A Tight Lower Bound for the Worst Case of Bottom-Up-Heapsort

Rudolf Fleischer

MPI–I–91–104 April 1991

A Tight Lower Bound for the Worst Case of Bottom-Up-Heapsort

Rudolf Fleischer

MPI–I–91–104 April 1991

A Tight Lower Bound for the Worst Case of Bottom-Up-Heapsort [1]

by

Rudolf Fleischer [2]

ABSTRACT Bottom-Up-Heapsort is a variant of Heapsort. Its worst case complexity for the number of comparisons is known to be bounded from above by $\frac{3}{2}n\log n + O(n)$, where n is the number of elements to be sorted. There is also an example of a heap which needs $\frac{5}{4}n\log n - O(n\log\log n)$ comparisons. We show in this paper that the upper bound is asymptotical tight, i.e. we prove for large n the existence of heaps which need at least $c_n \cdot (\frac{3}{2}n\log n - O(n\log\log n))$ comparisons where $c_n = 1 - \frac{1}{\log^2 n}$ converges to 1. This result also proves the old conjecture that the best case for classical Heapsort needs only asymptotical $n\log n + O(n\log\log n)$ comparisons.

1. Introduction

Bottom-Up-Heapsort is a variant of the classical Heapsort algorithm and was presented in 1989 by Ingo Wegener ([W90]). The input to the algorithms is an array $a[1..n]$ of n elements from an ordered set S which are to be sorted. We will measure the complexity of the algorithms in terms of number of comparisons; first, because comparisons are usually the most expensive operations, and second, because each comparison is preceded by only a (small) constant number of other calculations.

First the elements will be arranged in form of a heap (Heap Creation Phase) with biggest element at the root (see Section 2 for details). This requires $O(n)$ time ([Wi64]). Then follows the Selection Phase which consists of n Rearrangement Steps. In each Rearrangement Step the root element changes place with the last active element in the array and becomes inactive; then the heap is rearranged with respect to the remaining active elements. So the size of the heap decreases by one. Since the root always contains the biggest heap element, the array will be step by step filled from the end with elements in decreasing order.

The classical rearrangement procedure works as follows ([M84]). At the beginning, the root contains a former leaf element (the last active array element is always a leaf). This element is repeatedly swapped with the bigger one of its children until it is bigger than both of its children or it is a leaf. At each level two comparisons are made. Hence the total complexity of the Selection Phase might be as big as $2n\log n$.

In Bottom-Up-Heapsort, the rearrangement procedure is changed in the following way. We first compute the **special path** ([W90]) which is the path on which the leaf element

[1] This work was supported by the ESPRIT II program of the EC under contract No. 3075 (project ALCOM)

[2] Department of Computer Science, University of Saarland, D-6600 Saarbrücken, Germany

would sink in the classical rearrangement procedure. This is the unique path with the property that any node on it (except the root) is bigger than its sibling. This costs only one comparison per level. Then we let our leaf element climb the special path up to its destination node at the cost of one comparison per level.

This algorithm tries to make use of the intuitive idea that leaf elements are likely to sink down again nearly to the bottom of the heap so one can expect that climbing up might be cheaper than sinking down. In fact, Wegener ([W90]) showed an upper bound of $\frac{3}{2}n\log n + O(n)$ for Bottom-Up-Heapsort. He also conjectured a tighter upper bound of $n\log n + o(n\log n)$ but this was disproved by [FSU] who constructed a heap with an asymptotic lower bound of $\frac{5}{4}n\log n - O(n\log\log n)$. In this paper we give a construction of a heap that improves the lower bound to asymptotical $\frac{3}{2}n\log n - O(n\log\log n)$ which matches the upper bound. This bound also implies an asymptotic upper bound of $n\log n + O(n\log\log n)$ for the best case of the classical Heapsort algorithm as has been conjectured for many years.

The construction is an improvement of our previous work ([FSU]). In that paper we constructed a heap with the property that the first $\frac{n}{2}$ rearrangement steps of Bottom-Up-Heapsort need nearly $\frac{3}{4}n\log n$ comparisons. At this point, the active heap is the initial heap without its leaf-level which now contains the $\frac{n}{2}$ biggest elements in sorted order. Unfortunately, we could say nothing nontrivial about the remaining rearrangement steps; hence we only got a lower bound of $\frac{5}{4}n\log n$.

In this paper we will use the same general ideas but many details are quite different. Also the proof techniques have changed completely. The advantage is that we now can iterate the above procedure, i.e. we can prove that not only the leaf-level but many levels of the heap are expensive. This gives the asymptotic optimal lower bound.

The paper is organized as follows. In Section 2 we present the classical Heapsort algorithm and the Bottom-Up-Heapsort algorithm in detail. In Section 3 we give some definitions and prove some simple properties of heaps. In Section 4 we explain the heap construction followed by the complexity analysis in Section 5. We conclude with some remarks in Section 6.

2. Heapsort and Bottom-Up-Heapsort

In this section we give the programs of the two Heapsort-versions in detail. We follow the notaions of [W90] and [FSU].

The input to the algorithms is an array $a[1..n]$ with elements of an ordered set S. The **heap property** at position i is fulfilled if ($a[i] \leq a[2i]$ or $i > \lfloor\frac{n}{2}\rfloor$) and ($a[i] \leq a[2i+1]$ or $i \geq \lceil\frac{n}{2}\rceil$). The array is called a **heap** if the heap property is fulfilled for all positions. Thus the array is considered as a binary tree, where the children of node i are the nodes $2i$ (if $2i \leq n$) and $2i+1$ (if $2i+1 \leq n$). We now give the classical Heapsort algorithm.

```
Heapsort(n)
(* sort an array of size n *)
```

```
(1) for $i = \lfloor \frac{n}{2} \rfloor, \ldots, 1$ do rearrange($n, i$);      (* Heap Creation Phase *)
(2) for $m = n, \ldots, 2$ do      (* Selection Phase *)
(3)   interchange $a[1]$ and $a[m]$;
(4)   if $m > 2$ then rearrange($m - 1, 1$);
(5) od
```

Procedure rearrange(m, i)
(* check the i-th element of a heap of size m *)

```
(1) if $i < \frac{m}{2}$ then $max := \max(a[i], a[2i], a[2i+1])$;
    if $i = \frac{m}{2}$ then $max := \max(a[i], a[2i])$;
    if $i > \frac{m}{2}$ then STOP;
(2) if $max = a[i]$ then STOP;
(3) if $max = a[2i]$ then
      interchange $a[i]$ and $a[2i]$;
      rearrange($m, 2i$);
(4) else
      interchange $a[i]$ and $a[2i+1]$;
      rearrange($m, 2i + 1$);
    fi
```

Bottom-Up-Heapsort works like Heapsort, but *rearrange* is replaced by the procedure *bottom-up-rearrange*.

Procedure bottom-up-rearrange(m, i)
(* check the i-th element of a heap of size m *)

```
(1) leaf-search($m, i, j$);
(2) bottom-up-search($i, j$);
(3) interchange($i, j$);
```

We first search for the leaf that we can reach by starting at node i and going always down to the child containing the bigger element. We call this leaf the **special leaf** and the corresponding path **special path**. This is done by the procedure *leaf-search*.

Procedure leaf-search(m, i, j)
(* search the special path starting at node i in a heap of size m; the leaf will be returned in j *)

```
(1) $j := i$;
(2) while $2j < m$ do
      if $a[2j] > a[2j+1]$ then $j := 2j$
      else $j := 2j + 1$;
    od;
(3) if $2j = m$ then $j := m$;
```

We now climb up the special path and look for the destination position of element i which is the same position as computed by the *rearrange* procedure above. This is done by the procedure *bottom-up-search*.

```
Procedure bottom-up-search($i, j$)
(* let the $i$-th element climb up the tree starting at node $j$ *)
(1) while $a[i] < a[j]$ do $j := \lfloor \frac{j}{2} \rfloor$;
```

Now we have to shift up the elements of the ancestors of the computed position on the special path. This is done by the procedure *interchange*.

```
Procedure interchange($i, j$)
(* place the $i$-th element in node $j$ and shift up all elements above $j$ *)
(1) $x := a[j]$;
(2) $a[j] := a[i]$;
(3) while $j > i$ do
      interchange $a[\lfloor \frac{j}{2} \rfloor]$ and $x$;
      $j := \lfloor \frac{j}{2} \rfloor$;
    od
```

3. Basic Properties

In this section we will give some definitions and prove some basic lemmas about heaps. W.l.o.g. we only construct heaps of size $n = 2^m - 1$, i.e. the heap is a complete binary tree of height m. After the first 2^{m-1} steps of Bottom-Up-Heapsort the leaves of the initial tree are deleted (and filled with the 2^{m-1} biggest elements) and the remaining active heap is a complete binary tree of height $m-1$. We call this deletion of the leaf-level an **iteration**. Bottom-Up-Heapsort consists of $m-1$ iterations.

To achieve high complexity in an iteration, many leaf elements must climb up the tree to high destination nodes. The upper bound proof ([W90],[FSU]) shows that only about one half of the leaf elements can have this property at all. We will construct a heap which will for many consecutive iterations send nearly half of the leaf elements to high destinations; thus its complexity will be very close to the upper bound. Our construction is based on two types of heaps defined below; both types have the above property for the first iteration, and after the first iteration the resulting heap will be of the other type (at least if it is still big enough).

First we need some notations for some particular parts of a heap (see Fig. 3.1). Let k be a fixed constant to be defined later and H_m be a complete heap of height m with $k + \log k + 1 \leq m \leq 2k - 4$.

- The root is called r.
- The first k levels of H_m are a complete binary tree called B (it will always contain big elements).
- A_1 is a binary tree rooted at $leftson(r)$ of height $m-k$ ($\leq k-4$) with the $k-2$ rightmost nodes of the lowest level missing, i.e. A_1 contains only $a := 2^{m-k} - k + 1 (\geq k+1)$ nodes. A_2 is defined symmetrical. (Remark: For the proof of correctness, any other sets A_1 and A_2 in the two halves of B would work as well; but this definition yields optimal results in the complexity analysis).

- The leaves of B are from left to right $v_1, v_2, \ldots, v_{2l}$ with $l := 2^{k-2}$.
- The left subtree of v_i is called D_i, the right subtree is called E_i. D_i and E_i both have height $m-k$ and size $2^{m-k}-1 \geq 2k-1$.

We distinguish between **Type-I** and **Type-II Heaps** according to the following conditions (of course, there are also heaps which are of neither type but they do not occur in our construction). If no confusion is possible we also write w for the element stored in a node w and F for the set of elements stored in a heap F.

Def. 3.1 Let F be some heap.

(1) F^{left} is the leftmost leaf of F.

(2) Let G be a heap of smaller size than F. F is a **predecessor** of G if Bottom-Up-Heapsort started with input heap F will end up with heap G after some rearrangement steps.

(3) If F and G are complete heaps of same size, we say G **is below** F ($G \prec F$) iff there exists a heap H' as shown in Fig. 3.2 which is a predecessor of heap H (also Fig. 3.2). (Remark: $G < F \Rightarrow G \prec F$ but not vice versa !).

(4) If (I.1)–(I.6) are satisfied then F is a

Type-I Heap

(I.1) $D_i \prec E_{i-1}$, $2 \leq i \leq 2l$

(I.2) $E_2 < A_2$

(I.3) $\exists e \in D_1 : e < E_1^{\text{left}}$

(I.4) $\exists f \in D_1 \backslash \{e\} : f < E_2^{\text{left}}$

(I.5) $father^{(2)}(v_1) := father(father(v_1)) < E_{2l}$

(I.6) $v_i < E_{i+1}^{\text{left}}$, $2 \leq i \leq 2l-2$

(5) If (II.1)–(II.3) are satisfied then F is a

Type-II Heap

(II.1) $E_i \prec D_i$, $1 \leq i \leq 2l$

(II.2) $E_i < B$, $1 \leq i \leq 2l$

(II.3) $v_2 < A_2$

First we show that these definitions make sense, i.e. we show

Lemma 3.2 For all m, $k + \log k + 1 \leq m \leq 2k-4$, exists a type-I (type-II) heap of height m.

Proof: Place the smallest elements in D_i, $1 \leq i \leq 2l$, and then fill the heap from left to right with elements in increasing order, i.e. choose always the first node in symorder with no empty children to fill next. Then (I.1)–(I.6) are satisfied. Type-II heaps are similar (smallest elements in E_i instead). □

Now we make the following crucial observation.

Lemma 3.3 (*Onestep-lemma*) Let F_n be a heap of n elements stored in $a[1..n]$. Let b be any element bigger than F_n and c be any element of F_n with $c \leq a[\lfloor \frac{n+1}{2} \rfloor]$ (node $\lfloor \frac{n+1}{2} \rfloor$ is the father of node $n+1$). Then there exists a heap F_{n+1} of $n+1$ elements with b stored at the root and $a[n+1] = c$ which is a predecessor of F_n.

Proof: Let p be the path in F_n from the root to the node where c is stored. Move c to $a[n+1]$. Then move all remaining elements of p down one level along p. Finally put b into the root.

The tree obtained has the following properties: It is a heap (because it was a heap before) and p is the upper part of the special path (any element of p was replaced by its father thus becoming the bigger sibling). Hence the first rearrangement step of Bottom-Up-Heapsort will transform F_{n+1} into F_n. □

From this follow immediately two more useful lemmas.

Lemma 3.4 (*Filling-lemma*) Let F be a complete heap and F_1 be some subset of F with $F_1 < F - F_1$. Then there exists a heap which is a predecessor of F and whose lowest level consists only of the $|F_1|$ leftmost positions filled with the elements of F_1 (see Fig. 3.3).

Proof: An element of F_1 is called wrong if it is not at the final lowest level or if it has a F_1-father. Apply the *Onestep-lemma* $|F_1|$ times to the smallest wrong F_1-element. □

Lemma 3.5 (*Below-lemma*) Let F and G be complete heaps of same size. Let $F = F_1 \cup F_2$ where F_2 are some leaves of the left subtree of F, and let $G = G_1 \cup G_2$ where G_2 are some leaves of G (see Fig. 3.4). If $F_1 > G_1$, $F_2 > G_2$ and $|G_2| \geq 2|F_2|$ then $G \prec F$.

Proof: The leaves of G can all be placed as new leaves below the left subtree of F. Hence we can repeatedly apply the *Onestep-lemma* to heap H of Fig. 3.2, first moving all leaves of G in an appropriate order below F. □

4. Heap Construction

In this section we will show how to construct a heap of n elements which forces Bottom-Up-Heapsort to make many comparisons.

Theorem 4.1 *For any $m \geq 50$ we can construct a heap H_m of height m which forces Bottom-Up-Heapsort to make $\frac{n}{2} \cdot (3\log n - 8\log\log n - 2) \cdot (1 - \frac{1}{\log^2 n})$ comparisons, where $n = 2^m - 1$.*

Proof: Choose $k := m - \lfloor 4\log m \rfloor$ and let $H_{m-\lfloor 2\log m \rfloor}$ be a type-II heap of height $m - \lfloor 2\log m \rfloor$ (which exists for $m \geq 50$ by Lemma 3.2). Then apply alternately Theorems 4.2 and 4.3 until a heap H_m of height m is constructed. These theorems can be applied as long as the height of the current heap is at most $2k - 5$; this is always the case if $m \geq 50$. The complexity of Bottom-Up-Heapsort started with H_m will be analyzed in Section 5. □

We remark that there is a tradeoff (within some limits) between the factor $c_n := 1 - \frac{1}{\log^2 n}$ of the whole term and the factor 8 of the $\log\log n$ term by choosing other values of k together with another number of iterations.

Theorem 4.2 *Let H_m be an arbitrary type-II heap of height m, $k + \log k + 1 \leq m \leq 2k - 5$. Then we can construct a type-I heap H_{m+1} of height $m + 1$ which is a predecessor of H_m.*

Proof: The elements of H_m are called **old** elements whereas the new added elements are called **new** elements. New elements are bigger than any old element and they are added step by step in increasing order.

Algorithm 1

The algorithm runs in $2l$ rounds; in round i, $1 \leq i \leq 2l$, new leaves are created below D_i and E_i.

(i) Apply the *Onestep-lemma* to D_i^{left}; this creates the leftmost new leaf below D_i.

(ii) Move E_i below D_i according to (II.1). This fills E_i with new elements and $k - 1$ old elements for $i = 1$ (from the path to v_1; the root got a new element in (i)) and at most $k - 2$ old elements for $i \geq 2$ (the root contains a new element at the beginning of round i).

(iii) If $i = 1$ then apply the *Filling-lemma* to E_1 and its $k - 1$ smallest elements. If $i \geq 2$ then apply the *Filling-lemma* to E_i and its $k - 2$ smallest elements; then apply the *Onestep-lemma* to the leftmost new leaf (which contains one of these elements). Now E_i and its leftmost new leaf contain only new elements.

(iv) Hence the elements of A_2 (if $i \leq l$) or A_1 (if $i \geq l+1$) can be placed as new leaves below E_i (apply the *Onestep-lemma*).

Algorithm 1 has constructed a heap H_{m+1} of height $m+1$. It remains to show that it is a type-I heap. An upper index II denotes the old elements of H_m.

(I.1) We have to show $D_i \prec E_{i-1}$, $2 \leq i \leq 2l$.
D_i only contains the old elements v_i^{II}, D_i^{II} and E_i^{II}; v_i^{II} is the biggest of them.
E_1 contains only the $k-1$ old elements from the path to $father(v_1^{II}) = father(v_2^{II})$ and A_2^{II}; (II.3) implies $D_2 < E_1$.
If $i \geq 3$ then E_{i-1} contains at most $k-2$ old elements from the path to $father(v_{i-1}^{II})$; they are stored in the left half of E_{i-1} and they are bigger than E_i^{II} by (II.2); since $|E_i^{II}| \geq 2(k-2)$ the *Below-lemma* can be applied.

(I.2) A_2 was filled with new elements in rounds $i \geq l+1$ whereas E_2 can only contain new elements of rounds $i \leq l$. Hence $E_2 < A_2$.

(I.3) E_1^{left} is an old element from the path to $father(v_1^{II})$ which is bigger than D_1.

(I.4) E_2^{left} is a new element and hence bigger than D_1.

(I.5) E_{2l} only contains new elements which were added in rounds $i \geq l+1$ whereas $father^{(2)}(v_1^{II})$ can only contain a new element of rounds $i \leq l$ because it is below A_1.

(I.6) After step (iii) of the algorithm, E_i only contains new elements which were added during round i. Hence E_i is bigger than any element to the left of the path to E_i^{left}. □

The other construction is similar but some details are more complicated.

Theorem 4.3 *Let H_m be an arbitrary type-I heap of height m, $k + \log k + 1 \leq m \leq 2k-5$. Then we can construct a type-II heap H_{m+1} of height $m+1$ which is a predecessor of H_m.*

Proof : We first give the algorithm. As in Algorithm 1 we have old and new elements.

Algorithm 2

The algorithm runs in $2l+1$ rounds; in round i, $1 \leq i \leq 2l+1$, new leaves are created below E_{i-1} and D_i (if they exist).

(a) $i = 1$:

Analogous to the *Filling-lemma* we can move $D_1 \cup \{v_1\}$ into the new leaves below D_1 by filling the path to v_1 and the upper part of D_1 with 2^{m-k} new elements.

(b) $i = 2$:

(i) Apply the *Onestep-lemma* to the element $e \in D_1$ of (I.3) which now is in one of the new leaves below D_1.

(ii) Move D_2 below E_1 according to (I.1). This fills D_2 with new elements and one old element, v_2.

(iii) Apply the *Filling-lemma* to D_2 and its $k-1$ smallest elements. Now D_2 contains only new elements; the $k-1$ new leaves contain v_2 and new elements.

(iv) Hence all elements of A_2 can be placed as new leaves below D_2 (use the *Onestep-lemma*).

(c) $3 \leq i \leq l$:

(i) Apply the *Onestep-lemma* to the smallest element of D_{i-2}. This is possible by (I.4) for $i = 3$, by (I.6) and (b)(iii) for $i = 4$ and by (I.6) and (c)(iii) for $i \geq 5$.

(ii) Move D_i below E_{i-1} according to (I.1). This fills D_i with new elements and at most $k-1$ old elements (from the path to v_i; the root got a new element in (i)).

(iii) Apply the *Filling-lemma* to D_i and its $k-1$ smallest elements. Now D_i contains only new elements; some of the $k-1$ new leaves contain old elements, one of them is v_i.

(iv) Hence all elements of A_2 can be placed as new leaves below D_i (use the *Onestep-lemma*).

(d) $l + 1 \leq i \leq 2l$:

Same as (c) but in (iv) use A_1 instead of A_2.

(e) $i = 2l + 1$:

The new leaves of D_1 now contain some elements of $D_1 \cup \{v_1, father(v_1), father^{(2)}(v_1)\}$. By (I.5) we can apply the *Onestep-lemma* to move all these leaf-elements below E_{2l}.

Algorithm 2 has constructed a heap H_{m+1} of height $m+1$. It remains to show that it is a type-II heap. An upper index I denotes old elements of H_m.

(II.1) We have to show $E_i \prec D_i$ for all i.

<u>$i = 1:$</u> D_1 contains only $k-3$ old elements (from the path to $father^{(3)}(v_1^I)$). E_1 only contains old elements; these are smaller than the old elements of D_1 because they were below $father(v_1^I)$. Hence $E_1 < D_1$.

<u>$i = 2:$</u> D_2 contains only the old elements A_2^I. E_2 only contains the old elements E_2^I, D_3^I and a $f^I \in D_1^I$ (I.4). Since $E_2^I < A_2^I$ by (I.2) we conclude $E_2 < D_2$.

<u>$3 \le i \le 2l-1:$</u> D_i contains at most $k-1$ old elements from the path to v_i^I. E_i only contains the old elements E_i^I, D_{i+1}^I and v_{i-1}^I. Since $E_i^I < v_i^I$ we conclude $E_i < D_i$.

<u>$i = 2l:$</u> D_{2l} contains only one old element, v_{2l}^I. E_{2l} only contains the old elements E_{2l}^I and some elements smaller than $father^{(2)}(v_1^I)$. Since $E_{2l}^I < v_{2l}^I$ we conclude $E_{2l} < D_{2l}$.

(II.2) B is completely filled with new elements whereas all E_i only contain old elements. Hence $E_i < B$.

(II.3) A_2 was filled with new elements in rounds $i \ge l+1$ whereas v_2 can only contain a new element of rounds $i \le l$. Hence $v_2 < A_2$. □

5. Complexity Analysis

First we will prove that the heaps constructed in Theorems 4.2 and 4.3 have an expensive first iteration.

Lemma 5.1 Let H_{m+1} be the type-I heap constructed in Theorem 4.2. If $m \ge k + 2\log k$ then the first iteration of H_{m+1} needs at least $2^{m-1} \cdot (2m+k-2)$ comparisons.

Proof: We have to search 2^m special paths at the cost of at least $m-1$ each. Then the leaf elements will climb up the special path to their destination nodes. In each of the 2^{k-1} rounds, $a = 2^{m-k} - k + 1$ leaf elements climb up to A_1 or A_2 which costs at least k; the other leaf elements need at least one comparison each. Hence

$$\begin{aligned} T_I &\ge 2^{k-1} \cdot [a \cdot k + 2^{m-k} + (k-1)] + 2^m \cdot (m-1) \\ &= 2^{m-1} \cdot (k + 2m - 2) + 2^{k-1} \cdot (2^{m-k} + (k-1) - (k-1) \cdot k) \\ &\ge 2^{m-1} \cdot (2m + k - 2) \qquad \text{for } m \ge k + 2\log k. \end{aligned}$$

□

Lemma 5.2 Let H_{m+1} be the type-II heap constructed in Theorem 4.3. If $k \ge 4$ and $m \ge k + 2\log k + 1$ then the first iteration of H_{m+1} needs at least $2^{m-1} \cdot (2m+k-2)$ comparisons.

Proof : We have to search 2^m special paths at the cost of at least $m-1$ each. Then the leaf elements will climb up the special path to their destination nodes. In rounds i, $2 \le i \le 2l$, $a = 2^{m-k} - k + 1$ leaf elements climb up to A_1 or A_2 which costs at least k; all other leaf elements need at least one comparison each. Hence

$$\begin{aligned}
T_{II} &\ge (2^{k-1}-1)\cdot[a_2\cdot k + (k-1)] + (2^{k-1}+1)\cdot 2^{m-k} + 2^m\cdot(m-1)\\
&\ge 2^{m-1}\cdot(k+2m-2) + 2^{k-1}\cdot[(k-1)-(k-1)\cdot k]\\
&\quad -(2^{m-k}-k+1)\cdot k - (k-1) + 2^{m-1}\\
&\ge 2^{m-1}\cdot(2m+k-2) + 2^{m-1} - 2^{k-1}\cdot k^2 - 2^{m-k}\cdot k\\
&\ge 2^{m-1}\cdot(2m+k-2) \qquad \text{for } k \ge 4 \text{ and } m \ge k + 2\log k + 1.
\end{aligned}$$

□

Proof of Theorem 4.1 :

We apply Lemma 5.1 to heaps of height $m - \lfloor 2\log m\rfloor$, $m - \lfloor 2\log m\rfloor + 2, \ldots$ and Lemma 5.2 to heaps of height $m - \lfloor 2\log m\rfloor + 1$, $m - \lfloor 2\log m\rfloor + 3, \ldots$ until the construction stops with a heap of height m (with $k = m - \lfloor 4\log m\rfloor$ and $m \ge 50$ all constraints about m and k are satisfied). Hence we will have total complexity

$$\begin{aligned}
T_m &\ge (2(m-\lfloor 2\log m\rfloor) + k - 2)\cdot(2^{m-2} + 2^{m-3} + \cdots + 2^{m-\lfloor 2\log m\rfloor - 1})\\
&\ge 2^{m-1}\cdot(3m - 8\log m - 2)\cdot(1 - \tfrac{1}{m^2})\\
&= \tfrac{n}{2}\cdot(3\log n - 8\log\log n - 2)\cdot(1 - \tfrac{1}{\log^2 n})\ .
\end{aligned}$$

□

6 . Conclusions

For any given m, we showed how to construct a heap of height m which forces Bottom-Up-Heapsort to make nearly $\frac{3}{2}n\log n$ comparisons. This matches the upper bound asymptotical up to low-order terms ([W90]). Furthermore, this problem is closely related to the old problem of finding the best case for the classical Heapsort algorithm; the immediate consequence is that Heapsort needs only asymptotical $n\log n + O(n\log n)$ comparisons for our heap. Another open problem about both variants of Heapsort is their average running time. We refer to [W90] for details and [SS] for a good bound on the average running time.

Acknowledgements

We would like to thank C. Uhrig and B.P. Sinha for their previous joint work. We would also like to thank S. Meiser for providing some tools to produce nice TeX-pictures (even if they are not perfect).

References

[C87a] S. Carlsson
"A variant of HEAPSORT with almost optimal number of comparisons"
Information Processing Letters **24** (1987), 247–250

[C87b] S. Carlsson
"Average-case results on HEAPSORT"
BIT **27** (1987), 2–17

[FSU] R. Fleischer, B. Sinha, C. Uhrig
"A lower bound for the worst case of bottom-up-heapsort"
Technical Report No. A23/90, University Saarbrücken, December 1990

[MDR] C.J.H. McDiarmid, B.A. Reed
"Building heaps fast"
Journal of Algorithms **10** (1989), 352–365

[M84] K. Mehlhorn
"Data Structures and Algorithms, Vol. 1, Sorting and Searching"
Springer Verlag, Berlin, 1984

[W90] I. Wegener
"BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating on average QUICKSORT (if n is not very small)"
MFCS'90, Lecture Notes in Computer Science, 516–522, 1990

[W91] I. Wegener
"The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP-HEAPSORT is less than $n \log n + 1.1n$"
Proc. STACS 1991, 137–147

[Wi64] J.W.J. Williams
"Algorithm 232"
Communications of the ACM **7** (1964), 347–348

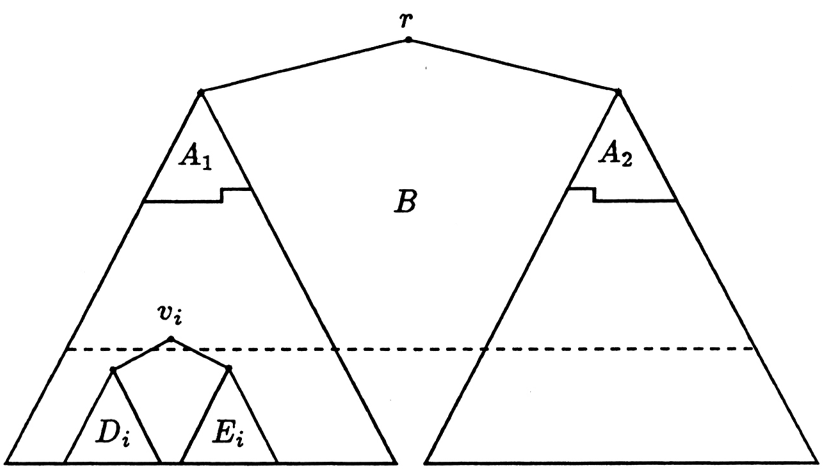

Figure 3.1

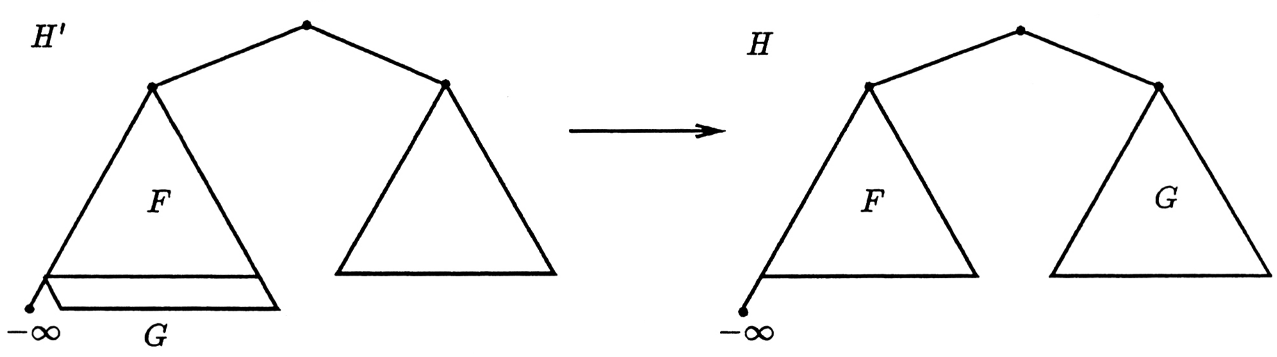

Figure 3.2

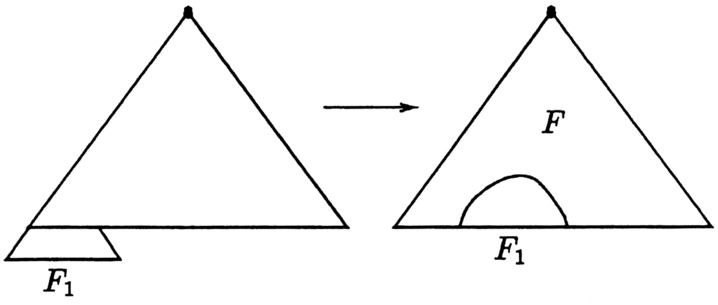

Figure 3.3

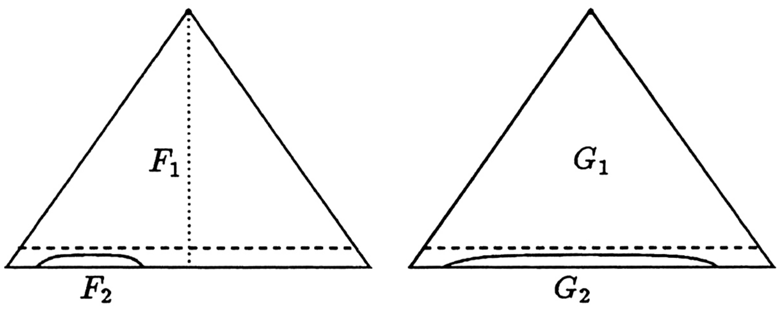

Figure 3.4