1) > S(B_i, 2^{2k}-1)$ for all i.
- Likewise, $L(C, 2^{2k}-n-1) > S(A_2, 2^{2k}-1)$ because the keys in A_2 are drawn from the second interval while those in C come from the last interval.

These three observations demonstrate that this arrangement of integers in $[1, 2^{n-k}-1]$ is a heap of the form required by Figure 4.4. The time required to sort any heap on $2^{n-k}-1$ keys is $O(N)$, so no attention need be paid to the cost of the pulldown sequence that constructs this heap.

The conditions that Figure 4.4 requires of the seed heap are just the special case $\alpha = 0$ of the following conditions, which will henceforth be referred to as the *pass invariants*:

- $L(A_i, 2^{2(k+\alpha)}-t) > S(B_i, 2^{2(k+\alpha)}-t)$ for all i. (4.1)
- $L(C, 2^{2(k+\alpha)}-t-n) > L(A_2, 2^{2(k+\alpha)}-1)$. (4.2)

In the remainder of this section, it will be proved that if a heap satisfies the pass invariants, then it is possible to execute one pass of the code in Figure 4.5; the resulting heap also satisfies the pass invariants upon incrementing α. This proof will examine each step of the code in turn, establishing the conditions that permit the step to be executed and stating the results of its execution.

The code in Figure 4.5 begins with Phase AB which adds a level to the heap by pulling the keys of each B_i into A_i and then pulling the keys of C or D into B_i. Phase AB begins with step AB(i) which has two purposes: it adds a new level to the left half of the heap, and for $i < p/2$, it causes most of the keys in B_i to be larger than most of the keys in A_{i+1}. This latter purpose is accomplished in part by guaranteeing at the beginning of

the ith iteration of step AB(i; a), that the largest keys in C (which will end up in B_i) are larger than the keys in A_{i+1}; put formally, $L(C, 2^{2(k+\alpha)} - t - n) > L(A_{i+1}, 2^{2(k+\alpha)} - 1)$. This condition holds prior to the first iteration by (4.2) of the pass invariants and is reestablished at the end of each iteration by step AB(i; d). The following is an enumeration of the effects of each of the steps that make up an iteration of step AB(i):

- At the end of step AB(i; a), all keys at or above level $2(k+\alpha)$ of A_i are larger than any key of $S(B_i, 2^{2(k+\alpha)} - t)$. This follows from three observations. First, at the beginning of this step, (4.1) of the pass invariants implies that there are at most $t-1$ keys in A_i that can be smaller than any key in $S(B_i, 2^{2(k+\alpha)} - t)$. Second, this step pulls all of these $t-1$ keys down to the new bottom level of A_i. Third, those keys entering A_i in this step are larger than any key that was in $L(A_i, 2^{2(k+\alpha)} - 1)$ to begin with. Since the keys being pulled down remain within A_i, it is clear that the specified pulldowns can be performed.

- Step AB(i; b) completes the new bottom level of A_i. It also makes sure that fewer than $t+n$ keys in B_i can be smaller than any key of C — formally this means that $L(B_i, 2^{2(k+\alpha)} - t - n) > L(C, 2^{2(k+\alpha)} - t - n)$. At the end of this step, the only keys in B_i that could fail to be larger than those in C are the $t-1$ keys that were in B_i at the beginning of the step and which were not pulled down, and the $n-3k$ keys that were along the path from the root of B_i to the root of the heap at the beginning of the step; the remaining $2^{2(k+\alpha)} - t - n + 3k$ keys in B_i at the end of this step are new to the heap since the beginning of the step and the inequality follows. The results of the last step ensure that the pulldowns required by this step can be performed.

- By the end of step AB(i; c), every key of B_i at or above level $2(k+\alpha)$ is larger than any key of $L(C, 2^{2(k+\alpha)} - t - n)$. This step does for B_i what the analogous step AB(i; a) does for A_i. The reasons for its correctness are also the same as those for AB(i; a).

- Step AB(i; d) completes the new bottom level of B_i. This step also establishes

$L(C, 2^{2(k+\alpha)} - t - n) > L(A_{i+2}, 2^{2(k+\alpha)} - 1)$. This inequality follows from the observation that this step removes the largest $2^{2(k+\alpha)} - t - n$ keys from C, and that the keys that take their places are new to the heap since the beginning of step AB(i; c). The results of the previous step guarantee that the pulldowns in this step can be performed.

Step AB(i) begins a new level of the heap, and step AB(ii) finishes it. Steps AB(i) and AB(ii) operate in essentially the same way, except that AB(ii) uses D as the source of keys to pull from the top of the heap; C is unsuitable for this purpose since it will be disturbed by pulldowns of keys from the A_i and B_i below it. It should be noted that in step AB(i), the pulldowns from the A_i and B_i purge D of the keys it contained at the beginning of that step. Since D thus contains keys that are newer than any key in $A_{p/2+2}$, it follows that $L(D, 2^{2(k+\alpha)} - t - n) > L(A_{p/2+2}, 2^{2(k+\alpha)} - 1)$ at the beginning of step AB(ii); $L(D, 2^{2(k+\alpha)} - t - n) > L(A_{i+1}, 2^{2(k+\alpha)} - 1)$ will always hold prior to iteration i of step AB(ii; a).

It has been shown that Phase AB of the construction is capable of adding one level to a heap satisfying the pass invariants. It is now necessary to examine the structure of the heap following the execution of Phase AB. In particular, the following complements to the pass invariants must be shown to hold:

- $L(B_i, 2^{2(k+\alpha)+1} - t - n) > S(A_{i+1}, 2^{2(k+\alpha)+1} - t - n)$ for $i < p$. (4.3)
- $L(B_p, 2^{2(k+\alpha)+1} - t - n) > S(A_1, 2^{2(k+\alpha)+1} - t - n)$. (4.4)
- $L(C, 2^{2(k+\alpha)+1} - t - 2n) > L(B_2, 2^{2(k+\alpha)+1} - 1)$. (4.5)

These conditions permit Phase BA to add another level to the heap and to restore the pass invariants with α incremented.

To prove (4.3), it is necessary to consider the origins of the keys in B_i and A_{i+1}. Fix $i < p/2$. Of the keys in B_i at the end of Phase AB, $2^{2(k+\alpha)} + 3k$ entered the heap during the ith iteration of steps AB(i; b, c). Another $2^{2(k+\alpha)} - t - n$ keys came to B_i

from C during the ith iteration of step AB(i; d). These keys from C were smaller than any of the $2^{2(k+a)} + 3k$ new keys in B_i at the beginning of step AB(i; d), but they were larger than all $2^{2(k+a)} - 1$ keys in A_{i+1} by the inequality $L(C, 2^{2(k+a)} - t - n) > L(A_{i+1}, 2^{2(k+a)} - 1)$ that held at the beginning of the ith iteration of step AB(i; a). Finally, in the $i+1$st iteration of step AB(i; b), A_{i+1} acquired $2^{2(k+a)} - t$ keys from B_{i+1} that were smaller than most of the keys that had been in A_{i+1} prior to Phase AB. Thus at the end of Phase AB, B_i contains $2^{2(k+a)+1} - t - n + 3k$ keys that are either new or from C, and all of these keys are larger than at least $2^{2(k+a)+1} - t - 1$ keys of A_{i+1} that were in A_{i+1} and B_{i+1} at the beginning of Phase AB. Inequality (4.3) follows immediately for $i < p/2$. As i makes the transition from the left side of the heap to the right side of the heap, (4.3) continues to hold by analogous reasoning as step AB(ii) replaces step AB(i) and D replaces C.

Inequality (4.4) holds by the same reasoning used to prove (4.3). Of the keys in B_p, at least $2^{2(k+a)+1} - t - n + 3k$ are either new to the heap in steps AB(ii; b, c) or from D (also new to the heap since the beginning of Phase AB), while at least $2^{2(k+a)+1} - t - 1$ keys of A_1 were already in the heap at the beginning of Phase AB.

Inequality (4.5) holds because step AB(ii) leaves B_2 untouched, while the many pulldowns from the A_i and B_i, $i > p/2$, refill C with new keys.

Inequalities (4.3), (4.4), and (4.5) permit Phase BA to add another level to the heap and to restore the pass invariants (4.1) and (4.2). Step BA(i) begins that process by adding the first row to A_1. Normally, Phase BA attempts to pull keys of A_i into B_{i-1}. In this particular case of $i = 1$ the appropriate B_0 does not exist, so the job is done by pulling down the smallest key of A_1, $2^{2(k+a)+1}$ times. These pulldowns are possible since the smallest key in A_1 is always smaller than any possible parent in A_1. Phase BA exhibits its normal mode of operation in steps BA(ii, iii), which add a new level to all but B_p. In a manner analogous to step AB(i), the inequalities $L(C, 2^{2(k+a)+1} - t - 2n) > L(B_{i+1}, 2^{2(k+a)+1} - 1)$ and $L(D, 2^{2(k+a)+1} - t - 2n) > L(B_{i+1}, 2^{2(k+a)+1} - 1)$ always hold prior to the ith iterations of steps BA(ii; a) and BA(iii; a) respectively. The incremental proof of correctness for steps BA(ii, iii) is the same as the the one given

for step AB(i) and is not repeated here.

To finish adding its level to the heap, Phase BA must add a new level to B_p. It does so in step BA(iv) by pulling into B_p those keys of A_1 that would have been pulled into B_0 if such a subheap existed. This has the nice side effect of eliminating the small keys from A_1 while adding very few large keys to B_p.

- Step BA(iv; a) is analogous to steps BA(ii, iii; a). When it finishes, every key at or above B_i's level $2(k+a)+1$ is larger than any key of $S(A_1, 2^{2(k+a)+1} - t - n)$. This holds for reasons seen previously: this step pulls the smallest $t+n$ keys of B_p to the new bottom level. By (4.4) above, the remaining $2^{2(k+a)+1} - t - n - 1$ keys originally in B_p are all larger than any key in $S(A_1, 2^{2(k+a)+1} - t - n)$, while the keys that enter B_p in this step are even larger than the other keys in B_p. (Note that step BA(i) did not change the composition of $S(A_1, 2^{2(k+a)+1} - t - n)$.)

- Step BA(iv; b) completes the new level of B_p and ensures that at the end of Phase BA, the keys in $L(A_1, 2^{2(k+a)+2} - t - 2n)$ are new to the heap since the beginning of Phase BA. The only keys of A_1 that did not enter the heap in Phase BA are the $n - 3k$ keys that were along the path from the root of A_1 to the root of the heap at the beginning of the phase, and the $n + t - 1$ keys that were in A_1 at the beginning of the phase but do not get pulled down in this step.

It now remains to show that Phase BA restores the pass invariants. Part (4.1) of the pass invariants holds for $i > 1$ by the same arguments used to prove inequality (4.3). Most keys in A_i come from two sources, they are either new to the heap in steps BA(ii, iii; b, c), or they are slightly smaller and come from C or D. In either case, these keys in A_i are larger than those in B_i, most of which were in B_i and $A_{i+1 \pmod p}$ at the beginning of Phase BA. For $i = 1$, (4.1) of the pass invariants holds by the results of step BA(iv); at the end of Phase BA, the keys in $L(A_1, 2^{2(k+a)+2} - t - 2n)$ are all new to the heap since the beginning of the phase, while the keys in $S(B_1, 2^{2(k+a)+2} - t - 2n)$ were in B_1 and A_2 at the beginning of the phase.

Part (4.2) of the pass invariants holds by the same reasoning that has been used before. Steps BA(*iii*, *iv*) leave A_2 untouched while filling C with new, large keys.

This concludes the proof that a single execution of the code in Figure 4.5 takes a heap on $2^{n-k+2\alpha}-1$ keys that satisfies the pass invariants, and transforms it into a heap on $2^{n-k+2(\alpha+1)}-1$ keys that also satisfies the pass invariants once α is incremented. This in turn completes the inductive proof that a heap on 2^n-1 keys can be built from a seed heap on $2^{n-k}-1$ keys by the code in Figure 4.5.

4.2.3 Costs of the Construction

So far, it has been shown that it is possible to execute the code given in Figures 4.4 and 4.5 to produce a valid pulldown sequence. Of course, the purpose of this code is to demonstrate a heap that causes Williams's Heapsort to exhibit its optimal asymptotic behavior. In order to achieve this purpose, it is necessary to bound the costs of the heap produced by the construction. This section provides such bounds by justifying and summing the cost bounds given with the code in Figure 4.5. This results in upper bounds of $\frac{1}{2}N\lg N+O(N\log\log N)$ and $\frac{1}{4}N\lg N+O(N\log\log N)$ on the B_W and C_W costs, respectively, of the construction.

The bounds given with each step of Figure 4.5 are upper bounds on the average cost per pulldown of one execution of that step. So, for example, the upper bound n given for the B_W cost of step AB(i; a) implies that the average cost of a pulldown from $S(A_i, t)$ is at most n. It is easy to see that the bounds on the B_W cost are correct for all steps. The final size of the heap is $N=2^n-1$ nodes, so no key can ever be pulled down from a level numbered greater than n. This explains the B_W cost bounds on pulldowns in all steps except for steps AB(*i*, *ii*; *d*) and BA(*ii*, *iii*; *d*). In the case of these exceptions, all pulldowns are from C or D. Since α never exceeds $k/2-1$, and since the heights of C and D never exceed $2(k+\alpha)+1$, it follows that no key of C or D is ever pulled from a level numbered greater than $3k$. This explains the remaining bounds on B_W cost.

The bounds on the C_W cost are not much harder to prove. Fix i and let x be any

position in A_i or B_i. Then the binary representation of x begins with a "1" followed by the $n-3k-1$ bit binary representation of $i-1$ since A$[p+i-1]$ is the parent of the roots of A_i and B_i and thus an ancestor of x. Beyond this, the binary representation of x contains at most $3k$ bits since $x < 2^n$. This gives an upper bound of $bit(i-1)+3k$ on the C_W cost of pulling down the key in position x. This implies the correctness of all bounds given in Figure 4.5 except for the bounds on steps AB(*i, ii; d*) and BA(*ii, iii; d*). All pulldowns in these steps are from C or D, which contain no positions numbered greater than $2^{3k}-1$; it follows that no pulldown in these steps has C_W cost greater than $3k$.

Having justified the bounds given in Figure 4.5, it remains to sum these bounds over all executions of each step. The total B_W cost of steps AB(*i, ii; a, b*), BA(*ii, iii; a, b*), and BA(*iv; a, b*) is bounded above by:

$$\begin{aligned}
&\sum_{\alpha=0}^{k/2-1}\left[\sum_{i=1}^{p}((t)+(2^{2k+2\alpha}-t))n+\sum_{i=1}^{p-1}((t+n)+(2^{2k+2\alpha+1}-t-n))n\right.\\
&\qquad\left.+((t+n)+(2^{2k+2\alpha+1}-t-n))n\right]\\
&\quad=\ np\sum_{\alpha=0}^{k/2-1}(2^{2k+2\alpha}+2^{2k+2\alpha+1})\\
&\quad=\ n2^{n-3k-1}\sum_{j=0}^{k-1}2^{2k+j}\\
&\quad=\ \frac{1}{2}N\lg N+O(N).
\end{aligned}$$

The total B_W cost of steps AB(*i, ii; c*) and BA(*ii, iii; c*) is bounded above by:

$$\begin{aligned}
\sum_{\alpha=0}^{k/2-1}\left[\sum_{i=1}^{p}(t+n)n+\sum_{i=1}^{p-1}(t+2n)n\right] &< np\sum_{\alpha=0}^{k/2-1}((2n\alpha+n)+(2n\alpha+2n))\\
&=\ n2^{n-3k-1}\sum_{j=0}^{k-1}(j+1)n\\
&=\ O\left(\frac{N\log^2\log N}{\log N}\right).
\end{aligned}$$

The total B_W cost of steps AB(i, ii; d) and BA(ii, iii; d) is bounded above by:

$$\begin{aligned}
&\sum_{a=0}^{k/2-1}\left[\sum_{i=1}^{p}(2^{2k+2a}-t-n)3k+\sum_{i=1}^{p-1}(2^{2k+2a+1}-t-2n)3k\right] \\
&\quad < 3kp\sum_{a=0}^{k/2-1}(2^{2k+2a}+2^{2k+2a+1}) \\
&\quad = 3k2^{n-3k-1}\sum_{j=0}^{k-1}2^{2k+j} \\
&\quad = O(N\log\log N).
\end{aligned}$$

The total B_W cost of step BA(i) is bounded above by:

$$\sum_{a=0}^{k/2-1} n2^{2k+2a+1} = n2^{2k+1}\frac{2^k-1}{3} = O(\log^4 N).$$

Summing all of these upper bounds shows that the total B_W cost of the construction is at most $\frac{1}{2}N\lg N + O(N\log\log N)$.

The total C_W cost of steps AB(i, ii; a, b), BA(ii, iii; a), BA(ii, iii; b), BA(iv; a), and BA(iv; b) is bounded above by:

$$\begin{aligned}
&\sum_{a=0}^{k/2-1}\left[\sum_{i=1}^{p}((t)+(2^{2k+2a}-t))(bit(i-1)+3k)\right. \\
&\qquad +\sum_{i=1}^{p-1}(t+n)(bit(i-1)+3k)+\sum_{i=1}^{p-1}(2^{2k+2a+1}-t-n)(bit(i)+3k) \\
&\qquad \left.+(t+n)(bit(p-1)+3k)+(2^{2k+2a+1}-t-n)(3k)\right] \\
&\quad = \sum_{a=0}^{k/2-1}\left[\sum_{i=1}^{p}2^{2k+2a}(bit(i-1)+3k)+\sum_{i=1}^{p}2^{2k+2a+1}(bit(i-1)+3k)\right] \\
&\quad = \sum_{i=1}^{p}(bit(i-1)+3k)\sum_{a=0}^{k/2-1}(2^{2k+2a}+2^{2k+2a+1}) \\
&\quad = \left(\frac{n-3k-1}{2}+3k\right)2^{n-3k-1}\sum_{j=0}^{k-1}2^{2k+j} \\
&\quad = \frac{1}{4}N\lg N + O(N\log\log N).
\end{aligned}$$

The total C_W cost of steps AB(*i*, *ii*; *c*) and BA(*ii*, *iii*; *c*) is bounded above by the B_W cost of these steps, which is $O(N \log^2 \log N / \log N)$. Likewise, the total C_W cost of step BA(*i*) is bounded above by the B_W cost of this step, which is $O(\log^4 N)$. The total C_W cost of steps AB(*i*, *ii*; *d*) and BA(*ii*, *iii*; *d*) is identical with that given for B_W on these steps, namely $O(N \log\log N)$. Summing all of these bounds gives an upper bound of $\frac{1}{4}N \lg N + O(N \log\log N)$ on the C_W cost of the construction.

It has been shown that Figures 4.4 and 4.5 of this section construct a pulldown sequence that results in a heap whose B_W and C_W costs are bounded above by $\frac{1}{2}N \lg N + O(N \log\log N)$ and $\frac{1}{4}N \lg N + O(N \log\log N)$. This completes the proof of Theorem 4.4. By the results of the previous section, it is impossible for any heap to have asymptotically lower costs than these. It follows that the construction given in this section is indeed a best case for Williams's Heapsort.

4.3 The Worst Case of Williams's Heapsort

In the previous section, a pulldown sequence was constructed that demonstrated the asymptotic behavior of Williams's Heapsort in the best case. The low cost of this pulldown sequence is a result of its alternation between pulldowns of large keys from the top of the heap and small keys from the bottom of the heap. It is known, [18] exercise (5.2.3.23), that a heap can be created with the largest possible B_W cost by pulling down the smallest key $N - 1$ times. To obtain the worst possible B_W and C_W costs simultaneously, it is natural to consider pulling all keys from the bottom of the heap, and to pull as many as possible from the bottom right side of the heap. This suggests proceeding as in the last section, except that C should now be taken to be what had been the rightmost pair of A_i and B_i, and D need not exist. This approach does indeed yield a construction for the worst case of Williams's Heapsort, and is presented in detail in this section.

Figure 4.6 illustrates schematically the construction of a worst case input sequence to Williams's Heapsort. The basic idea is the same here as in the previous section. The

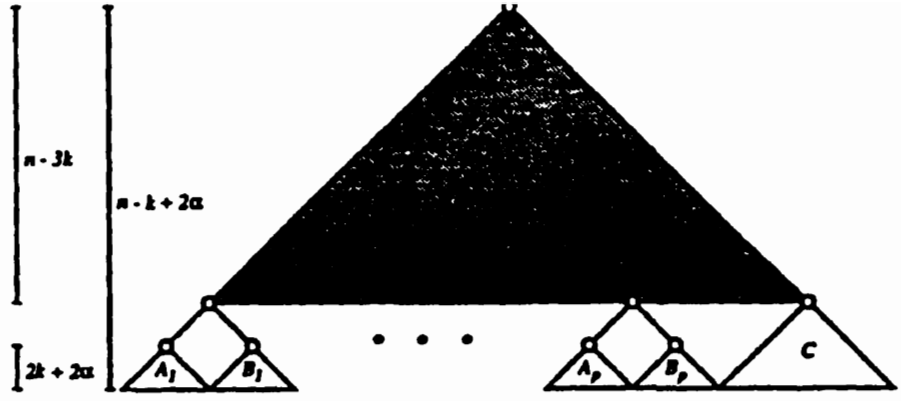

Figure 4.6: Schematic of the Williams's Heapsort worst case construction.

keys in heap B_i are pulled down to add a level to A_i, leaving B_i full of large keys. Keys from C are then pulled down to add a new level to B_i. As before, it is necessary to add levels to the heap in pairs, first pulling keys from the B_i into the A_i, and then from the A_{i+1} into the B_i. As in the previous section, it is assumed that the size of the final heap is desired to be $N = 2^n - 1$; the construction starts with a seed heap of size $2^{n-k} - 1$ where $k = 2\lceil \frac{\lg n}{2} \rceil$, and is expanded to a heap on N keys by $k/2$ passes, each of which adds a pair of levels to the heap. Again set $p = 2^{n-3k-1}$ so that the A_i are numbered from 1 to $p-1$ and have roots at $A[2p+2i-2]$ while the roots of the B_i are $A[2p+2i-1]$. Define C to be the subheap (all levels) rooted at $A[2p-1]$. As before, set $t = 2n\alpha + 1$ where α is the current pass of the construction, and let $L(H, x)$ and $S(H, x)$ be the x largest and smallest keys of H.

THEOREM 4.5: Figures 4.7 and 4.8 specify a pulldown sequence P with $B_W(H(P)) = N \lg N + O(N)$ and $C_W(H(P)) = \frac{3}{4} N \lg N + O(N \log \log N)$. The costs given for each step bound the average cost of the pulldowns in that step.

Proof: The next subsection will prove that Figures 4.7 and 4.8 describe a legitimate pulldown sequence. The following subsection justifies and sums the claimed cost bounds. □

I. Build a seed heap on $2^{n-k}-1$ keys that satisfies the following inequalities:

1. $L(A_i, 2^{2k}-1) > S(B_i, 2^{2k}-1)$ for all i.
2. $L(C, 2^{2k}-n-1) > S(A_2, 2^{2k}-1)$.
3. $L(C, 2^{2k}-n-1) > S(A_3, 2^{2k}-1)$.

II. Complete the heap by executing:

For $a = 0$ to $k/2-1$ execute the code in Figure 4.8.

Figure 4.7: Construction for Williams's Heapsort worst case.

Action Performed	B_W Lower Bound	C_W Lower Bound
Phase AB:		
(*i*) For $i = 1$ to $p-1$ do:		
a) Pull down $S(A_i, t)$	$n-k+2a$	$bit(i-1)$
b) Pull down $S(B_i, 2^{2k+2a}-t)$	$n-k+2a$	$bit(i-1)$
c) Pull down $S(B_i, t+n)$	$n-k+2a$	0
d) Pull down $L(C, 2^{2k+2a}-t-n)$	$n-k+2a-3$	$n-3k-1$
(*ii*) Pull down $S(C,1)$, $2^{2k+2a+1}$ times	0	0
Phase BA:		
(*i*) Pull down $S(A_1,1)$, $2^{2k+2a+1}$ times	0	0
(*ii*) For $i = 1$ to $p-2$ do:		
a) Pull down $S(B_i, t+n)$	$n-k+2a+1$	$bit(i-1)$
b) Pull down $S(A_{i+1}, 2^{2k+2a+1}-t-n)$	$n-k+2a+1$	$bit(i)$
c) Pull down $S(A_{i+1}, t+2n)$	$n-k+2a+1$	0
d) Pull down $L(C, 2^{2k+2a+1}-t-2n)$	$n-k+2a-2$	$n-3k-1$
(*iii*)		
a) Pull down $S(B_{p-1}, t+n)$	$n-k+2a+1$	0
b) Pull down $S(A_1, 2^{2k+2a+1}-t-n)$	$n-k+2a+1$	0
(*iv*) Pull down $S(C,1)$, $2^{2k+2a+2}$ times	0	0

Figure 4.8: The Williams's Heapsort worst case main loop, with cost bounds.

4.3.1 Correctness of the Construction

The general outline of the proof of correctness for this construction is the same as for the proof given of the best case construction for Williams's Heapsort in the previous section. The argument is essentially inductive, showing first that a seed heap satisfying the inequalities of Figure 4.7 can be created on $2^{n-k}-1$ keys; it is then shown that this seed heap can be extended to a heap on N keys by $k/2$ passes of Figure 4.8 where each pass preserves a set of invariants.

Inequalities (4.6), (4.7), and (4.8) are the pass invariants for the construction given in Figures 4.7 and 4.8. These are basically the same inequalities as the pass invariants for the construction of the previous section, the only differences being that the keys of C must be larger than the keys of both A_2 and A_3, and that the range of the variable i is smaller by one.

- $L(A_i, 2^{2(k+\alpha)}-t) > S(B_i, 2^{2(k+\alpha)}-t)$ for all i. (4.6)
- $L(C, 2^{2(k+\alpha)}-t-n) > L(A_2, 2^{2(k+\alpha)}-1)$. (4.7)
- $L(C, 2^{2(k+\alpha)}-t-n) > L(A_3, 2^{2(k+\alpha)}-1)$. (4.8)

It will first be shown that a seed heap can be created to satisfy these invariants with α set to equal zero. It will then be shown that it is possible for an execution of the code given in Figure 4.8 to add two levels to a heap that satisfies these invariants, and that the resulting heap satisfies the pass invariants as well.

One way to create the necessary seed heap on $2^{n-k}-1$ keys is to divide these keys into four intervals: $[1,(p-1)(2^{2k}-1)]$, $[(p-1)(2^{2k}-1)+1, 2(p-1)(2^{2k}-1)]$, $[2(p-1)(2^{2k}-1)+1, 2p(2^{2k}-1)+1]$, and $[2p(2^{2k}-1)+2, 2^{n-k}-1]$. The integers in the first interval should be placed in the B_i, $1 \le i \le p-1$, and arranged in heap order within each B_i. The $(p-1)(2^{2k}-1)$ keys in the second interval should be placed in the A_i and again arranged in heap order. The $2^{2k+1}-1$ keys in the third interval should be placed in C in heap order. Finally, the $2p-2$ keys in the last interval should be

arranged in heap order in the the open positions of the top $n - 3k$ levels of the heap (note that one position from these levels, namely the root of C has already been filled). The resulting configuration of keys can be seen to satisfy the pass invariant (4.6) because the keys in the A_i come from the second interval and are thus larger than the keys in the B_i which come from the first interval. Likewise, the pass invariants (4.7) and (4.8) can be seen to hold since the keys in C come from the third interval and are thus greater than any key in the A_i or B_i. Finally, it is clear that this arrangement of keys is a heap, since the keys within each interval are arranged in heap order, and the roots of the A_i, B_i, and C have as their parents keys from the fourth interval.

It remains to show that a heap satisfying the pass invariants permits the execution of the code given in Figure 4.8, which adds two new levels to the heap and restores the pass invariants. Since a similar argument was given in detail in the previous section, the following discussion gives details only where they differ from those that have already been presented.

The execution of Phase AB is basically the same as Phase AB of the construction from the previous section. Step AB(i) adds a level to all of the A_i and B_i: Iteration i of step AB(i; a) starts level $2k+2a+1$ of A_i and guarantees that all keys of A_i's level $2k+2a$ are greater than any key of $S(B_i, 2^{2(k+a)} - t)$. This permits step AB(i; b) to finish the new level of A_i. Likewise, step AB(i; c) guarantees that all keys on level $2k + 2a$ of B_i are greater than any key of $L(C, 2^{2(k+a)} - t - n)$ permitting step AB(i; d) to complete the new level of B_i. Once step AB(i) has added a new level to all but C, step AB(ii) finishes the new level by pulling down the smallest key in C, $2^{2(k+a)+1}$ times.

At the end of Phase AB, the following inequalities hold:

- $L(B_i, 2^{2(k+a)+1} - t - n) > S(A_{i+1}, 2^{2(k+a)+1} - t - n)$ for $i \leq p - 2$. (4.9)
- $L(B_{p-1}, 2^{2(k+a)+1} - t - n) > S(A_1, 2^{2(k+a)+1} - t - n)$. (4.10)
- $L(C, 2^{2(k+a)+1} - t - 2n) > L(B_2, 2^{2(k+a)+1} - 1)$. (4.11)
- $L(C, 2^{2(k+a)+1} - t - 2n) > L(B_3, 2^{2(k+a)+1} - 1)$. (4.12)

Inequality (4.9) follows from the origins of the keys in A_i and B_i. Fix $i \leq p-2$. Then at the end of Phase AB, B_i contains at least $2^{2(k+\alpha)}$ keys that entered the heap in the i'th iteration of steps AB(i; b, c). These keys are larger than the $2^{2(k+\alpha)} - t - n$ keys that were pulled into B_i from C in step AB(i; d). The keys from C are in turn larger than any of the keys that were originally in A_{i+1} — for $i = 1, 2$ this follows from inequalities (4.7) and (4.8) of the pass invariant; for larger i, it follows from the observation that after the 2nd iteration of step AB(i; d), all keys in $L(C, 2^{2(k+\alpha)+1} - t - n)$ are new to the heap since the beginning of Phase AB. Finally, A_{i+1} contains all of its keys from the beginning of Phase AB as well as $2^{2(k+\alpha)} - t$ keys from B_{i+1} that are smaller than many of the keys that were already in A_{i+1}. Inequality (4.9) follows immediately.

Inequality (4.10) follows the same logic; the keys in $L(B_{p-1}, 2^{2(k+\alpha)+1} - t - n)$ have all entered the heap since the beginning of Phase AB, while every one of the keys in $S(A_1, 2^{2(k+\alpha)+1} - t - n)$ was in either A_1 or B_1 at the beginning of Phase AB.

Inequalities (4.11) and (4.12) follow directly from the observation that in step AB(ii), C picks up at least $2^{2(k+\alpha)+1} - n$ new keys that are larger than any key in any A_i or B_i.

These four inequalities permit Phase BA to add an additional level to the heap. Phase BA begins with step BA(i) adding a level to A_1. Step BA(ii) adds the new level to the remaining A_i and B_i with the exception of B_{p-1}. The operation of step BA(ii) is the same as that of step AB(i), with corresponding substeps affecting the subheaps on which they operate in corresponding ways. Step BA(iii) adds the new level to B_{p-1}; BA(iii; a) first guarantees that every key of level $2k + 2\alpha + 1$ in B_i is greater than any key of $S(A_1, 2^{2(k+\alpha)+1} - t - n)$; BA($iii$; b) then uses this fact to pull down the keys of $S(A_1, 2^{2(k+\alpha)+1} - t - n)$ to complete the new level of B_{p-1}. Finally, step BA(iv) completes the new level of the heap by extending it across C.

At the end of Phase BA, the pass invariants are restored once α is incremented. Pass invariant (4.6) follows for $i > 1$ by the same reasoning as was used to establish the first inequality above. To see that (4.6) holds for $i = 1$, note that $L(A_1, 2^{2(k+(\alpha+1))} - t - 2n)$ contains keys that are new to the heap since the beginning of Phase BA, which it

acquired during steps BA(i) and BA(iii; b), while $S(B_1, 2^{2(k+(\alpha+1))} - t - 2n)$ contains keys that were already in the heap, in B_1 and A_2, at the beginning of Phase BA.

Pass invariants (4.7) and (4.8) follow from the fact that in step BA(iv), C acquires at least $2^{2(k+(\alpha+1))} - n$ new keys that are larger than any key in any A_i or B_i.

It has been shown that it is possible to carry out the construction detailed in Figure 4.7 to create a heap on N keys: a seed heap on $2^{n-k} - 1$ keys can be created that satisfies the pass invariants for $\alpha = 0$; this seed heap can then be expanded to a heap on $2^n - 1$ keys by $k/2$ passes of Figure 4.8. It remains only to verify and sum the cost bounds given for the steps of the construction.

4.3.2 Costs of the Construction

The construction given in Figures 4.7 and 4.8 is intended to demonstrate the worst case asymptotic behavior of Williams's Heapsort. This permits a slight simplification of the analysis of the costs of the construction. In particular, the only steps of the construction whose costs need to be bounded and summed are those that contribute to the leading term of the total cost. For this reason, a number of zero entries appear in the cost columns of Figure 4.8 in positions where somewhat tighter bounds would have been possible. In the remainder of this section, the non-zero bounds are justified and totaled.

Consider first the B_W costs of steps AB(i; d) and BA(ii; d). Both of these steps pull down almost half of the keys of C. Since the keys pulled down are the large keys of C, they form a subtree within C at the beginning of the step. Because each key is pulled down from a position at most as high as the one it occupied at the beginning of the step, the average B_W cost of a pulldown in these steps is at least the level of the root of C plus the average internal path length of the subtree of keys being pulled down. This yields the claimed bounds. The B_W costs of the other steps are more easily bounded. In all other steps, the keys pulled down are the smallest keys in their subheaps. Since these keys are pulled down in increasing order, each key is at the old bottom level of the

heap when it is pulled down; this is level $n-k+2\alpha$ in Phase AB and level $n-k+2\alpha+1$ in Phase BA.

The claimed bounds on C_W cost follow from logic similar to that used to bound the C_W costs of the previous construction. Given any position x in A_i or B_i, the most significant bits of the binary representation of x are a "1" followed by the $n-3k-1$ bit binary representation of $i-1$. This proves the bounds on the costs of steps AB(i; a, b) and BA(ii; a, b). Similarly, the binary representation of a position x in C begins with the binary representation of $2^{n-3k}-1 = 2p-1$ since A[$2p-1$] is the root of C. This justifies the remaining non-zero bounds.

The total B_W cost for steps AB(i; a, b), BA(ii; a, b), and BA(iii; a, b) is bounded from below by:

$$\begin{aligned}
&\sum_{\alpha=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-1}((t)+(2^{2k+2\alpha}-t))(n-k+2\alpha) \\
&\qquad\qquad +\sum_{i=1}^{p-2}((t+n)+(2^{2k+2\alpha+1}-t-n))(n-k+2\alpha+1) \\
&\qquad\qquad +((t+n)+(2^{2k+2\alpha+1}-t-n))(n-k+2\alpha+1)\Bigg] \\
&= (p-1)\sum_{\alpha=0}^{k/2-1}(2^{2k+2\alpha}(n-k+2\alpha)+2^{2k+2\alpha+1}(n-k+2\alpha+1)) \\
&= (2^{n-3k-1}-1)2^{2k}\left[\sum_{j=0}^{k-1}2^j j+\sum_{j=0}^{k-1}2^j(n-k)\right] \\
&= (2^{n-3k-1}-1)2^{2k}[(k-2)2^k+2+(n-k)(2^k-1)] \\
&= (2^{n-3k-1}-1)(n2^k-2^{k+1}-n+k+2)2^{2k} \\
&= \frac{1}{2}N\lg N+O(N).
\end{aligned}$$

The total B_W cost for steps AB(i; c, d) and BA(ii; c, d) is bounded below by:

$$\sum_{\alpha=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-1}((t+n)(n-k+2\alpha)+(2^{2k+2\alpha}-t-n)(n-k+2\alpha-3))$$

$$
\begin{aligned}
&\quad + \sum_{i=1}^{p-2}((t+2n)(n-k+2\alpha+1)+(2^{2k+2\alpha+1}-t-2n)(n-k+2\alpha-2))\Bigg] \\
&> (p-2)\sum_{\alpha=0}^{k/2-1}(2^{2k+2\alpha}(n-k+2\alpha)+2^{2k+2\alpha+1}(n-k+2\alpha+1)) \\
&\quad - 3(p-1)\sum_{\alpha=0}^{k/2-1}(2^{2k+2\alpha}+2^{2k+2\alpha+1}) \\
&= \frac{1}{2}N\lg N + O(N).
\end{aligned}
$$

The last step in this derivation can be seen to follow from the previous derivation. Combining these two bounds yields a lower bound of $N \lg N + O(N)$ on the total B_W cost of this construction.

The total C_W cost of steps AB(i; a, b), BA(ii; a), and BA(ii; b) is bounded below by:

$$
\begin{aligned}
&\sum_{\alpha=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-1}((t)+(2^{2k+2\alpha}-t))bit(i-1)+\sum_{i=1}^{p-2}(t+n)bit(i-1) \\
&\qquad + \sum_{i=1}^{p-2}(2^{2k+2\alpha+1}-t-n)bit(i)\Bigg] \\
&\quad > \sum_{\alpha=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-2}2^{2k+2\alpha}bit(i-1)+\sum_{i=1}^{p-2}2^{2k+2\alpha+1}bit(i-1)\Bigg] \\
&\quad = \sum_{i=1}^{p-2}bit(i-1)\sum_{\alpha=0}^{k/2-1}(2^{2k+2\alpha}+2^{2k+2\alpha+1}) \\
&\quad = \left(\frac{n-3k-1}{2}2^{n-3k-1}-2n+6k+3\right)\sum_{j=0}^{k-1}2^{2k+j} \\
&\quad = \frac{1}{4}N\lg N + O(N\log\log N).
\end{aligned}
$$

The total C_W cost of steps AB(i; d) and BA(ii; d) is bounded below by:

$$
\sum_{\alpha=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-1}(2^{2k+2\alpha}-t-n)(n-3k-1)+\sum_{i=1}^{p-2}(2^{2k+2\alpha+1}-t-2n)(n-3k-1)\Bigg]
$$

$$
\begin{aligned}
&> \sum_{\alpha=1}^{k/2-1} \left[\sum_{i=1}^{p-2} (2^{2k+2\alpha} - t - n)(n-6k) + \sum_{i=1}^{p-2} (2^{2k+2\alpha+1} - t - 2n)(n-6k) \right] \\
&= (n-6k)(p-2) \sum_{\alpha=1}^{k/2-1} ((2^{2k+2\alpha} - 2n\alpha - n) + (2^{2k+2\alpha+1} - 2n\alpha - 2n)) \\
&= (n-6k)(2^{n-3k-1} - 2) \sum_{j=2}^{k-1} (2^{2k+j} - (j+1)n) \\
&= \frac{1}{2} N \lg N + O(N \log\log N).
\end{aligned}
$$

These bounds combine to give a lower bound of $\frac{3}{4} N \lg N + O(N \lg\lg N)$ on the C_W cost of this construction.

This section has shown that Figures 4.7 and 4.8 specify a pulldown sequence that constructs a heap with B_W cost $N \lg N + O(N)$ and C_W cost $\frac{3}{4} N \lg N + O(N \log\log N)$; this completes proof of Theorem 4.5. The results of the first section of this chapter state that asymptotically it is impossible for any heap to have higher costs than these.

4.4 The Best Case of Floyd's Heapsort

Pulldowns from the bottom of the heap are expensive for Williams's Heapsort to reverse since they require that a key be moved all the way to the bottom of the heap with two key comparisons at each step. For Floyd's Heapsort, however, such pulldowns are the least costly to reverse since a path must be traced to the bottom of the heap every time a key is moved into place; only when a key must be moved back up that path is additional expense incurred. It is known that the best case of Floyd's Heapsort requires only $N \lg N + O(N)$ comparisons, [18] exercise (5.2.3.23) notes that such a heap results from pulling down the smallest key $N - 1$ times. A heap that simultaneously exhibits optimal B_F and C_F costs can constructed in a manner similar to that of the previous section, by a pulldown sequence whose structure is the mirror image of the worst case for Williams's Heapsort, as is shown in Figure 4.9.

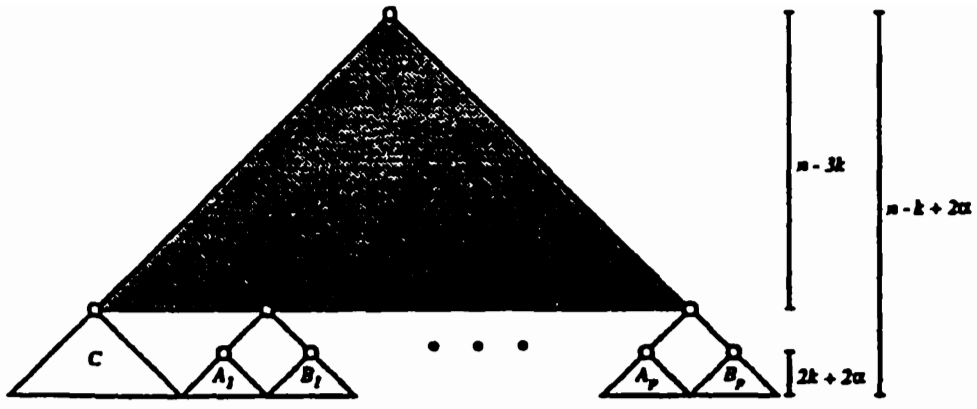

Figure 4.9: Schematic of the Floyd's Heapsort best case construction.

The details of this construction are very similar to those of the preceeding one. As before, the heap is built from a seed heap on $2^{n-k}-1$ keys where $k = 2\lceil \frac{\lg n}{2} \rceil$, to a heap on $N = 2^n - 1$ keys by $k/2$ passes, each of which adds two levels to the heap. As before, if $p = 2^{n-3k-1}$ then there are $p-1$ subheaps A_i and B_i whose roots are at positions $A[2p+2i]$ and $A[2p+2i+1]$ respectively. C is defined to be the subheap rooted at $A[p]$. If $t = 2n\alpha + 1$, and $L(H,x)$ and $S(H,x)$ are defined to be as before, then the following Theorem holds:

THEOREM 4.6: Figures 4.10 and 4.11 specify a pulldown sequence P with $B_F(H(P)) = O(N)$ and $C_F(H(P)) = \frac{1}{4}N \lg N + O(N \log\log N)$. The costs given with each step are upper bounds on the average cost of the pulldowns in that step.

Proof: The next subsection proves that Figures 4.10 and 4.11 describe a legitimate pulldown sequence. The following subsection justifies and sums the claimed cost bounds. □

4.4.1 Correctness of the Construction

The arguments needed to prove this construction correct are very similar to those presented with the previous construction, so most of their details will be omitted here. The proof is again inductive; the code in Figure 4.10 can be executed if it is possible to

I. Build a seed heap on $2^{n-k}-1$ keys that satisfies the ineqnality:
$L(A_i, 2^{2k}-1) > S(B_i, 2^{2k}-1)$ for all i.

II. Complete the heap by executing:
For $a = 0$ to $k/2-1$ execute the code in Figure 4.11.

Figure 4.10: Construction for Floyd's Heapsort best case.

Action Performed	B_F Upper Bound	C_F Upper Bound
Phase AB:		
(*i*) Pull down $S(C,1)$, $2^{2k+2a+1}$ times	1	$3k$
(*ii*) For $i = 1$ to $p-1$ do:		
a) Pull down $S(A_i, t)$	1	$bit(i)+3k$
b) Pull down $S(B_i, 2^{2k+2a}-t)$	1	$bit(i)+3k$
c) Pull down $S(B_i, t+n)$	1	$bit(i)+3k$
d) Pull down $L(C, 2^{2k+2a}-t-n)$	4	$3k$
Phase BA:		
(*i*) Pull down $S(C,1)$, $2^{2k+2a+2}$ times	1	$3k$
(*ii*) Pull down $S(A_1,1)$, $2^{2k+2a+1}$ times	1	$3k+1$
(*iii*) For $i = 1$ to $p-2$ do:		
a) Pull down $S(B_i, t+n)$	1	$bit(i)+3k$
b) Pull down $S(A_{i+1}, 2^{2k+2a+1}-t-n)$	1	$bit(i+1)+3k$
c) Pull down $S(A_{i+1}, t+2n)$	1	$bit(i+1)+3k$
d) Pull down $L(C, 2^{2k+2a+1}-t-2n)$	4	$3k$
(*iv*)		
a) Pull down $S(B_{p-1}, t+n)$	1	$bit(p-1)+3k$
b) Pull down $S(A_1, 2^{2k+2a+1}-t-n)$	1	$3k+1$

Figure 4.11: The Floyd's Heapsort best case main loop, with cost bounds.

create the required seed heap and if the pass invariant given below both permits and is restored by an execution of the code given in Figure 4.11.

There is only one pass invariant for this construction, namely:

- $L(A_i, 2^{2(k+\alpha)} - t) > S(B_i, 2^{2(k+\alpha)} - t)$ for all i. (4.13)

A seed heap satisfying the pass invariant for $\alpha = 0$ can be created in exactly the same manner as in the previous section; the $2^{n-k} - 1$ keys are divided into the same intervals, which are assigned to the B_i, A_i, C, and top levels, as before. The only difference from the previous section is that this time C is on the left side of the heap. It remains to show that the pass invariant permits and is restored by a pass of Figure 4.11.

Assume that the pass invariant holds as Phase AB of Figure 4.11 begins. Then Phase AB operates in the same manner as Phase AB of the construction from the previous section, except that the new level is added to C before it is added to the A_i and B_i. Note that this implies that by the end of step AB(i), and for the remainder of Phase AB, all keys in $L(C, 2^{2(k+\alpha)} - t - n)$ are new to the heap since the beginning of Phase AB. At the end of Phase AB, the following inequalities hold:

- $L(B_i, 2^{2(k+\alpha)+1} - t - n) > S(A_{i+1}, 2^{2(k+\alpha)+1} - t - n)$ for $i \leq p - 2$. (4.14)
- $L(B_{p-1}, 2^{2(k+\alpha)+1} - t - n) > S(A_1, 2^{2(k+\alpha)+1} - t - n)$. (4.15)

As before, these inequalities are proved by examining the sources of the keys in the A_i and B_i. Fix i. At least $2^{2(k+\alpha)}$ keys of B_i are new to the heap in steps AB(*ii; b, c*), and another $2^{2(k+\alpha)} - t - n$ keys of B_i came from C in step AB(*ii; d*) and thus are also new to the heap since the beginning of Phase AB, as noted previously. This means that all keys in $L(B_i, 2^{2(k+\alpha)+1} - t - n)$ are new to the heap since the beginning of Phase AB. A_i, on the other hand, contains all $2^{2(k+\alpha)} - 1$ keys that it contained at the beginning of Phase AB, plus $2^{2(k+\alpha)} - t$ keys pulled down from B_i in step AB(*ii; b*). It follows that all keys in $S(A_i, 2^{2(k+\alpha)+1} - t - n)$ were in the heap prior to the beginning of Phase AB. Since i was arbitrary, inequalities (4.14) and (4.15) follow immediately.

These inequalities allow Phase BA to add another level to the heap. The same reasoning used to establish the inequalities at the end of Phase AB shows that at the end of Phase BA, the keys in $L(A_i, 2^{2(k+(\alpha+1)} - t - 2n)$ are all new to the heap since the beginning of Phase BA, while those in $S(B_i, 2^{2(k+(\alpha+1))} - t - 2n)$ were already in the heap at the beginning of Phase BA. This implies that when α is incremented, the pass invariant (4.13) is restored.

Because it is possible to create the seed heap required by Figure 4.10, and because the pass invariant holds for the seed heap and after every iteration of the code in Figure 4.11, it follows that it is possible to create the heap specified by Figure 4.10. It remains to justify and total the specified bounds on the cost of each step.

4.4.2 Costs of the Construction

The best case construction for Floyd's Heapsort differs little from the worst case construction for Williams's Heapsort, so one would expect that the cost bounds for the two heaps would be similar in flavor. This would be the case, except that the measures of cost for Floyd's Heapsort are different from those for Williams's Heapsort. For example, the B_F cost of a pulldown from a position x is the distance from the bottommost level of the heap to the level of position x. Because keys are pulled down in increasing order within a given group of pulldowns, all keys pulled down in steps AB(*ii; a, b, c*) are pulled from level $n-k+2\alpha$ while the bottommost level of the heap is $n-k+2\alpha+1$; these pulldowns thus have unit B_F costs. The same reasoning applies to steps BA(*iii; a, b, c*) and BA(*iv; a, b*). Similar logic shows that in steps AB(*i*) and BA(*i, ii*), the first pulldown has a B_F cost of 1 with subsequent pulldowns being free. The claimed bound on the B_F cost of step AB(*ii; d*) follows from the reasoning of the previous section by which the keys being pulled down initially form a subtree of C, so the average key is pulled down from a level numbered at least $n - k + 2\alpha - 3$ while the bottommost level is $n - k + 2\alpha + 1$. BA(*iii; d*) is bounded similarly.

The C_F cost of a pulldown from a position x in A_i or B_i equals the number of times

the right child is selected along the path that goes from the root of the heap, through position x, and down to some key at the bottom of the heap. Observe that such a path must go through the parent of the roots of A_i and B_i, and from there it visits at most $3k$ additional positions before hitting the bottom of the heap. Observe also that the right child is selected exactly $bit(i)$ times along the path from the root of the heap to the parent of the roots of A_i and B_i. These two observations imply the claimed C_F cost bounds for pulldowns from the A_i and B_i. The cost of pulling down keys in C is similarly bounded since a path from the root of the heap to a position in C will pass from each node to its left child until it is below the root of C.

The bounds given in Figure 4.11 must be totaled now that they have been justified. Steps AB(*ii; a, b*), BA(*iii; a, b*), BA(*iv; a, b*), AB(*ii; c, d*), and BA(*iii; c, d*) have a total B_F cost that is bounded above by:

$$\begin{aligned}
&\sum_{a=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-1}((t)+(2^{2k+2a}-t)) \\
&\qquad+\sum_{i=1}^{p-2}((t+n)+(2^{2k+2a+1}-t-n))+((t+n)+(2^{2k+2a+1}-t-n)) \\
&\qquad+\sum_{i=1}^{p-1}((t+n)+4(2^{2k+2a}-t-n))+\sum_{i=1}^{p-2}((t+2n)+4(2^{2k+2a+1}-t-2n))\Bigg] \\
&< 5(p-1)\sum_{a=0}^{k/2-1}(2^{2k+2a}+2^{2k+2a+1}) \\
&= 5(2^{n-3k-1}-1)\sum_{j=0}^{k-1}2^{2k+j} \\
&= O(N).
\end{aligned}$$

The total B_F cost of steps AB(i), BA(i), and BA(ii) is bounded above by:

$$\sum_{a=0}^{k/2-1}(2^{2k+2a+1}+2^{2k+2a+2}+2^{2k+2a+1})=\sum_{a=0}^{k/2-1}2^{2k+2a+3}=O(\log^3 N).$$

Together, these expressions show that the B_F cost of the construction given in Figure 4.10 is $O(N)$.

The total C_F cost of steps AB(*ii; a, b*), BA(*iii; a*), BA(*iii; b*), BA(*iv; a*), and BA(*iv; b*) is bounded above by:

$$\begin{aligned}
&\sum_{a=0}^{k/2-1}\Bigg[\sum_{i=1}^{p-1}((t)+(2^{2k+2a}-t))(bit(i)+3k) \\
&\qquad +\sum_{i=1}^{p-2}(t+n)(bit(i)+3k)+\sum_{i=1}^{p-2}(2^{2k+2a+1}-t-n)(bit(i+1)+3k) \\
&\qquad +(t+n)(bit(p-1)+3k)+(2^{2k+2a+1}-t-n)(3k+1)\Bigg] \\
&< \sum_{a=0}^{k/2-1}\Bigg[\sum_{i=1}^{p}2^{2k+2a}(bit(i-1)+3k)+\sum_{i=1}^{p}2^{2k+2a+1}(bit(i-1)+3k)\Bigg] \\
&= \frac{1}{4}N\lg N+O(N\log\log N).
\end{aligned}$$

The last step of this derivation has already been worked out in the analysis of the best case for C_W of Williams's Heapsort. The total C_F cost of steps AB(*ii; c*) and BA(*iii; c*) is $O(N\log^2\log N/\log N)$ since it is bounded above by the C_W cost of steps AB(*i, ii; c*) and BA(*ii, iii; c*) of the best case construction for Williams's Heapsort. Likewise, the bounds on steps AB(*ii; d*) and BA(*iii; d*) are almost identical with those given for steps AB(*i, ii; d*) and BA(*ii, iii; d*) of the best case of C_W for Williams's Heapsort; the only difference being that the latter bounds run over a larger set of pulldowns. It follows that the work performed in these steps is $O(N\log\log N)$. The total C_F cost of steps AB(*i*), BA(*i*), and BA(*ii*) is bounded above by:

$$\sum_{a=0}^{k/2-1}(3k2^{2k+2a+1}+3k2^{2k+2a+2}+(3k+1)2^{2k+2a+1})=O(\log^3 N\log\log N).$$

Combining these expressions gives $\frac{1}{4}N\lg N+O(N\log\log N)$ as an upper bound on the C_F cost the heap constructed by the pulldown sequence specified by Figures 4.10 and 4.11. This completes proof of Theorem 4.6.

4.5 The Worst Case of Floyd's Heapsort

When the construction for the best case of Williams's Heapsort was presented, it was mentioned that it could also be taken as an input on which Floyd's Heapsort exhibits its worst case asymptotic behavior. Figures 4.12 and 4.13 thus give exactly the same construction as Figures 4.4 and 4.5, but with different cost bounds for each step.

THEOREM 4.7: Figures 4.12 and 4.13 specify a pulldown sequence P with $B_F(H(P)) = \frac{1}{2}N\lg N + O(N\log\log N)$ and $C_F(H(P)) = \frac{3}{4}N\lg N + O(N\log\log N)$. The costs given with each step are upper bounds on the average cost of the individual pulldowns in that step.

Proof: Since it has already been verified that these figures specify a legitimate pulldown sequence, it remains only to justify and total the new set of cost bounds. Bounds on the B_F costs of the various steps aren't really necessary to prove that this pulldown sequence constructs a heap with B_F cost equal to $\frac{1}{2}N\lg N + O(N\log\log N)$. This follows directly from Theorem 4.4 and part 5 of Lemma 2.1.

Bounds on the C_F costs of steps AB(*i*, *ii*; *a*, *b*) and BA(*ii*, *iii*; *a*, *b*) are also quite straightforward. These are derived by the techniques that have been used previously; any path from the root of the heap to a position x in A_i or B_i must pass through the parent of the roots of A_i and B_i and thus selects at least $bit(i-1)$ right children between the root of the heap and x.

In steps AB(*i*, *ii*; *d*) and BA(*ii*, *iii*; *d*), the cost of a pulldown is not a strict function of the position of the key being pulled down, but depends on the history of previous pulldowns. Assume that a key is pulled from a position x that is high in the heap. When Floyd's Heapsort reverses this pulldown, it will trace a path from the root of the heap, through x, to the bottom of the heap. Since x was pulled from high in the heap, most of the nodes along this path lie below x. If any bound is to be obtained on the number of right children selected by this path, it must not be based the binary representation of position x. Rather, some information must be gained about how often a right child contains a larger key than its left sibling.

I. Build a seed heap on $2^{n-k}-1$ keys that satisfies the following inequalities:
   1. $L(A_i, 2^{2k}-1) > S(B_i, 2^{2k}-1)$ for all i.
   2. $L(C, 2^{2k}-n-1) > S(A_2, 2^{2k}-1)$.

II. Complete the heap by executing:
   For $\alpha = 0$ to $k/2-1$ execute the code in Figure 4.13.

Figure 4.12: Construction for Floyd's Heapsort worst case.

Action Performed	B_F Lower Bound	C_F Lower Bound
Phase AB:		
(i) For $i = 1$ to $p/2$ do:		
a) Pull down $S(A_i, t)$	0	$bit(i-1)$
b) Pull down $S(B_i, 2^{2k+2\alpha}-t)$	0	$bit(i-1)$
c) Pull down $S(B_i, t+n)$	0	0
d) Pull down $L(C, 2^{2k+2\alpha}-t-n)$	0	$n-6k^*$
(ii) For $i = p/2+1$ to p do:		
a-d) As above, replacing C by D		
Phase BA:		
(i) Pull down $S(A_1, 1)$, $2^{2k+2\alpha+1}$ times	0	0
(ii) For $i = 1$ to $p/2-1$ do:		
a) Pull down $S(B_i, t+n)$	0	$bit(i-1)$
b) Pull down $S(A_{i+1}, 2^{2k+2\alpha+1}-t-n)$	0	$bit(i)$
c) Pull down $S(A_{i+1}, t+2n)$	0	0
d) Pull down $L(C, 2^{2k+2\alpha+1}-t-2n)$	0	$n-6k$
(iii) For $i = p/2$ to $p-1$ do:		
a-d) As above, replacing C by D		
(iv)		
a) Pull down $S(B_p, t+n)$	0	0
b) Pull down $S(A_1, 2^{2k+2\alpha+1}-t-n)$	0	0

* *This bound holds only for* $\alpha > 0$.

Figure 4.13: The Floyd's Heapsort worst case main loop, with cost bounds.

Consider any pulldown performed in one of steps AB(*ii; d*) or BA(*iii; d*). When Floyd's Heapsort reverses this pulldown, the path it traces to the bottom of the heap will never select a left child between levels $3k$ and $n-3k-1$. To see why, Let x be any position in the left half of the heap on a level numbered between $3k$ and $n-3k-1$ inclusive; for convenience, let $\ell_x = \lceil \lg(x+1) \rceil$ be the level of x. Now the last key to have been pulled down from a position below x was pulled down from the rightmost B_i below x during the execution of step AB(*i; c*) or BA(*ii; c*). That is, if $r = (x+1)2^{(n-3k-1)-\ell_x} - 1$, then the most recent pulldown of any key in a position below x was a pulldown from B_r, performed during iteration r of step AB(*i; c*) or BA(*ii; c*). Since the last key to be pulled down from a position below x was thus pulled down from below the right child of x, it follows that A[$2x+1$] > A[$2x$]. Since x was an arbitrary position, $3k \le \ell_x \le n-3k-1$, in the left half of the heap, it follows that at least $n-6k$ right children must be selected along any path traced by Floyd's Heapsort from a position in D to the bottom of the heap. This explains the bounds on the cost of steps AB(*ii; d*) and BA(*iii; d*). The same reasoning justifies the cost bounds on steps AB(*i; d*) and BA(*ii; d*) following at least one execution of Phase BA of Figure 4.13.

The C_F cost of steps AB(*i, ii; a, b*), BA(*ii, iii; a*), and BA(*ii, iii; b*) is bounded below by:

$$\begin{aligned}
&\sum_{\alpha=0}^{k/2-1}\left[\sum_{i=1}^{p}((t)+(2^{2k+2\alpha}-t))bit(i-1)+\sum_{i=1}^{p-1}(t+n)bit(i-1)\right.\\
&\qquad\qquad\left.+\sum_{i=1}^{p-1}(2^{2k+2\alpha+1}-t-n)bit(i)\right]\\
&> \sum_{\alpha=0}^{k/2-1}\left[\sum_{i=1}^{p-2}2^{2k+2\alpha}bit(i-1)+\sum_{i=1}^{p-2}2^{2k+2\alpha+1}bit(i-1)\right]\\
&= \frac{1}{4}N\lg N+O(N\log\log N).
\end{aligned}$$

The last step in this derivation follows from the bound proved on C_W in the worst case construction for Williams's Heapsort. The total C_F cost of steps AB(*i, ii; d*) and

BA(*ii*, *iii*; *d*) is bounded below by:

$$\begin{aligned}
&\sum_{\alpha=1}^{k/2-1}\left[\sum_{i=1}^{p}(2^{2k+2\alpha}-t-n)(n-6k)+\sum_{i=1}^{p-1}(2^{2k+2\alpha+1}-t-2n)(n-6k)\right] \\
&\quad > \sum_{\alpha=1}^{k/2-1}\left[\sum_{i=1}^{p-2}(2^{2k+2\alpha}-t-n)(n-6k)+\sum_{i=1}^{p-2}(2^{2k+2\alpha+1}-t-2n)(n-6k)\right] \\
&\quad = \frac{1}{2}N\lg N+O(N\log\log N).
\end{aligned}$$

The last step of this manipulation follows from the equations worked out in the worst case of C_{IV} for Williams's Heapsort. Adding these two bounds gives an asymptotic C_F cost of $\frac{3}{4}N\lg N+O(N\log\log N)$ for the construction. This concludes proof of Theorem 4.7. □

4.6 Conclusions and Comments

There are still a few unexamined issues that must be addressed before this chapter has achieved its purpose. First, there is the question of the sizes of the heaps constructed by the previous four sections. Figures 4.3 and 4.2 depicted a heap in which the subheap C would have overlapped the roots of the subheaps A_2 and B_2 had these been included in the illustration. It was then noted that in the actual constructions, the heaps in question should be large enough that such overlap never occurs. In all constructions, the variable k was defined to equal $2\lceil\frac{\lg n}{2}\rceil$ where the number of keys was $N=2^n-1$. In the construction that demonstrated the best and worst behaviors of Williams's and Floyd's Heapsorts respectively, the top $3k$ levels had to be reserved for C and D while the bottom $3k$ levels had to be reserved for the A_i and B_i. Since these subheaps thus take up $6k$ levels, overlap can be avoided only if $6k \geq n$ where n is the height of the heap. This inequality and the equation that defines k imply that, for this construction, n must be at least 36. The two constructions that demonstrate the worst and best behavior of Williams's and Floyd's Heapsorts respectively can be much smaller. They

only need to reserve $3k$ rows for the A_i and B_i and one row for the root, so n and k must satisfy the inequality $3k+1 \geq n$. In this case, it is sufficient for n to be at least 19.

The other issue that must be addressed also deals with the size of the heaps being constructed. In particular, all three constructions assumed that the number of keys N in the final heap was of the form $2^n - 1$. It is easy to see that the constructions are also valid for general N. The following discussion will justify this assertion for the construction given in Figures 4.4 and 4.5; the others follow similar lines.

Given N, let $n = \lceil \lg(N+1) \rceil$. Assuming $n \geq 36$, Figures 4.4 and 4.5 specify the construction of a heap on $2^n - 1$ keys. Let P be the prefix of this pulldown sequence that constructs a heap on N keys. It is the case that $B_W(P) = \frac{1}{2}N \lg N + O(N \log\log N)$ and $C_W(P) = \frac{1}{4}N \lg N + O(N \log\log N)$. This follows from two facts. First, the initial $2^{n-1}-2$ pulldowns of P have respective B_W and C_W costs of $\frac{1}{2}2^{n-1}n + O(2^{n-1}\log n)$ and $\frac{1}{4}2^{n-1}n + O(2^{n-1}\log n)$. Second, the last $N - 2^{n-1} + 1$ pulldowns of P (that create the truncated bottom row) have B_W and C_W cost equal to $\frac{1}{2}(N - 2^{n-1})n + O(N \log\log N)$ and $\frac{1}{4}(N - 2^{n-1})n + O(N \log\log N)$ respectively. This second claim results from the following four observations about the last $N - 2^{n-1} + 1$ pulldowns in P:

- The number of keys pulled down from the A_i and B_i is $\frac{1}{2}(N - 2^{n-1} + 1) + O(\max(n^3, (N - 2^{n-1} + 1)(\log n / n^2)))$; their total B_W cost is $\frac{1}{2}(N - 2^{n-1})n + O(N \log\log N)$ as desired.

- Although these pulldowns are from the side of the heap with low C_W cost, they still have C_W cost at least $\frac{1}{4}(N - 2^{n-1} + 1)\lg(N - 2^{n-1} + 1) + O(N \log\log N)$ which equals $\frac{1}{4}(N - 2^{n-1})n + O(N \log\log N)$.

- When a heap is constructed on $2^n - 1$ keys, the total B_W cost of pulldowns from C and D is $O(2^n \log n) = O(N \log\log N)$. It follows that when creating the truncated bottom row of $H(P)$, the B_W cost of pulldowns from C and D is $O(N \log\log N)$.

- Likewise, the C_W cost of pulldowns from C and D is $O(N \log\log N)$.

In conclusion, for arbitrary N sufficiently large, the previous four sections have shown how to construct heaps H with the following costs:

$$B_W(H) = \tfrac{1}{2}N\lg N + O(N\log\log N) \qquad C_W(H) = \tfrac{1}{4}N\lg N + O(N\log\log N)$$
$$B_W(H) = N\lg N + O(N) \qquad C_W(H) = \tfrac{3}{4}N\lg N + O(N\log\log N)$$
$$B_F(H) = O(N) \qquad C_F(H) = \tfrac{1}{4}N\lg N + O(N\log\log N)$$
$$B_F(H) = \tfrac{1}{2}N\lg N + O(N\log\log N) \qquad C_F(H) = \tfrac{3}{4}N\lg N + O(N\log\log N)$$

Chapter 5

Conclusions and Related Work

The previous four chapters presented technical results that count instruction executions by Williams's and Floyd's versions of Heapsort. This chapter concludes the thesis by applying these results and establishing the research context into which they fall.

5.1 Summary and Applications

Since this chapter puts the results of the previous chapters into perspective, the best place to begin is with a concise summary of those results. The following table summarizes the bounds that have been proved on the cost of heaps relative to Williams's and Floyd's Heapsorts:

	B_W Cost	C_W Cost	B_F cost	C_F cost
Best Case:	$\sim \frac{1}{2} N \lg N$	$\sim \frac{1}{4} N \lg N$	$O(N)$	$\sim \frac{1}{4} N \lg N$
Average Case:	$\sim N \lg N$	$\sim \frac{1}{2} N \lg N$	$O(N)$	$\sim \frac{1}{2} N \lg N$
Worst Case:	$\sim N \lg N$	$\sim \frac{3}{4} N \lg N$	$\sim \frac{1}{2} N \lg N$	$\sim \frac{3}{4} N \lg N$

By the first four parts of Lemma 2.1, the values specified for B_W, C_W, B_F, and C_F describe equally well the values of $B_{\mathcal{W}}$, $C_{\mathcal{W}}$, $B_{\mathcal{F}}$, and $C_{\mathcal{F}}$ respectively.

Now that these parameters are known, it is possible to perform a detailed analysis of Heapsort in the manner of Knuth. Fix an assembly level implementation of either

version of Heapsort. For any given instruction in this implementation, it is possible to determine how many times the instruction is executed. Assuming a simple machine where each instruction takes a fixed amount of time to execute, it is easy to figure the asymptotic running time of the implementation; all one has to do is multiply the execution time of an instruction by the number of times it is executed, and sum over all instructions in the implementation.

Real computers complicate this model somewhat. Code is typically written in a high level language; compilers can map each line of code to several instructions, not all of which may be executed the same number of times. Interrupts cause additional overhead to be associated with code execution. Even in the absence of interrupts, pipelining invalidates the assumption that total execution time is a linear combination of individual instruction execution times. Also, caching and virtual memory introduce variable memory access times.

Even in the face of these complications, however, the detailed analysis conducted by this thesis is a valuable predictor of an algorithm's performance. The techniques of analysis outlined above are as applicable to assembly code generated by a compiler as to assembly code generated by hand. Some of the behavior caused by pipelining can also be predicted by the results of this thesis. For example, a pipelined processor may predict that branch instructions are taken, and begin executing instructions at the target of a branch before the branch condition has been evaluated. In this case, it takes less time to execute a branch instruction when the branch is taken than when the branch is not taken; restarting the pipeline contributes pipeline latency to the execution time of an untaken branch. The results of this thesis can deal with such complications since they predict the frequency with which each branch is taken. The problems raised by interrupts and variable memory access times are more properly classified as systems questions than as questions in the analysis of algorithms.

The values determined in the preceding two chapters will now be applied to a specific implementation of Heapsort. In particular, B_W and C_W are essentially the same as the quantities that Knuth [18] calls B and C respectively, so they can be applied to Knuth's

MIX implementation of Williams's Heapsort. Knuth gives the following formula for the running time in MIX units of his implementation of Heapsort:

$$7A + 14B + 4C + 20N - 2D + 15\lfloor N/2 \rfloor - 28$$

where A and D are always $O(N)$. Plugging in C_W for C and B_W for B, yields the following asymptotic running times for Knuth's MIX implementation:

- $\sim 8N \lg N$ units in the best case.
- $\sim 16N \lg N$ units in the average case.
- $\sim 17N \lg N$ units in the worst case.

On the one hand, there is nothing shocking about these results. Knuth found empirically that the average case of his implementation required $16N \lg N + 0.2N$ units, and noted that "Heapsort has the interesting property that its worst case isn't much worse than the average." The results above certainly reflect these observations. On the other hand, this presentation is the first rigorous statement of these facts. It is also possible now to answer Knuth's exercise (5.2.3.30) which asks whether C ($= C_W$) differs from its average case, and if so, by how much. In the absence of the bounds above, Knuth had to give a conservative bound of $18N\lfloor \lg N \rfloor + 38N$ units on the maximum running time of his implementation.

5.2 Other work on Heapsort

It was mentioned earlier that the bounds that are presented here are the first bounds of their kind to be accompanied by proof. This does not imply that no research has been done on Heapsort. Papers have been published approaching Heapsort from several angles, ranging from rigorous bounds on small parts of the Heapsort process to analyizing the whole algorithm under empirically backed assumptions, to introducing new flavors of the basic Heapsort algorithm. That research will be discussed in this section.

Recall from Chapter 2 that $\Phi : H_N \rightarrow H_{N-1}$ is the map corresponding to the effects of one iteration of the main loop of Heapsort. Recall also that any heap in H_{N-1} can be transformed by Φ^{-1} into exactly $A[\lfloor N/2 \rfloor]$ distinct heaps in H_N. This means that Φ does not map R_N to R_{N-1}; $\Phi(R_N)$ takes on the value of each element $H \in H_{N-1}$ with probability proportional to the value of $A[\lfloor N/2 \rfloor]$ in H. Put simply, Heapsort does not preserve uniformity. Doberkat [9] acknowledges this fact and then proceeds to analyze the expected number of comparisons and data movements required when Williams's Heapsort performs the first iteration of its main loop. The primary results of [9] are that given R_N, where $N = 2^n$, it is expected that the first iteration of the main loop of Williams's Heapsort will perform $n - 1 + o(1)$ data movements and $2n - 1 + o(1)$ comparisons when moving a into place. Doberkat's methods assume that the main loop of Heapsort operates on a uniformly generated heap, so they don't work past the first iteration.

Doberkat's result falls short of a complete analysis of Heapsort. Carlsson [3] took a similar approach and got a bit further. Carlsson first obtained bounds on the expected number of comparisons that are performed during a single iteration of the main loop of Floyd's Heapsort, operating on R_N for any size N. Carlsson then made the assumption, based on empirical evidence and plausibility arguments, that a single iteration of the main loop of Heapsort makes at least as many comparisons when applied to R_N as it does when applied to the heap $\Phi^{M-N}(R_M)$ for $M > N$. Once this assumption is made, the average case analysis of Heapsort becomes much simpler. Carlsson could prove that the expected number of comparisons required by Floyd's Heapsort to sort R_N is at most $N \lg N - .98N + O(\log N)$. Wegener [26] makes more assumptions and comes up with a bound of $N \lg N - 1.26N + O(\log N)$, which supports both his empirical results and those of [3].

The authors of the papers discussed in the previous paragraph claim that these results demonstrate that Floyd's Heapsort is faster than Quicksort [26], or even the "the fastest in-place sorting algorithm. [3]" These claims may seem a bit dubious since they rest upon unproved assumptions and judge the efficiency of an algorithm on the basis of

comparisons alone. Nonetheless, they point out one of the few remaining open questions concerning Heapsort. Theorem 3.1 proved that on the average, Floyd's Heapsort does indeed require $N \lg N + O(N)$ comparisons when sorting R_N, but the constant under the Big Oh is rather large (essentially 9). Munro [20] has shown how to simplify the argument of Theorem 3.1 while reducing the constant under the Big Oh to at most about 5. It would be nice to see rigorous arguments that further reduce this constant to the range of the results in the previous paragraph.

Most results on Heapsort beyond those that have been discussed so far can be described as new variants of the algorithm. For example, Carlsson [4] developed a variant on Floyd's Heapsort that finds a path to the bottom of the heap, as before, and then does a binary search on the path to find the correct position for a. This guarantees a worst case of $N \lg N + N \lg\lg N + O(N)$ comparisons. On a similar note, Xunrang and Yuzhang [29] have modified Floyd's Heapsort to find a path 2/3 of the way down the heap; at this point a either needs to go farther down or back up. If a must go farther down, the path is continued in the manner of Williams's Heapsort with two comparisons per level; if it must go up, it is inserted by linear insertion into the path segment that has already been found. This guarantees at most $(4/3)N \lg N$ comparisons in the worst case. An earlier paper by Gonnet and Munro [16] supersedes both of these results by modifying Floyd's Heapsort as follows. A path is found to within $\lg\lg N$ of the bottom of the heap. If a needs to go back up, binary insertion can find its place in this path in $\lg\lg N$ comparisons. If a must farther down the heap, this procedure is repeated recursively. This procedure sorts a list using at most $N \lg N + N \log^* N + O(N)$ comparisons in the worst case. Also presented in [16] is a related adversary argument that demonstrates that in the worst case, at least $\lg N + \log^* N + O(1)$ comparisons must be performed when fixing a heap after deletion of its root.

Other variants on Heapsort have been proposed. For example, Knuth [18], exercise (5.2.3.28), suggests that sorting with ternary heaps could yield increased efficiency. Carlsson proposed "scattering the leaves" of a heap; that is, the children of node i become $i + a$ and $i + 2a$ where $a = 2^{\lfloor \lg i \rfloor}$. This distributes the bottom level of a

complete tree to ensure that as few nodes as possible have siblings. The result is to eliminate a few key comparisons that occur at the bottom of the heap thereby reducing the number of comparisons performed by a small linear quantity. McDiarmid and Reed [19] have modified Floyd's Heapsort to keep track of comparisons performed when finding the path of largest children to the bottom of the heap. Sometimes the key a does not have to go all the way to the bottom of the path that is found by Floyd's Heapsort; this means some unnecessary comparisons were performed at the bottom of the heap. By keeping track of the results of these comparisons, one can reduce the number of comparisons performed when sorting. While this variant of Heapsort has the disadvantage of requiring additional storage, Wegener [27] has shown by amortized analysis that it never performs more than $N \lg N + 1.1N$ comparisons.

5.3 Work on Heap Building

In the previous section, the last paper to be discussed [19] was actually a paper on heap building, not properly on Heapsort. In fact, research on heap building has been relatively more successful than research on Heapsort. This section thus describes results concerning heap building, including the analysis of makeheap from Chapter 1.

5.3.1 Floyd's Heap Building Algorithm

The best known heap building algorithm is the one that was proposed by Floyd [13] and which forms the basis for makeheap in Chapter 1. As was described earlier, this method takes two heap-ordered subtrees with a common parent a and merges them to form a single heap-ordered tree by moving a down the heap in the manner of the main loop of Heapsort. Although makeheap in Chapter 1 uses the main loop of Floyd's Heapsort to move a into position, Floyd's original presentation was based on the main loop of Williams's Heapsort. Unless otherwise noted, all further references to Floyd's heap building algorithm will assume the use of the main loop of Williams's Heapsort.

Knuth [18] performed first analysis of Floyd's heap building algorithm. Knuth used basic combinatorial arguments to determine that $0.7440N - 1.31 \ln N$ closely approximates the average number of data movements performed when building a heap from P_N where $N = 2^n - 1$. Knuth also gave values from which the expected number of comparisons can be determined to be about $1.8814N - 2.6 \ln N$. Doberkat [7] used a continuous model to duplicate Knuth's figure for the number of data movements, but erred when estimating the expected number of comparisons. Later, Doberkat [10] used generating functions to obtain bounds that are more accurate and precise. Doberkat computed the expectation and variance of the number of comparisons and interchanges when building a heap on $N = 2^n - 1$ keys. These results are summarized below:

Expected Data Movements:

$$(\alpha_1 + \alpha_2 - 2)N - n + \alpha_1 + \alpha_2 - \frac{n}{3N} + o\left(\frac{n}{N}\right)$$

Data Movement Variance:

$$(2 - \alpha_4)N - \alpha_4 + \frac{n^2}{3N} + o\left(\frac{n^2}{2^n}\right)$$

Expected Comparisons:

$$(\alpha_1 + 2\alpha_2 - 2)N - 2n + \alpha_1 + 2\alpha_2 + 1 - \frac{2n}{3N} + o\left(\frac{n}{N}\right)$$

Comparison Variance:

$$\left(4\alpha_3 - 4\alpha_1 - \alpha_2 - \frac{\alpha_4}{2} + \frac{9}{2}\right)N + 4\alpha_3 - 4\alpha_1 - \alpha_2 - \frac{\alpha_4}{2} + \frac{7}{2} + n + \frac{n^2}{6N} + o\left(\frac{n^2}{2^n}\right)$$

where

$$\alpha_1 = \sum_{i=1}^{\infty} \frac{1}{2^i - 1} = 1.6066951\ldots \qquad \alpha_2 = \sum_{i=1}^{\infty} \frac{1}{(2^i - 1)^2} = 1.1373387\ldots$$

$$\alpha_3 = \sum_{i=1}^{\infty} \frac{i2^{i+1}}{(2^{i+1} - 1)^2} = 1.3085437\ldots \qquad \alpha_4 = \sum_{i=1}^{\infty} \frac{i2^{i+1}(2^{i+1} + 1)}{(2^{i+1} - 1)^3} = 1.7387828\ldots$$

Doberkat also proved that for any N, Floyd's heap building algorithm expects to perform $(\alpha_1 + \alpha_2 - 2)N + O(\log N)$ comparisons and $(\alpha_1 + 2\alpha_2 - 2)N + O(\log N)$ exchanges to build a heap from P_N.

The version of Floyd's heap building algorithm that was presented as makeheap in Chapter 1 has also been analyzed. McDiarmid and Reed [19], Carlsson [3], and Wegener [26] have all determined that $1.65N$ comparisons are expected to be performed

when Floyd's heap building algorithm uses the main loop of Floyd's Heapsort to build a heap from P_N.

5.3.2 Williams's Heap Building Algorithm

Another well known heap building algorithm is the one that was proposed by Williams in the paper in which Heapsort was first presented [28]. Williams's method works by repeatedly inserting new keys into a heap. Given a heap on $N-1$ keys in the first $N-1$ positions of $A[1 \ldots N]$, a new key can be added as follows. First place the new key in position $A[N]$. Then repeatedly swap the new key with its parent until it either becomes smaller than its parent or reaches the root. The result is a heap on N keys. It is clear that, given N keys in $A[1 \ldots N]$, a heap can be built from them by considering $A[1]$ to be a heap on 1 key and repeatedly inserting the remaining keys.

Williams's heap building algorithm has two problems. The first is that it runs in $\sim N \lg N$ time in the worst case. If, for example, the keys are presented to the algorithm in increasing order, each key in succession will have to travel to the root of the heap. The second problem is that it does not map P_N to R_N. This greatly complicates the analysis of this algorithm. Nonetheless, it has been fairly thoroughly analyzed.

The first step toward the analysis of Williams's heap building algorithm was to show the effect of inserting a random key into a random heap. Porter and Simon [21] showed that given P_N where $N = 2^n - 1$, if a heap is built from the first $N-1$ keys by Floyd's heap building algorithm, then inserting the N'th key in the manner of Williams's algorithm is expected to require $1 - n/N$ data movements. They also showed for $N = 2^n$ that the expected number of data movements approaches α_1 when the N'th key is inserted in this manner, where α_1 is the constant used by Knuth and Doberkat in their analyses of Floyd's heap building algorithm. Porter and Simon also showed that the insertion of a random key into a random heap in this manner is never expected to take more than α_1 data movements for any heap size. Doberkat [8] extends these results, presenting in closed form both the expectation and variance of the number

of comparisons required to insert the N'th key when $N = 2^n - 1$; tight asymptotic expressions are also given to bound the expectation and variance of the number of comparisons when $N = 2^n$.

The work discussed in the previous paragraph has a major drawback. Because Williams's heap building algorithm does not map P_N to R_N, insertion into a random heap doesn't really say much about Williams's heap building algorithm. Porter and Simon and Doberkat recognized this problem but were unable to overcome it. Bollobás and Simon [2] presented the first arguments that truly bound the performance of heap building by repeated insertion. They concentrated on heaps with sizes $2^i - 1$ and examined the effect on the average key value in any given level of the heap each time a new level is added to the heap. From bounds on the expected size of keys in each level they determined how high each key is expected to rise as a new level is added to the heap. This yields a bound of $\phi N + o(N)$, $\phi = \sum_{i=1}^{\infty} 1/(2^i + 1)$, on the number of data movements required when a heap on $N = 2^n - 1$ keys is constructed by Williams's heap building algorithm.

Frieze [14] went somewhat further than Bollobás and Simon, giving a simpler proof of the results in [2], sketching how these results can be applied to d-ary heaps, and giving a non-trivial bound on the probability that a random permutation causes Williams's heap building algorithm to make substantially more than the expected number of data movements. Frieze also proved a weak but non-trivial lower bound of $(3/4)N + o(N)$ on the expected number of data movements.

Hayward and McDiarmid [17] have improved upon all of these results. Their main theorem states that there exists a constant ω, $1.2778 < \omega < 1.2995$ such that when Williams's heap building algorithm is applied to P_N, the expected number of data movements performed approaches $N\omega$ as N becomes large. This is not only a substantial improvement of both the upper and lower bounds proved by [2] and [14], this bound also applies to general N and not just to N of the form $2^i - 1$. The proof given for these bounds entailed two parts. First, it was shown that when a key is inserted, the number of data movements required is expected to increase with the number of levels in the heap,

eventually approaching a constant ω. Second, an algorithm was developed to compute the expected number of data movements when the ith key is inserted; the output of this algorithm for large heaps was then used to compute bounds on ω. Also proved in [17] is that given $\epsilon > 0$, the probability that Williams's heap building algorithm performs more than $N(\omega + \epsilon)$ data movements to build a heap from P_N is $o(e^{-N/\log^4 N})$. Together, these results form a relatively complete analysis of Williams's heap building routine.

5.3.3 Bounds

So far, this section has concentrated on the analysis of the two classical heap building algorithms. The two algorithms that have been discussed are both practical and efficient ways to build a heap from a random list. One might still ask, however, how much one can improve upon the number of comparisons performed by these algorithms in the average and worst cases.

The heap building algorithm with the best known worst case performance is due to Gonnet and Munro [16]. Given a list of keys, this algorithm first constructs heap ordered binomial trees which are then "converted" to ordinary heaps. The algorithm can build a heap on N keys for arbitrary N in at most $1.625N + O(\log^* N \lg N)$ comparisons. Both [16] and [5] conjecture that this algorithm is optimal in the worst case. According to [16] and a correction in [19], this algorithm can be modified to require $(177/112)N + o(N)$ comparisons in the average case. McDiarmid and Reed [19] have presented a heap building algorithm that achieves a better average case comparison count of about $1.521288N$. Recall from the end of Section 5.2 that the main loop of Floyd's Heapsort can be modified to keep track of extra comparisons at the bottom of the heap. McDiarmid and Reed's heap building algorithm is what results when Floyd's heap building algorithm employs this modified main loop of Heapsort — essentially makeheap with a memory. McDiarmid and Reed have conjectured that this heap building algorithm is asymptotically optimal in the average case.

On the lower bound side, Gonnet and Munro [16] used straightforward informa-

tion theoretic arguments to show that, in both the average and worst cases, at least $(1.3644\ldots)N + O(N)$ comparisons are required to build a heap. This result has recently been improved upon by Carlsson and Chen [5], who presented an adversary argument to prove that, in the worst case, at least $(1.5)2^n - n - 2$ comparisons must be performed to build a heap on $2^n - 1$ keys.

This leaves a gap between the best known algorithms for building heaps and lower bounds on the number of comparisons required in both the worst and average cases. To conclude, then, this leaves three basic open problems concerning Heapsort:

1. Tighten the error estimates for Floyd's and Williams's Heapsorts. The leading terms of the important quantities are known, but empirical evidence suggests that the error bounds are still somewhat loose.

2. Determine the number of comparisons required, asymptotically, to construct a heap in the worst and average cases.

3. Further refine the, already rather tight, bounds on the expected performance of Williams's heap building algorithm.

Appendix A

LEMMA 3.2: Let a_1, a_2, and a_3 be real variables and let p, q, and r be positive real constants. Suppose that the constraints:

$$a_1 + a_2 + a_3 = p \qquad (r-1)a_1 + (r)a_2 + (r+1)a_3 = q \qquad a_i \geq 0, (i = 1,2,3)$$

are satisfied by some assignment to the a_i where at least two a_i are positive. Then $a_1^{a_1} a_2^{a_2} a_3^{a_3}$ is minimized subject to these constraints by exactly one assignment to the a_i for which all the a_i are positive; this assignment satisfies the relation $a_1/a_2 = a_2/a_3$. ($a_i^{a_i}$ is assumed to equal 1 when $a_i = 0$.)

Proof: Let (x_1, x_2, x_3) be an assignment to the a_i that satisfies the constraints with $a_i > 0$ for at least two i. Then all assignments to the a_i that satisfy the constraints lie along the line $(x_1 + b, x_2 - 2b, x_3 + b)$. The goal is thus to describe the minimum of the function $f(b) = (x_1+b)^{(x_1+b)}(x_2-2b)^{(x_2-2b)}(x_3+b)^{(x_3+b)}$ where b ranges in the interval $[\max\{-x_1, -x_3\}, x_2/2]$. This minimum may occur either at one of the endpoints (where 0^0 is taken to equal 1), or in the open interval inbetween. The minimum of f in the open interval is found by minimizing the logarithm of f, namely:

$$g(b) = (x_1 + b)\ln(x_1 + b) + (x_2 - 2b)\ln(x_2 - 2b) + (x_3 + b)\ln(x_3 + b).$$

The first derivative of g is:

$$g'(b) = 1 + \ln(x_1 + b) - 2 - 2\ln(x_2 - 2b) + 1 + \ln(x_3 + b).$$

Setting $g'(b) = 0$ and exponentiating the result gives:

$$\frac{x_1 + b}{x_2 - 2b} = \frac{x_2 - 2b}{x_3 + b}. \tag{A.1}$$

Expression (A.1) establishes the desired relation $a_1/a_2 = a_2/a_3$ at minimality, assuming that (A.1) is satisfied for some b in the appropriate interval, and that it indeed represents a minimum.

Clearly, there is a b in $(\max\{-x_1, -x_3\}, x_2/2)$ that satisfies (A.1) since $(x_1+b)(x_3+b)$ is zero at the left endpoint and positive at the right, while $(x_2 - 2b)^2$ is positive at the left endpoint and zero at the right. This further implies that (A.1) is satisfied by a unique b in this interval since (A.1) is a quadratic.

Next it must be shown that $g(b)$ is minimized when b satisfies (A.1). The second derivative of g is:

$$g''(b) = \frac{1}{x_1 + b} + \frac{4}{x_2 - 2b} + \frac{1}{x_3 + b} > 0.$$

Since f is continuous and approaches limiting values at its endpoints, it follows that the value of b in $(\max\{-x_1, -x_3\}, x_2/2)$ that minimizes f is a minimum for f in the closed interval as well. □

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] Béla Bollobás and Istvan Simon. Repeated random insertion into a priority queue. *Journal of Algorithms*, 6:466–477, 1985.

[3] Svante Carlsson. Average-case results on Heapsort. *BIT*, 27:2–17, 1987.

[4] Svante Carlsson. A variant of Heapsort with almost optimal number of comparisons. *Information Procesing Letters*, 24:247–250, 1987.

[5] Svante Carlsson and Jingsen Chen. The complexity of heaps. In *Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 393–402, 1992.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[7] Ernst-Erich Doberkat. Some observations on the average behavior of Heapsort (preliminary report). In *21st Annual Symposium on Foudations of Computer Science*, pages 229–237, 1980.

[8] Ernst-Erich Doberkat. Inserting a new element into a heap. *BIT*, 21:255–269, 1981.

[9] Ernst-Erich Doberkat. Deleting the root of a heap. *Acta Informatica*, 17:245–265, 1982.

[10] Ernst-Erich Doberkat. An average case analysis of Floyd's algorithm to construct heaps. *Information and Control*, 61:114–131, 1984.

[11] R. Fleischer. A tight lower bound for the worst case of Bottom-Up-Heapsort. Technical Report MPI-I-91-104, Max-Planck-Institut für Informatik, 1991.

[12] R. Fleischer, B. Sinha, and C. Uhrig. A lower bound for the worst case of Bottom-Up-Heapsort. Technical Report A23/90, University of Saarbrücken, 1990.

[13] Robert W. Floyd. Algorithm 245, Treesort3. *Communications of the ACM*, 7:701, 1964.

[14] Alan M. Frieze. On the random construction of heaps. *Information Processing Letters*, 27:103–109, 1988.

[15] G. H. Gonnet and R. A. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, MA, second edition, 1991.

[16] Gaston H. Gonnet and J. Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15:964–971, 1986.

[17] R. Hayward and C. J. H. McDiarmid. Average case analysis of heap building by repeated insertion. *Journal of Algorithms*, 12:126–153, 1991.

[18] Donald E. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[19] C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10:352–365, 1989.

[20] J. Ian Munro. Private Communication.

[21] Thomas Porter and Istvan Simon. Random insertion into a priority queue structure. *IEEE Transactions on Software Engineering*, SE-1:292–298, 1975.

[22] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipies, The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1986.

[23] Russel Schaffer and Robert Sedgewick. The analysis of Heapsort. Technical Report CS-TR-330-91, Princeton University, Department of Computer Science, 1991.

[24] Robert Sedgewick. Private Communication.

[25] Robert Sedgewick. *Algorithms, Second Edition*. Addison-Wesley, Reading, MA, 1988.

[26] Ingo Wegener. Bottom-Up-Heap Sort, a new variant of heap sort beating on average Quick Sort (if n is not very small). In *MFCS 1990, Lecture Notes in Computer Science 452*, pages 516–522, 1990.

[27] Ingo Wegener. The worst case complexity of McDiarmid and Reed's variant of Bottom-Up-Heap Sort is less than $n \log n + 1.1n$. In *STACS 1991, Lecture Notes in Computer Science 480*, pages 137–147, 1991.

[28] J. W. J. Williams. Algorithm 232, Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[29] G. Xunrang and Z. Yuzhang. A new HEAPSORT algorithm and the analysis of its complexity. *The Computer Journal*, 33:281–282, 1990.