Heap Building Bounds

Zhentao Li[1] and Bruce A. Reed[2]

[1] School of Computer Science, McGill University
zhentao.li@mail.mcgill.ca
[2] School of Computer Science, McGill University
breed@cs.mcgill.ca

Abstract. We consider the lower bound for building a heap in the worst case and the upper bound in the average case. We will prove that the supposedly fastest algorithm in the average case[2] does not attain its claimed bound and indeed is slower than that in [6]. We will then prove that the adversarial argument for the claimed best lower bound in the worst case[1] is also incorrect and the adversarial argument used yields a bound which is worse than that in [5] given by an information theory argument. Finally, we have proven a lower bound of $1.37n + o(n)$ for building a heap in the worst case.

1 Introduction

Heaps are a classical and commonly used implementation of priority queues. They are so fundamental that computer science students typically learn about them in their first year of university study. In this paper, we discuss bounds on building heaps.

We will prove that the supposedly fastest algorithm in the average case[2] does not attain its claimed bound and indeed is slower than that in [6]. We will then prove that the adversarial argument for the claimed best lower bound in the worst case[1] is also incorrect and the adversarial argument used yields a bound which is worse than that in [5] given by an information theory argument. Finally, we have proven a lower bound of $1.37n + o(n)$for building a heap in the worst case. Forthwith the details.

A heap [7, 4, 3] is a binary tree in each node of which we have stored a key. The tree has a special shape. All of its levels are full except the last one. The nodes on the last level are all as much to the left of the tree as possible. A min-heap has the property that every node has value less than or equal to its children. All heaps in this paper are min-heaps. A perfect heap is a heap whose last level is full.

The height of a tree is defined as the number of arcs of the longest path from the root to a leaf. Therefore, a perfect heap of height k has $2^{k+1} - 1$ nodes.

One of the attractive features of heaps is that they can be implemented using an array where the children of a node at A[i] are located at A[2i] and A[2i+1].

We will only consider building heaps in the comparisons model. That is, at each step, an algorithm chooses two keys and compares them to determine

F. Dehne, A. López-Ortiz, and J.-R. Sack (Eds.): WADS 2005, LNCS 3608, pp. 14–23, 2005.

which is bigger. Since we are dealing with min-heaps, we will call the winner of a comparison the key that is smaller and the loser of a comparison the key that is bigger.

The first heap-building algorithm due to Williams [7] runs in $O(n \log(n))$ by inserting the keys one by one. More precisely, it's worst case running time is $\sum_{i=1}^{n-1} \lfloor \log(i+1) \rfloor$. The key is added in a new leaf node so that the tree remains heap-shaped and "bubbled up" the tree until the heap order is restored.

A classical algorithm of Floyd [4] for building heaps from the bottom up yields an upper bound of $2n$ comparisons. This algorithm builds a heap on $n = 2^k - 1$ nodes by first recursively builds 2 heaps of size $2^{k-1} - 1$. It then "trickle down" another node to merge these two heaps. An information theory lower bound of $1.364n$ comparisons to build a heap on n keys is shown in [5]. An algorithm which uses $1.521n$ comparisons on average is developed in [6] by combining ideas from Floyd's and Williams algorithm. Faster algorithms for building heaps in the worst case were developed with the aid of binomial trees.

The binomial tree of size 1 (height 0) is a single node. A binomial tree of height k is defined as follows: It has a root that has k children. The subtrees rooted at the children of the root are binomial trees of each of heights 1 to $k-1$. As in the min-heap, every node has a key whose value less or equal to that of its children. Clearly a binomial tree of height k has 2^k nodes.

A binomial tree on 2^k nodes can be built recursively using $2^k - 1$ comparisons by first building two binomial trees of 2^{k-1} nodes and then comparing the keys at their roots. This is clearly best possible since we know that the root contains the min, any key that has not lost at least once could still be the min and each comparison can only make one key, that has not yet lost, lose.

Faster algorithms for building heaps on 2^k elements first build a binomial tree and then recursively convert this into a heap. As discussed in [5], this approach can be used to build a heap on n nodes in $1.625n + o(n)$ comparisons in the worst case.

The contributions of this paper are threefold:

1. The algorithm shown in [2] claims to have an average case running time of $1.5n + o(n)$ or faster. We will show that the analysis gives a lower bound of $\frac{43}{28}n + o(n)$ which is slower than the algorithm shown in [6].
2. The authors of [1] claim that their adversary yields a lower bound of $1.5n + o(n)$ comparisons in the worst case. We will show that this adversary yields a lower bound which is at best $\frac{5}{4}n + o(n)$ comparisons. This is worse than that of the information theory lower bound of $1.364n$ comparisons [5].
3. We have proven a new lower bound of $1.3701\ldots n + o(n)$ for building heaps.

In what follows, we consider only heaps of size 2^k and $2^k - 1$. This is not really a restriction. For example, to build a heap with 23 elements,we can first build the 15 element heap rooted at the left child of the root, then build the 7 elements heap rooted at the right child of the root and then "trickle down" the remaining element from the root using $2\log(n)$ comparisons (see [3]) to construct our heap. In the same vein, if we can construct heaps of size $n = 2^k - 1$ in

$\alpha n + o(n)$ comparisons for all k, then we can build heaps of any size in $\alpha n + o(n)$ comparisons.

A pseudo-binomial tree is a binomial tree with one leaf missing somewhere in the tree.

2 Average Case Algorithm

The algorithm described by Carlsson and Chen [2] is as follows:

1. To build a perfect heap of size $n = 2^k - 1$, first build a binomial trees of height i for $i = 1, 2, 4, \ldots, 2^{k-1}$.
2. Repeatedly compare the keys of the roots of two smallest trees until a pseudo-binomial tree of size $2^k - 1$ is created. Note that these first two steps take $n - 1$ comparisons in total.
3. Let $\bar{T}(2^k - 1)$ denote the number of comparisons required on average by this algorithm to transform a pseudo-binomial tree of size $2^k - 1$ into a heap. Note that $\bar{T}(1) = \bar{T}(3) = 0$ since these pseudo-binomial trees are heaps.
   Note that the subtree rooted at the children of the root are all binomial trees except for one which is a pseudo-binomial tree. If $k < 2$ then we have constructed the desired heap. otherwise we proceed depending on where the missing leaf is:
   Case 1. If the largest subtree of the root is the pseudo binomial tree, recurse on it. Then compare the keys of the roots of the other subtrees of the root to create a pseudo-binomial tree (i.e.:same as step 2) and recurse on it.
   This takes $\bar{T}(2^{k-1} - 1) + k - 2 + \bar{T}(2^{k-1} - 1)$ comparisons. It happens only if the min was in the largest binomial tree (before step 2) and this occurs $\frac{n+1}{2n}$ of the time as discussed in [2].
   Case 2. Otherwise, transform the largest subtree, T, of the root R, which is now a binomial tree, into a heap plus an extra element x. We do this as follows: The root r of T will be the root of the heap. The subheap rooted at the right child of r will be formed from the union of the elements of the subtrees rooted at the children of r except for the largest. The largest subtree of T rooted at the left child of r will be used to form the subheap rooted at the left child of r (and will yield an extra element). To form the right subheap, we build a pseudo-binomial tree from the union of the trees under consideration and we recursively apply this algorithm starting at step 2. To form the subheap rooted at the left child of r, we recursively apply the procedure described in this paragraph.
   At this point, we have built a heap on $T - x$. We now need to build a heap on the elements not in $T - x + r$. To do so, we consider x as a child of R. Recurse on the children of R, excluding T but including x, starting from step 2.
   According to Carlsson and Chen, this takes $\sum_{i=2}^{k-2} \big((i-1) + \bar{T}(2^i - 1)\big) + k - 1 + \bar{T}(2^{k-1} - 1)$ comparisons on average. Since $\bar{T}(3) = 0$ this is just

$2 + \sum_{i=3}^{k-1} \left((i-1) + \bar{T}(2^i - 1)\right)$ comparisons. This happens $\frac{n-1}{2n}$ of the time.

4. Stop the recursion at the heaps of size 7 and build them in $\frac{9}{7}$ comparisons as discussed in [2].

It seems to us that Carlsson and Chen's analysis is faulty as they ignore important conditioning on x. We show now that their analysis is faulty even assuming their conditioning assumptions are correct. Accepting their hypothesis, we have:

Theorem 1. *For the algorithm in [2],* $\bar{T}(2^k - 1) \geq \frac{15}{28}2^k - k \quad \forall k \geq 3$

Proof. First note that $\bar{T}(7) = \frac{9}{7} \geq \frac{15}{28}2^k - k$ for this algorithm.

If $\bar{T}(2^i - 1) \geq \frac{15}{28}2^i - i$ for $i = 3, \ldots, k-1$ and $k \geq 4$ then

$$\begin{aligned}
\bar{T}(2^k - 1) &= \frac{n-1}{2n}\left(2 + \sum_{i=3}^{k-1}\left((i-1) + \bar{T}(2^i - 1)\right)\right) \\
&\quad + \frac{n+1}{2n}\left(\bar{T}(2^{k-1} - 1) + k - 2 + \bar{T}(2^{k-1} - 1)\right) \\
&= \frac{n-1}{2n}\left(2 + \sum_{i=3}^{k-1}\left((i-1) + \bar{T}(2^i - 1)\right)\right) \\
&\quad + \frac{n+1}{2n}\left(k - 2 + 2\bar{T}(2^{k-1} - 1)\right) \\
&\geq \frac{n-1}{2n}\left(2 + \sum_{i=3}^{k-1}\left((i-1) + \frac{15}{28}2^i - i\right)\right) \\
&\quad + \frac{n+1}{2n}\left(k - 2 + 2\left(\frac{15}{28}2^{k-1} - k + 1\right)\right) \\
&= \frac{n-1}{2n}\left(2 + \frac{15}{28}\left(2^k - 8\right) - (k-3)\right) \\
&\quad + \frac{n+1}{2n}\left(\frac{15}{28}2^k - k\right) \\
&= \frac{n-1}{2n}\left(\frac{15}{28}2^k - k + \frac{5}{7}\right) + \frac{n+1}{2n}\left(\frac{15}{28}2^k - k\right) \\
&\geq \frac{n-1}{2n}\left(\frac{15}{28}2^k - k\right) + \frac{n+1}{2n}\left(\frac{15}{28}2^k - k\right) \\
&= \frac{15}{28}2^k - k
\end{aligned}$$

$\therefore \bar{T}(2^k - 1) \geq \frac{15}{28}2^k - k$

By induction, $\bar{T}(2^k - 1) \geq \frac{15}{28}2^k - k \quad \forall k \geq 3$

Note that this implies that the algorithm in [2] is worse than the algorithm in [6].

3 Worst Case Adversary

The adversary described by Carlsson and Chen [1] does the following: For all keys x, define $Up(x) = \{y|y < x\}$ and $Down(x) = \{y|y \geq x\}$. When comparing two keys x and y, the adversary will answer as follows:

If $x \in Up(y)$, we must answer $x < y$. If $y \in Up(x)$, we must answer $y < x$.

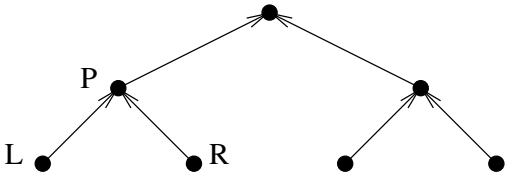
If $x \notin Up(y)$ and $y \notin Up(x)$ then answer $x < y$ according to the first rule that can apply:

Rule 1. If $\|Down(x)\| > \|Down(y)\|$ then x is the winner, otherwise
Rule 2. if $\|Up(x)\| < \|Up(y)\|$ then x is the winner.
Rule 3. For all other cases, answer $x < y$.

We will now show a counter-example for which the adversarial argument given in [1] fails to attain the claimed bound of $1.5n$.

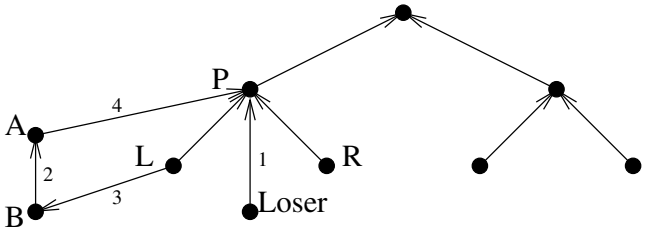
Theorem 2. *Given a complete heap H of height $k \geq 2$, in which the key at its leaves have never won, a key Loser which has never won, and a set S of $2^{k+1}+2^{k+2}$ keys which have not yet been compared, we can build in $\frac{5}{4}(2^{k+1}+2^{k+2})$ comparisons, against this adversary, a heap H' of height $k+2$ containing S and all the nodes of H such that no leaf of H' contains a key which has won a comparison.*

We proceed in the following way: We consider a node P with both children being leaves. We call the key at the left child L and the key at the right child R.



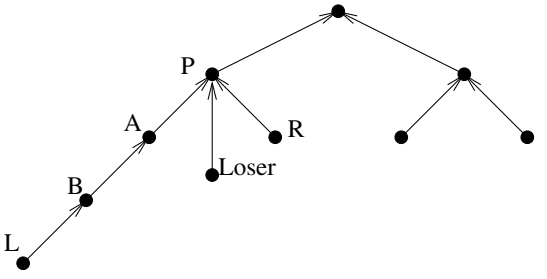
It is enough to prove the theorem for heaps of size 7 as we can treat the 2^{k-2} heaps of size 7 at the bottom of H separately. In order to add 12 nodes to this heap, we will do the following:

Step 1. – *Compare Loser to P. Loser will lose since $\|Down(Loser)\|$=1 (and $\|Down(Loser)\|$ remains 1 after this comparison).*

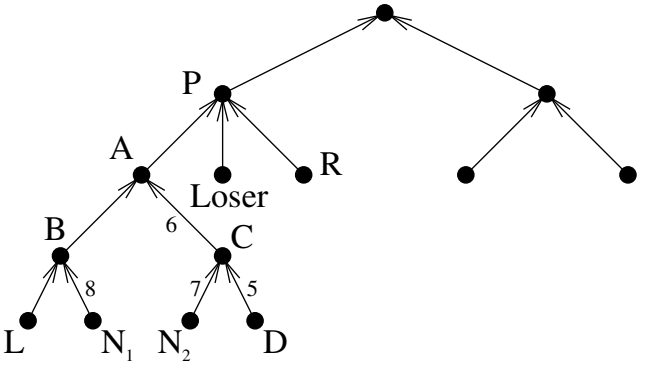


A number n on an edge is the nth comparison that we are making.

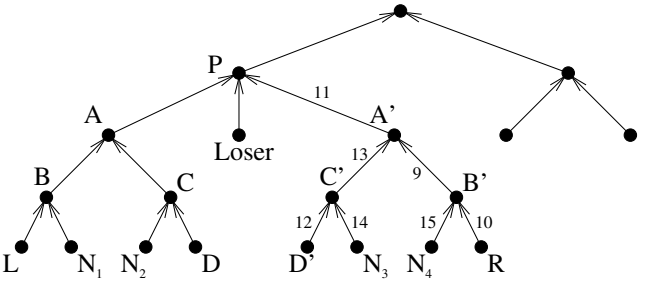
Step 2. – *Compare two keys in S and call the winner A and the loser B.*
- *Compare B to L. B will win since* $\|Down(L)\| = \|Down(B)\| = 1$ *and* $\|Up(L)\| \geq 2$ *while* $\|Up(B)\| = 1$.
- *Compare P to A. P will win since* $\|Down(P)\| \geq 4$ *and* $\|Down(A)\| = 3$. *We can redraw the tree to record the current information:*



- *Compare two more keys in S and call the winner C and the loser D.*
- *Compare C to A. C will lose since* $\|Down(C)\|$*=2 and* $\|Down(A)\|$*=3.*
- *Compare a key* N_1 *in S to B and a key* N_2 *in S to C. The new keys will lose since* $\|Down(B)\|$*=2,* $\|Down(C)\|$*=2 and* $\|Down(N_1)\|$*=*$\|Down(N_2)\|$*=1.*



Step 3. *Do step 2 on R and P instead of L and P.*

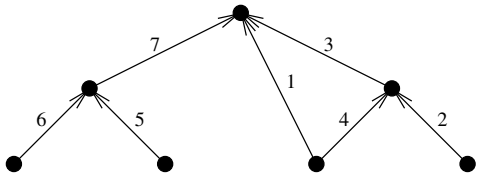


We have taken 15 comparisons to add 12 keys (Loser doesn't count as an added key). We can repeat this process using the same Loser key.

Note that we did not use the fact that the adversary chooses arbitrarily if both $\|Up\|$ and $\|Down\|$ are equal.

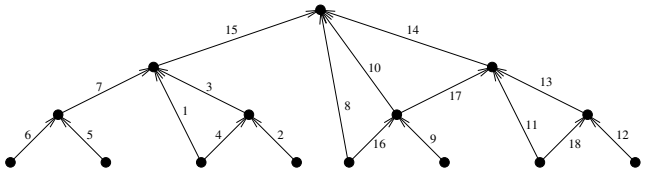
Also note that the only property that we used was that $\|Down(L)\| = 1$ (and $\|Down(R)\| = 1$), $\|Up(L)\| > 1$ (and $\|Up(R)\| > 1$) and $\|Down(P)\| \geq 2$). These properties are kept for the keys on the last two levels after we have inserted the new keys.

Here is a possible way of building the initial 7 nodes heap:



Note that we put the winner of the 3rd comparison at the root. This allows us to build perfect heaps of odd height. To build perfect heaps of even height, we can just start with a heap of 15 nodes instead.

Here is a possible way of building the 15 nodes heap:



Now we can use Gonnet and Munro's algorithm [5] to build heaps of any height from perfect heaps.

Therefore, the lower bound that the adversary provides is at most $\frac{5}{4}n + O(\log^2 n)$ which is worse than the information theory lower bound of $1.362n$.

4 A Simple Adversary

We now describe an adversary which yields a lower bound of $1.3701\ldots n + o(n)$ comparisons for building a heap H on $2^k - 1$ elements.

Since we are dealing with min-heaps, we will call the winner of a comparison the key that is smaller and the loser of a comparison the key that is bigger.

The adversary decides how to answer comparisons by looking at the first loss graph. This is a directed acyclic graph which contains, for every node x which has lost, an edge from x to the first node to which it lost. These are the only edges of the graph. Note that each component of this graph is a tree all of whose edges are directed towards some root. Note further that this graph changes as the algorithm progresses. Initially, it is empty and when the algorithm terminates, it has $n - 1$ edges.

There are $n - 1$ comparisons which are first losses and hence correspond to edges of the final first loss graph. We bound the number of comparisons used in building the heap which are not part of the final first loss graph. If in the final

first loss graph, everybody but the minimum lost to the minimum, then since all but the top three nodes of the heap lose to somebody who is not the minimum, there must be at least $n-3$ such extra comparisons in total.

More generally, our approach is to try to ensure that there are many vertices of large indegree high up (i.e. close to the root) in the final first loss graph.

Ideally, we would like the indegree in the first loss tree of a node to be less than the corresponding value for its parent. This is difficult to ensure as the indegree of x may increase after its first loss. So instead, we colour an edge xy of the first loss graph red if x lost to y before y had lost and blue otherwise. We let $a(x)$ be the red indegree of x. We note that $a(x)$ can only increase during the heap building process and that after x loses, $a(x)$ does not change. So if we insist that:

1. When comparing two nodes, the node with the higher a value wins
2. If the two nodes have equal a value then the node which has not yet lost wins.

then the $a(x)$ value of any node is indeed strictly less than that of its parent in the final first loss tree.

We use $b(x)$ to denote $a(y)$ where y is the parent of x in the first loss graph. If x has not yet lost, $b(x)$ is undefined.

We analyze this adversary using a LP. We define some variables such as $p_0 = \|\{x|a(x) = 0\}\|/n$ and $p_{(0,i)} = \|\{x|a(x) = 0, b(x) = i\}\|/n$ by looking at the first loss graph. We also define variables depending on the shape of the final heap that is built. They includes $q_{(0,1)} = \|\{x|a(x) = 0, b(x) = 1$ and x is a leaf $\}\|/n$ as well as $q_{(0,1,4,5,2,3)} = \|\{x|a(x) = 4, b(x) = 5,\ x$ is not a leaf, x has children c_1, c_2 and $a(c_1) = 0, b(c_1) = 1, a(c_2) = 2, b(c_2) = 3\}\|/n$. Our LP has a total of 4099 variables.

Recall that we consider only heaps on $n = 2^k - 1$ nodes. Thus every internal vertex has two children.

With these variables, we define some constraints by simply counting the nodes. An example of such constraint is $\sum p_i = 1$. We also have constraints due to the structure of a heap such as $\sum q_{(i,j)} = 0.5 + \frac{1}{2n}$ since there are 2^{k-1} leaves in a heap on $2^k - 1$ elements. We note that instead of an equation with RHS $= 0.5 + \frac{1}{2n}$, we use two inequalities which this implies. One with RHS $\leq \frac{1}{2}$ and the other with RHS $\geq \frac{1}{2} + \epsilon$ for a small but fixed ϵ. Our final analysis uses 209 constraints.

To give a flavour of the LP, we close this section by proving here a lower bound of $(1 + \frac{1}{13})n - 1$ on the number of comparisons needed to build a heap. We actually consider the number of comparisons not in the first loss graph. We denote this number by $Extra$. We also use p_1^* to denote $\|\{x|a(x) = 1, x$ has a unique child in the first loss graph$\}\|/n$

Now, consider a node y with $a(y) \geq 1$. When $a(y)$ increases from 0 to 1, y won a comparison against a node x. Since y won, $a(x)$ had value 0. Since $a(y)$ increased, x had not yet lost and xy is a red edge of the first loss graph. It follows that for all i, $p_{(0,i)} \geq p_i$. Thus, $p_0 \geq \frac{1}{2}$ and more strongly

$$p_0 \geq \frac{1}{2} + \frac{1}{2}(p_{(0,i)} - p_i) \tag{1}$$

If y has $a(y) \geq 2$ then for $a(y)$ to become 2, it either has to have at least two children in the first loss graph with a value 0, or one child with a value 0 and another with a value 1. It follows easily that:

$$p_0 + p_1^* \geq \frac{3}{5} \tag{2}$$

$$p_0 + p_1^* \geq \frac{2}{3} - \frac{1}{3}(p_{(0,1)} - p_1) \tag{3}$$

Now, if a vertex x with $a(x) = 0$ is a non-leaf of the heap, then x must win a comparison and this is not a comparison of the first loss graph. If a vertex x with $a(x) = 1$ which has a unique child z in the first loss graph, is a non-leaf of the heap, then one of z or x must win a comparison which is not a comparison of the first loss graph. Since there are at most $\frac{n}{2} + \frac{1}{2}$ leaves, amortizing over x and z in the second case, we have:

$$Extra \geq \frac{1}{2}(p_0 n + p_1^* n - (\frac{n}{2} + \frac{1}{2})) \tag{4}$$

Combining this with (2) gives

$$Extra \geq \frac{n}{20} - \frac{1}{4} \tag{5}$$

The point of our LP is that we can combine one such argument with others. For example, if we consider only x with $a(x) = 0$, for each such x which is not a leaf in the heap, each child y of x in the heap has $a(x) = 0$, and must have lost twice so we have:

$$Extra \geq 2(p_0 n - \frac{n}{2} - \frac{1}{2}) \tag{6}$$

Combining (3) with (4) gives:

$$Extra \geq \frac{n}{12} - \frac{1}{6}(p_{(0,1)} - p_1)n - \frac{1}{4} \tag{7}$$

Combining (6) with (1) gives:

$$Extra \geq (p_{(0,1)} - p_1)n - 1 \tag{8}$$

Finally, combining (7) and (8) which arise via two similar but different argument give

$$Extra \geq \frac{n}{13} - 1 \tag{9}$$

Our LP of course involves much more sophisticated arguments and many more than two of them. Details can be found at

`www.cs.mcgill.ca/~zli47/heaps.html`

5 Conclusions

We have shown that the analysis of the average case running time of algorithm [2] is incorrect. We have also shown that the lower bound on heap building from [1] is incorrect. The full proof of our adversary which yields a lower bound of $1.37n + o(n)$ is available at `www.cs.mcgill.ca/~zli47/heaps.html` .

References

1. Svante Carlsson and Jingsen Chen. The complexity of heaps. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 393–402. SIAM, 1992.
2. Svante Carlsson and Jingsen Chen. Heap construction: Optimal in both worst and average cases? In *Algorithms and Computation*, pages 254–263. Springer, 1995.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Sten. *Introduction to Algorithms*. The MIT Press, McGraw-Hill Book Company, 2 edition, 2001.
4. Robert W. Floyd. Algorithm 245: Treesort. *Commun. ACM*, 7(12):701, 1964.
5. Gaston H Gonnet and J Ian Munro. Heaps on heaps. *SIAM Journal of Computing*, 15(4):964–971, 1986.
6. C. J. McDiarmid and B. A. Reed. Building heaps fast. *J. Algorithms*, 10(3):352–365, 1989.
7. J. W. J. Williams. Algorithm 232: Heapsort. *Commun. of the ACM*, 7(6):347–348, 1964.