Heaps and Balanced Trees

For in-class use only in CSC 501/401 course

Dr. Marek A. Suchenek ©

April 10, 2014

1 Heaps and Balanced Trees

1.1 Binary representations of positive natural numbers

How many bits are needed to represent a number $M > 0$ in binary? Let's say it's n. We have:

$$M \leq \underbrace{11\ldots1}_{n}$$

$$\underbrace{11\ldots1}_{n}+1 = 1\,\underbrace{00\ldots0}_{n} = 2^n$$

So,

$$\underbrace{11\ldots1}_{n} = 2^n - 1$$

or

$$M \leq 2^n - 1$$

or

$$M + 1 \leq 2^n$$

or

$$\log_2(M+1) \leq n$$

or

$$\lceil \log_2(M+1) \rceil \leq n$$

or

$$\lfloor \log_2 M \rfloor + 1 \leq n.$$

So, the smallest n that is large enough is $\lfloor \log_2 M \rfloor + 1$; that is how bits are needed to represent number $M > 0$ in binary.

Exercise. Do you see a set of binary sequences on Figure 1? Do you see a complete binary tree there? If so then explain why.

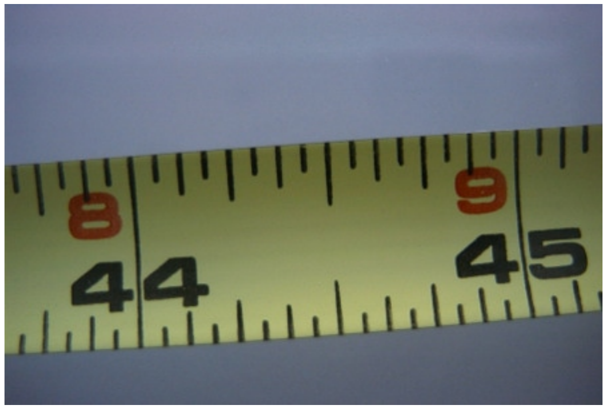


Figure 1: Do you see a set of binary sequences and a complete binary tree here?

1.2 Heaps

A heap is a contiguous, partially ordered binary tree. *Contiguous* means that all levels of the tree in question, except, perhaps, for the last level, contain the maximum number of nodes, and if the last level of the tree contains a lesser number of nodes then they are flushed all the way to the left.

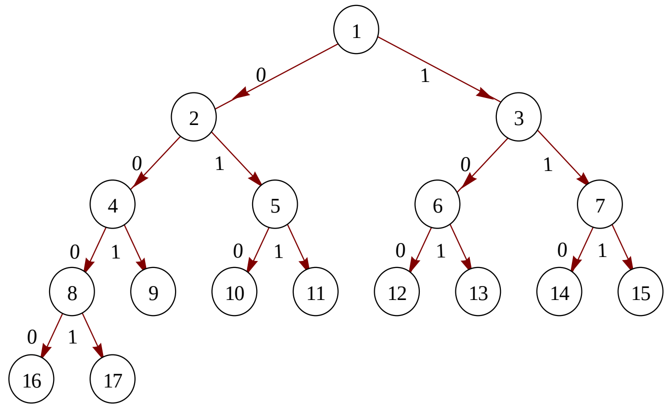


Figure 2: A heap with 17 nodes showing nodes' ordinal numbers of nodes in decimal.

Figure 2 visualizes an example of heap with 17 nodes. It shows nodes' ordinal numbers in decimal. Their binary representations are of the form: 1 followed by a sequence of edges' labels along the path from the root to the node in question. For instance, the sequence of labels along the path from the root to node number 17 is 0001 and the binary representation of 17 is 10001. The depth of the node number 17, defined as the length of path from the root to that node, is 4 and may be computed as one less than the length of the binary representation of 17, that is, $\lfloor \log_2 17 \rfloor$. Since it is the last node of the heap, it is also the depth of the heap. (We will comment more on this later.)

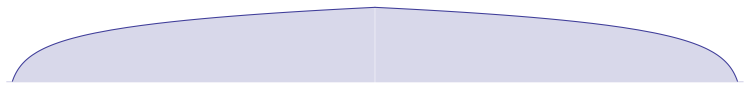


Figure 3: A really large heap on a small picture.

In addition to providing navigation information, labels of the edges indicate orientation of children: an edge labeled with 0 points to the left child and one labeled with 1 points to the right child. It so happens that the children of node i are $2i$ and $2i + 1$, as it has been visualized on Figure 4.

This important fact may be easily established by looking at binary representations of nodes' ordinal numbers. Since each such representation is a sequence of bits that determine the path from the root to the node in question, the binary representation of a parent node is a result of truncating the

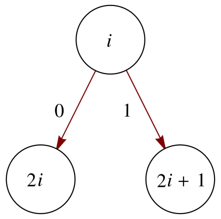


Figure 4: Ancestral information translated onto ordinal numbers (shown) of the nodes of heap.

last bit from the binary representation of any of its children. Truncating the last bit yields the same result as the `shiftright` operation, that performs the integer division by 2. So, if j is the ordinal number of a child then the ordinal number i of its parent is

$$i = \lfloor \frac{j}{2} \rfloor,$$

or, in other words,

$$j = \begin{cases} 2i \text{ if } j \text{ is the left child of } i \\ \\ 2i+1 \text{ if } j \text{ is the right child of } i. \end{cases} \tag{1}$$

For example, the path from the root to node 13 in the heap on Figure 2, is 101 which can be obtained from the binary representation of 13, that is, from 1101, by dropping its first digit 1. So the path to the parent of 13 is 10 and the ordinal number of the node at the end of that path (the parent of 13) is 110, or in 6 in decimal. So, 6 is the parent of 13. Of course, $6 = \lfloor \frac{13}{2} \rfloor$.

Also, the children of 6, if it has any, must have ordinal numbers that in binary read 1100 and 1101 since these are the only numbers that when divided by 2 will yield 110 or 6. These are 12 and 13.

In a similar fashion one can determine if a node i has a child of children by comparing $2i$ to n. If $2i \leq n$ then i has a child or children and if $2i > n$ then it has not (is a leaf, that is). If $2i = n$ then $2i + 1 > n$ and so node i does not have the right child. If $2i < n$ then $2i + 1 \leq n$ and so node i does have the right child.

Partially ordered means that every sequence of nodes along a path from the root to a leaf in the tree is ordered in a non-increasing order. Or, in other words, that children, if any, of a node are not larger than their parent.

Figure 5 visualizes an example of a heap with 10 nodes with the values of the nodes shown instead of their ordinal numbers.

One of the most amazing things about contiguous trees is how cleverly are they represented with one-dimensional arrays. An array stores the nodes of the tree according to their level-by-level order. The root of the tree is stored at index 1, and the children, if any, of a node stored at index i are stored at indices $2i$ (the left child) and $2i + 1$ (the right child). A node i has a child j (left or right) if, and only if, $j \leq N$, where N is the number of nodes of the tree.

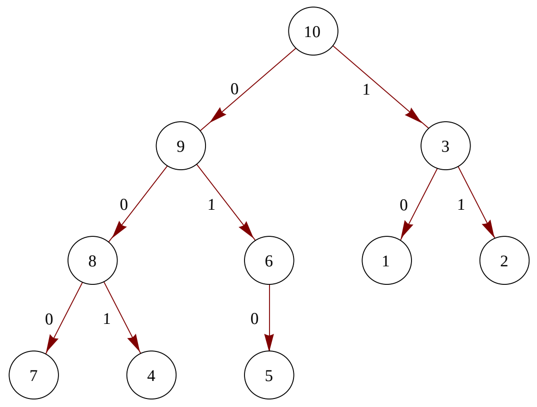


Figure 5: A heap with 10 nodes showing their values and not the ordinal numbers.

The table in Figure 6 shows an array that represents the heap of Figure 2 with the indicies of the array shown in the top row of the table.

1.3 Heapsort

HeapSort (see, e.g., [Knu97] for its description and partial analysis) consists of two phases: *heap construction* and a sequence of removals from the

1	2	3	4	5	6	7	8	9	10
10	9	3	8	6	1	2	7	4	5

Figure 6: Array representation of the heap of Figure 5 showing both the ordinal numbers (indicies, in the upper row) and the values (in the lower row) of the nodes.

constructed heap that I call *heap deconstruction*.

Both phases use a subroutine (referred to as FixHeap in this paper) that inherits an *almost heap*, defined as a heap whose root may violate the *partially ordered tree* condition, and turns it onto a heap by bubble-sorting its root into one of its paths. This is done by demoting the said root down the heap while promoting the largest of its current children until the demotee reaches the level where it is not less than any of its current children, if it still has any at that level.

The heap-construction phase (referred to as MakeHeap in this paper and credited to Floyd [Flo64]) inherits an array that represents a contiguous tree T of numbers and rearranges it onto a heap by calling FixHeap for its parts that represent subtrees of T that have been already rearranged onto almost heaps, beginning from the one that has the last non-leaf (stored at the index $\lfloor \frac{N}{2} \rfloor$ in the array) of T as the root[1] and ending with the entire tree T (the root of which is stored at index 1). This is accomplished by the following

[1]MakeHeap could have begun calling FixHeap from the last node of T, but this would

Java statement:

$$\text{for (int i} = \text{N/2; i} > \text{0; i--) FixHeap(i);} \qquad (2)$$

The heap-deconstruction phase (referred to as RemoveAll in this paper and credited to Williams [Wil64]) consists of N calls to a subroutine (referred to as RemoveMax in this paper) that removes the root of the heap, fills the resulting vacancy with the last node of the heap, and calls FixHeap in order to turn the resulting almost heap onto a heap after each removal. The removed nodes are then stored in the array from the last index up in the order they were removed, which process yields an array that is sorted in an increasing order.

A complete code of HeapSort may be easily found in about every standard text on Data Structures and Algorithms, or in [WWW].

1.4 The depth of a heap

Each node of of a heap with n nodes is represented by a binary sequence (a path from the root of the heap to that particular node). The depth of the heap is equal to the maximal (over all nodes of the heap) length of such a path. Since the last node in the heap belongs to the last level of the heap, the length of the path from the root to that last node is maximal.

produce the same sequence of comparisons of keys and demotions because FixHeap does not do anything to a one-node tree.

Let p be the path from the root to the last node of the heap. As we noticed before, the binary representation of that node's ordinal number (n, that is) is 1 followed by p.

In other words, the depth D_n of the heap with n nodes, which is equal to the length of p, is one less than the number of bits needed to represent n. So

$$D_n = \lfloor \log_2 n \rfloor.$$

Note. The above equality was also proved in file `2 − trees.pdf`, Section *Properties of balanced trees*, pages 26 and on.

Exercise Show that a heap with l leaves (<u>not</u> nodes) has a depth D that satisfies

$$\lceil \log_2 l \rceil \leq D \leq \lfloor \log_2 l \rfloor + 1.$$

Since the depth of a heap with n nodes is also the level of node n, it follows that the level of node i is:

$$level(i) = \lfloor \log_2 i \rfloor.$$

To see why, remove from the heap all the nodes after i. The resulting heap will have i nodes so its height is $\lfloor \log_2 i \rfloor$, which (by the definition of the height of a heap) happens to be the same as the level of i.

2 The worst-case running time of FixHeap

A worst-case scenario for FixHeap forces it to demote the current root i of the subtree (an almost heap) all the way down to become the leftmost leaf in the last level of that subtree or the parent of that leaf. In such a case, the number of comparisons of keys that FixHeap must perform is maximal. Estimation of the depth of the subtree in question proves useful in determination exactly what that number is.

Lemma 2.1 *The depth D_n^i of a subtree H_n^i rooted at node i of a contiguous binary tree T with n elements satisfies this inequality:*

$$D_n^i \leq \lfloor \lg n \rfloor - \lfloor \lg i \rfloor. \tag{3}$$

Proof. The depth D_n^i of subtree H_n^i is either equal to the difference

$$level(n) - level(i)$$

in the case the subtree H_n^i contains some nodes of the last level of tree T, or to

$$level(n) - level(i) - 1$$

otherwise. Since $level(n) = \lfloor \log_2 n \rfloor$ and $level(i) = \lfloor \log_2 i \rfloor$, the inequality (3) holds in either case. □

Lemma 2.1 allows for computing the worst-case number of comparisons of keys performed by FixHeap at node i. It is equal to the depth of the subtree rooted at i plus the depth of the subtree at i with the last node n removed from the entire tree.

Corollary 2.2 *The worst-case number $C_{\texttt{FixHeap}(i)}(n)$ of comparisons of keys that the* FixHeap *must perform in order to turn an* almost heap *subtree H_n^i rooted at node $i \leq \lfloor \frac{n}{2} \rfloor$ of a contiguous binary tree T with $n \geq 2$ elements onto a heap satisfies this inequality:*

$$C_{\texttt{FixHeap}(i)}(n) \leq 2(\lfloor \lg n \rfloor - \lfloor \lg i \rfloor). \tag{4}$$

Proof. In the worst-case, FixHeap will demote the root i of a given *almost heap* subtree H_n^i all the way to the H_n^i's last level, performing at most two comparisons (one to find the larger of the two children as a candidate for promotion, and one to compare it to the node that is being demoted) per level, except for the level 0. This will result in no more than twice the depth D_n^i of subtree H_n^i comparisons, that is no more than $2D_n^i = 2(\lfloor \lg n \rfloor - \lfloor \lg i \rfloor)$ comparisons. □

Because FixHeap(i) is called only if $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$; in particular, $n \geq 2$ is required. So the constrains in Corollary 2.2 do not obstruct its applications to further analysis of MakeHeap.

3 The worst-case running time of MakeHeap

Theorem 3.1 The worst-case number $C_{\texttt{MakeHeap}}(n)$ of comparisons of keys that the MakeHeap must perform in order to turn an array with $n \geq 2$ elements onto a heap satisfies this inequality:

$$C_{\texttt{MakeHeap}}(n) \leq 3n - 2 \lg n + 2. \tag{5}$$

Proof. Recall from file `LowerBoundAverageCaseSorting.nb` that the minimum *external path length* in a binary tree with m external nodes is given by this formula:

$$epl_{min}(m) = m(\lg m + \varepsilon),$$

where

$$\varepsilon = 1 + \theta - 2^{\theta} \text{ and } \theta = \lceil \lg\, m \rceil - \lg\, m,$$

Putting $m = n + 1$, where n is the number of internal nodes, we get

$$ipl_{min}(n) = epl_{min}(m) - 2n = (n+1)(\lg(n+1) + \varepsilon) - 2n =$$

$$= (n+1)(\lg(n+1) + \varepsilon) - 2(n+1) + 2 =$$

$$= (n+1)(\lg(n+1) + \varepsilon - 2) + 2 =$$

or

$$ipl_{min}(n) = (n+1)(\lg \frac{n+1}{4} + \varepsilon) + 2. \tag{6}$$

Since $ipl_{min}(n) = \sum_{i=1}^{n} \lfloor \lg i \rfloor$, from (6) we conclude:

$$\sum_{i=1}^{n} \lfloor \lg i \rfloor = (n+1)(\lg \frac{n+1}{4} + \varepsilon) + 2, \tag{7}$$

or, by $0 \leq \varepsilon$,

$$\sum_{i=1}^{n} \lfloor \lg i \rfloor \geq (n+1) \lg \frac{n+1}{4} + 2. \tag{8}$$

Now, the proof.

The Java statement (2) of page 10 should make it clear that

$$C_{\texttt{MakeHeap}}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} C_{\texttt{FixHeap}(i)}(n). \tag{9}$$

Application of equality (4) of Corollary 2.2 to (9) yields

$$C_{\texttt{MakeHeap}}(n) \leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 2(\lfloor \lg n \rfloor - \lfloor \lg i \rfloor) = 2 \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (\lfloor \lg n \rfloor - \lfloor \lg i \rfloor) =$$

$$= 2(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lfloor \lg n \rfloor - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lfloor \lg i \rfloor) = 2(\lfloor \frac{n}{2} \rfloor (\lfloor \lg n \rfloor - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lfloor \lg i \rfloor) \leq$$

[by the inequality (8), substituting $\lfloor \frac{n}{2} \rfloor$ for n]

$$= 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - (\lfloor \frac{n}{2} \rfloor + 1) \lg \frac{\lfloor \frac{n}{2} \rfloor + 1}{4} - 2) =$$

$$= 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - (\lfloor \frac{n}{2} \rfloor + 1)(\lg(\lfloor \frac{n}{2} \rfloor + 1) - \lg 4) - 2) \leq$$

[since $\lfloor \frac{n}{2} \rfloor + 1 \geq \frac{n}{2}$]

$$\leq 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - (\lfloor \frac{n}{2} \rfloor + 1)(\lg \frac{n}{2} - \lg 4) - 2) =$$

$$= 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - (\lfloor \frac{n}{2} \rfloor + 1)(\lg n - \lg 2 - 2) - 2) =$$

$$= 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - (\lfloor \frac{n}{2} \rfloor + 1)(\lg n - 3) - 2) =$$

$$= 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - \lfloor \frac{n}{2} \rfloor \lg n + 3\lfloor \frac{n}{2} \rfloor - \lg n + 3 - 2) \leq$$

$$\leq 2(\lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor - \lfloor \frac{n}{2} \rfloor \lfloor \lg n \rfloor + 3\lfloor \frac{n}{2} \rfloor - \lg n + 1) = 2(3\lfloor \frac{n}{2} \rfloor - \lg n + 1) =$$

$$= 3 \times 2\lfloor \frac{n}{2} \rfloor - 2\lg n + 2 \leq$$

[since $2\lfloor \frac{n}{2} \rfloor \leq n$]

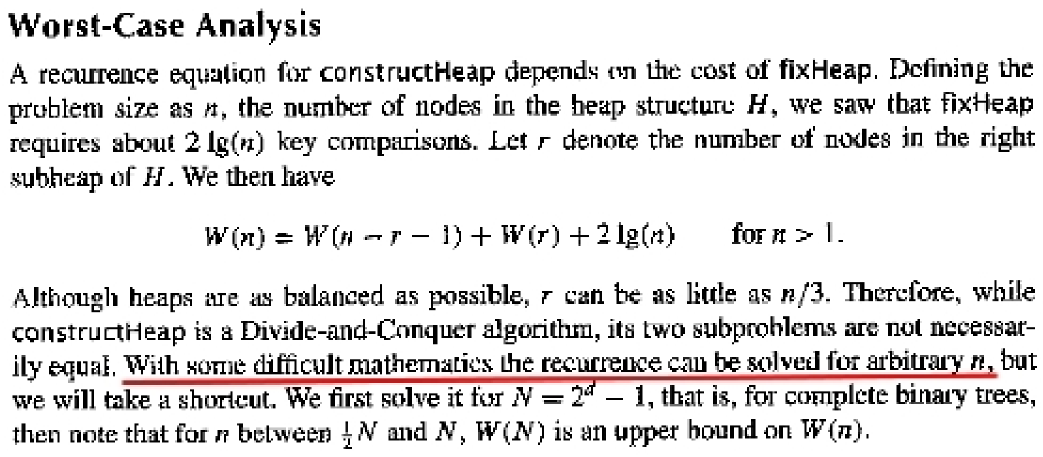
$$\leq 3n - 2\lg n + 2.$$

This completes the proof. □

Therefore,

$$C_{\texttt{MakeHeap}}(n) \in \Theta(n) \tag{10}$$

Note. The derivation of a weaker result (10) in the textbook [BG00], page 188, shown on the Figure 7, is incomplete and rather complicated (it uses the Master Theorem).



Worst-Case Analysis

A recurrence equation for constructHeap depends on the cost of fixHeap. Defining the problem size as n, the number of nodes in the heap structure H, we saw that fixHeap requires about $2\lg(n)$ key comparisons. Let r denote the number of nodes in the right subheap of H. We then have

$$W(n) = W(n - r - 1) + W(r) + 2\lg(n) \qquad \text{for } n > 1.$$

Although heaps are as balanced as possible, r can be as little as $n/3$. Therefore, while constructHeap is a Divide-and-Conquer algorithm, its two subproblems are not necessarily equal. With some difficult mathematics the recurrence can be solved for arbitrary n, but we will take a shortcut. We first solve it for $N = 2^d - 1$, that is, for complete binary trees, then note that for n between $\frac{1}{2}N$ and N, $W(N)$ is an upper bound on $W(n)$.

Figure 7: A derivation of (10) in the textbook [BG00], page 188.

Note. The exact worst-case number of comparisons of `MakeHeap` is given by this formula (published in [Suc12]):

$$C_{\texttt{MakeHeap}}(n) = 2n - 2s_2(n) - e_2(n), \qquad (11)$$

where $s_2(n)$ is the sum of all digits of the binary representation of n and $e_2(n)$ is the exponent of 2 in the prime factorization of n.

It satisfies this inequality:

$$2n - 2\lg(n+1) \le C_{\texttt{MakeHeap}}(n) \le 2n - 4 \text{ for } N \ge 3. \tag{12}$$

Below is a graph of $2n - 2s_2(n) - e_2(n)$ plotted against an upper bound $3n - 2\lg n + 2$.

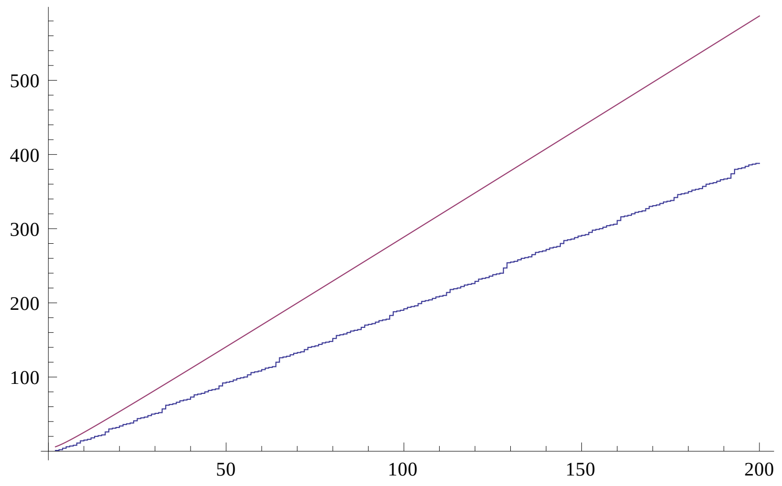


Figure 8: A graph of $2n - 2s_2(n) - e_2(n)$ plotted below an upper bound $3n - 2\lg n + 2$.

4 The worst-case running times of H.RemoveMax() and H.RemoveAll()

Let H be a heap with n nodes. Because the only comparisons of keys are made by FixHeap(1), the number of comparisons $C_{RemoveMax}(n)$ done in the worst case by H.RemoveMax() is equal to 0 if $n \leq 2$ or, by Corollary 2.2, less than or equal to $2(\lfloor \lg(n-1) \rfloor - \lfloor \lg 1 \rfloor) = 2\lfloor \lg(n-1) \rfloor$ if $n > 2$.

Therefore,

$$C_{RemoveMax}(n) \in \Theta(\log n).$$

Exercise. Prove it!

Now, H.RemoveAll() is composed of a stream of consecutive H.RemoveMax() on a shrinking heap H that initially has n nodes and loses 1 node each time H.RemoveMax() is executed. So, the number $C_{RemoveAll}(n)$ of comparisons of keys during H.RemoveAll() is equal to the sum

$$C_{RemoveAll}(n) = \sum_{k=3}^{n} C_{RemoveMax}(k),$$

that is, it satisfies this inequality:

$$C_{RemoveAll}(n) \leq \sum_{k=3}^{n} 2\lfloor \lg(k-1) \rfloor = 2\sum_{k=3}^{n} \lfloor \lg(k-1) \rfloor = 2\sum_{k=2}^{n-1} \lfloor \lg k \rfloor =$$

[by the formula (7), substitute $n-1$ for n]

$$= 2(n(\lg \frac{n}{4} + \varepsilon) + 2) \leq$$

[by $\varepsilon < 0.08607133205593421$]

$$\leq 2(n(\lg \frac{n}{4} + 0.08607133205593421) + 2) =$$
$$= 2(n(\lg n - \lg 4 + 0.08607133205593421) + 2) =$$
$$= 2(n(\lg n - 1.91392866794406579) + 2) =$$
$$= 2n \lg n - 3.82785733588813158n + 4.$$

Hence,

$$C_{RemoveAll}(n) \leq 2n \lg n - 3.82785733588813158n + 4.$$

This way we proved the following theorem whose thesis is illustrated on Fig. 9.

Theorem 4.1 The worst-case number $C_{RemoveAll}(n)$ of comparisons of keys that the removeAll must perform in order to empty a heap with n nodes satisfies this inequality:

$$C_{RemoveAll}(n) \leq 2n \lg n - 3.82785733588813158n + 4. \qquad (13)$$

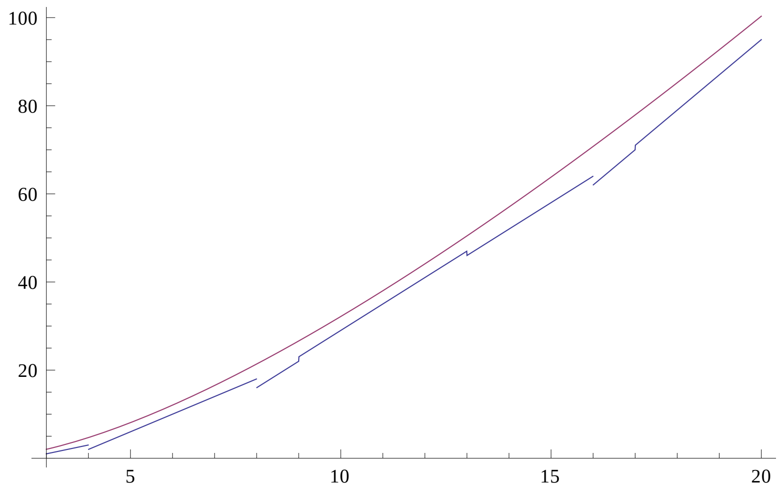


Figure 9: A graph of $C_{RemoveAll}(n)$ plotted below its upper bound $2n \lg n - 3.82785733588813158n + 4$.

Therefore,

$$C_{RemoveAll}(n) \in \Theta(n \lg n).$$

5 The running time of HeapSort

Combining inequalities (5) and (13) of theorems 3.1 and 4.1, we conclude:

Theorem 5.1 The worst-case number $C_{HeapSort}(n)$ of comparisons of keys that the Heapsort must perform in order to sort an n-element array satisfies this inequality for $n \geq 2$:

$$C_{HeapSort}(n) \leq 2n \lg n. \tag{14}$$

Proof. For $n = 1$, $C_{HeapSort}(n) = 0 = 2 \times 1 \times 0 = 2 \times 1 \times \lg 1 = 2n \lg n$.

For $n = 2$, $C_{HeapSort}(n) = 1 \leq 4 = 2 \times 2 \times 1 = 2 \times 2 \times \lg 2 = 2n \lg n$.

For $n = 3$, $C_{HeapSort}(n) = 3 \leq 2 \times 3 \times 1.5849625007211563 = 2 \times 3 \times \lg 3 = 2n \lg n$.

For $n > 3$,

$$C_{HeapSort}(n) = C_{\texttt{MakeHeap}}(n) + C_{RemoveAll}(n) \leq$$

$$\leq 3n - 2 \lg n + 2 + 2n \lg n - 3.8278573358881315 8n + 4 \leq$$

$$\leq 2n \lg n - 0.827857335888131 58n - 2 \lg n + 6 \leq$$

[since for $n > 3$, $-0.827857335888131 58n - 2 \lg n + 6 < 0$]

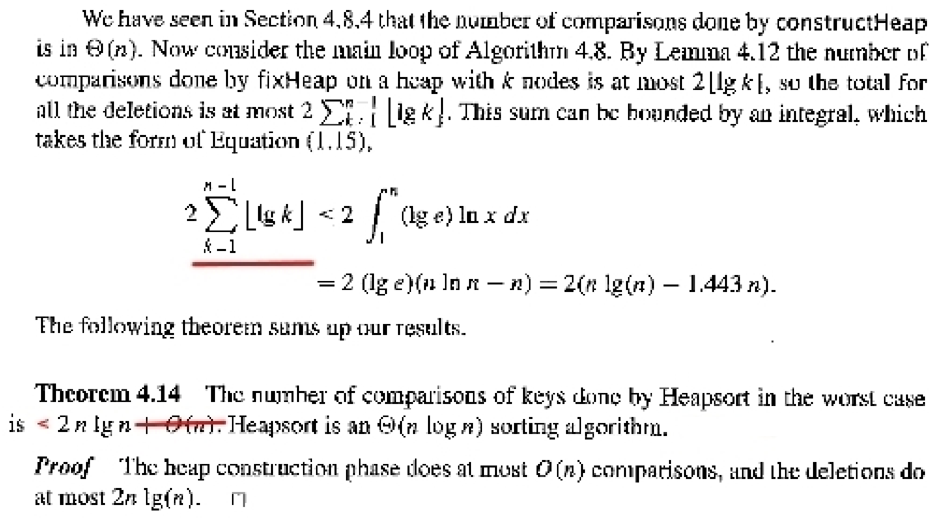
$$\leq 2n \lg n.$$

This completes the proof. $\square$

Therefore,

$$C_{HeapSort}(n) \in \Theta(n \lg n). \tag{15}$$

Note. The derivation of a weaker result (15) in the textbook [BG00], page 191, shown on the Figure 10, is incomplete and rather complicated (it implicitly uses the Master Theorem, quoted on p. 188). Moreover, there is an error in that derivation. $\int_1^n \ln x dx = n \ln n - n + 1$ and not $n \ln n - n$. As a result, the correct right-hand side of the inequality derived that way is $2(n \lg -1.443(n-1))$.



We have seen in Section 4.8.4 that the number of comparisons done by constructHeap is in $\Theta(n)$. Now consider the main loop of Algorithm 4.8. By Lemma 4.12 the number of comparisons done by fixHeap on a heap with k nodes is at most $2\lfloor \lg k \rfloor$, so the total for all the deletions is at most $2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor$. This sum can be bounded by an integral, which takes the form of Equation (1.15),

$$\underline{2\sum_{k=1}^{n-1} \lfloor \lg k \rfloor} < 2\int_1^n (\lg e) \ln x \, dx$$
$$= 2\,(\lg e)(n \ln n - n) = 2(n \lg(n) - 1.443\, n).$$

The following theorem sums up our results.

Theorem 4.14 The number of comparisons of keys done by Heapsort in the worst case is $< 2n \lg n$ ~~$+ O(n)$.~~ Heapsort is an $\Theta(n \log n)$ sorting algorithm.

Proof The heap construction phase does at most $O(n)$ comparisons, and the deletions do at most $2n \lg(n)$. $\square$

Figure 10: A derivation of (15) in the textbook [BG00], page 191.

Note. The exact formula for the worst-case number of comparisons of `HeapSort` is rather complicated and has not been known until recently; it will be published in a forthcoming paper [Suc]. Fig. 11 shows its graph plotted below the upper bound given by $2n \lg n$.

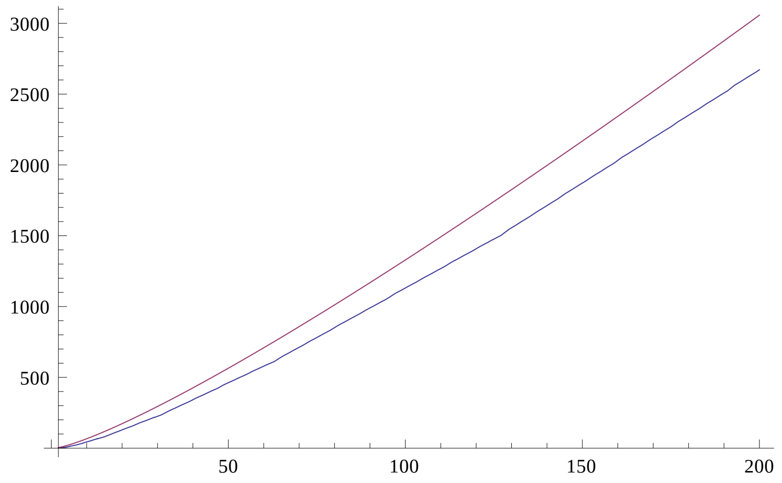


Figure 11: The worst-case number of comparisons of `HeapSort` (the lower curve) plotted below $2n \lg n$ (the upper curve).

The value given by the exact formula can be approximated by its smoother upper bound

$$2(n-1)(\lg(n-1) - \lg \frac{e}{\lg e}) \approx 2(n-1)(\lg(n-1) - 0.91392866794406579)$$

as it is shown on the Fig. 12.

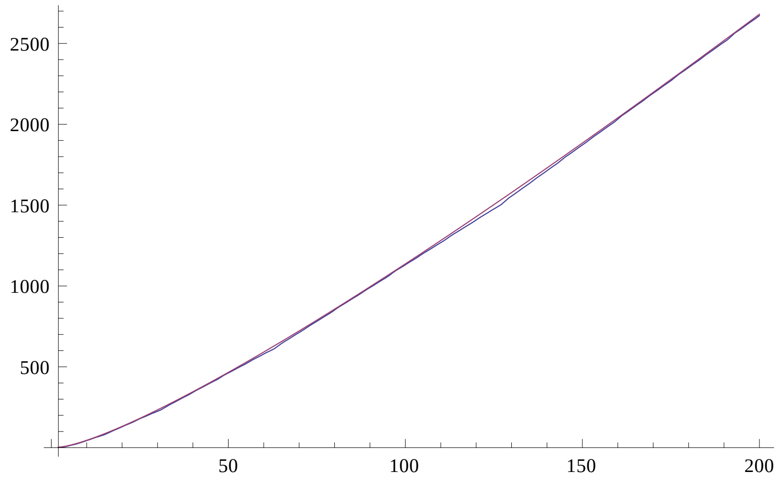


Figure 12: The worst-case number of comparisons of `HeapSort` (the lower curve) plotted below $2(n-1)(\lg(n-1) - \lg \frac{e}{\lg e})$ (the upper curve).

A smoother lower bound of the exact value is given by this formula:

$$2(n-1)(\lg(n-1)-1)-2\lg(n+1)+6.$$

Fig. 13 provides a close-up view on the exact value plotted between both bounds.

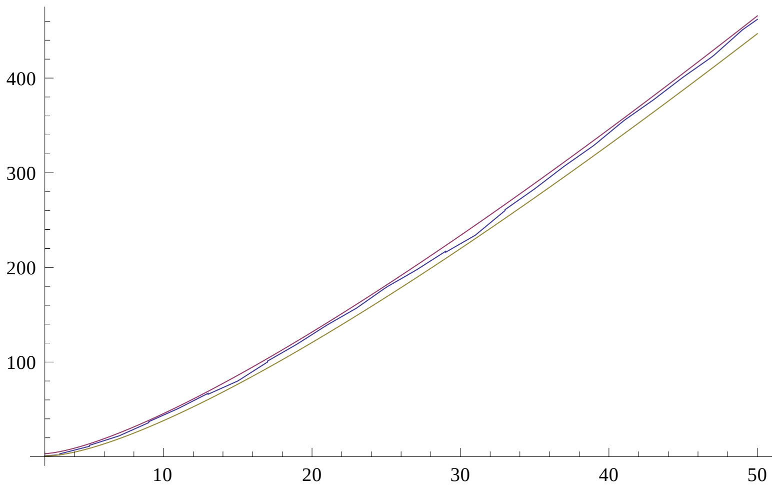


Figure 13: The worst-case number of comparisons of `HeapSort` plotted between its upper $2(n-1)(\lg(n-1)-\lg\frac{e}{\lg e})$ and lower $2(n-1)(\lg(n-1)-1)-2\lg(n+1)+6$ bounds.

References

[BG00] Sara Baase and Allen Van Gelder. *Computer Algorithms; Introduction to Design & Analysis*. Asddison Wesley, 3rd edition, 2000.

[Flo64] Robert W. Floyd. Algorithm 245: Treesort 3. *Communications of the A.C.M.*, 7(12):701, 1964.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing, 2nd edition, 1997.

[Suc] Marek A. Suchenek. The worst-case analysis of heapsort. *A forthcoming article.*

[Suc12] Marek A. Suchenek. Elementary yet precise worst-case analysis of floyd's heap-construction program. *Fundam. Inform.*, 120(1):75–92, 2012. doi 10.3233/FI-2012-751.

[Wil64] John W. J. Williams. Algorithm 232: Heapsort. *Communications of the A.C.M.*, 7(6):347–348, 1964.

[WWW] WWW. Heapsort in Java: http://csc.csudh.edu/suchenek/MakeHeap.html.