A Lower Bound for the Worst Case of Bottom–Up–Heapsort *

by

R. Fleischer, B. P. Sinha† and C. Uhrig

A 23/90

FB Informatik, Universität des Saarlandes, 6600 Saarbrücken, Federal Republic of Germany

December 1990

Abstract: Bottom–Up–Heapsort is a variant of Heapsort. Till now, its worst case complexity for the number of comparisons is known to be bounded above by $1.5n \log n + O(n)$, where n is the number of elements to be sorted; but it was conjectured to be $n \log n + O(n)$. In this paper we give a construction that proves an asymptotic lower bound of $1.25n \log n - O(n \log \log n)$ comparisons for the worst case.

---

* This research was supported by the Alexander von Humboldt–Stiftung, Germany, by the ESPRIT II Basic Research Action Program, ESPRIT P. 3075 – ALCOM, and by the DFG, grant SPP ME 620/6.

† on leave from Indian Statistical Institute, Calcutta.

I. Introduction

Bottom–Up–Heapsort is a variant of the classical Heapsort algorithm and was presented in 1989 by Ingo Wegener ([W89]). To fix our ideas, we assume that the array to be sorted has n elements which are initially arranged in the form of a heap (Heap creation phase) with the property that i) the i–th element of the array will have $(2i)$–th and $(2i+1)$–th elements (if they exist) as its left and right children respectively and ii) every element in the heap is smaller than or equal to the smaller of its two children. The heap creation phase requires $O(n)$ time (see [Wi64]). After the heap is so created, the smallest element will be at the root. Then the root element changes place with the last element in the array and the heap is rearranged with one element less, that means the last position of the heap is considered to be deleted. This is repeated n times, each time at $O(\log n)$ cost (Selection phase). The rearrangement procedure proceeds as follows. At the beginning, the root contains a former leaf element. This element is swapped with the smaller of its new children and this is repeated until it is smaller than both of its children or it is a leaf. At each level two comparisons are made. In Bottom–Up–Heapsort, the rearrangement procedure is changed in the following way. We first compute the so called *special path* ([W89]), that is the path, on which the sinking element would sink in the original rearrangement procedure. This is done by only one comparison per level. Then we let the element climb up this path to its destination position. For this variant of the Heapsort–algorithm, Wegener ([W89]) showed an upper bound of $\frac{3}{2}n\log n + O(n)$. He also conjectured a tighter upper bound of $n\log n + O(n)$. We give a construction of a heap that disproves this conjecture and gives an asymptotic lower bound of $\frac{5}{4}n\log n$.
This paper is organized as follows. In section II we present the original Heapsort algorithm and the Bottom–Up–Heapsort algorithm. In section III we give some definitions and repeat Wegeners proof for the upper bound. Section IV gives an example for our construction algorithm under the restriction that we first consider only the construction of the right subtree of the root. Section V contains the restricted construction algorithm in detail. In section VI we show how to extend the construction to the whole tree and in section VII we give the proofs and the running time analysis for Bottom–Up–Heapsort on the created heap. We conclude with some remarks in section VIII.

II. Heapsort and Bottom–Up–Heapsort

In the original version of Heapsort we consider an array $a[1..n]$ with the elements of an ordered set S. For our purpose, we assume w.l.o.g. that $S = \{1, ..., n\}$. The heap property is fulfilled if $(a[i] \leq a[2i]$ or $i > \lfloor n/2 \rfloor)$ and $(a[i] \leq a[2i+1]$ or $i \geq \lceil n/2 \rceil)$. 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 position i are the positions $2i$ (if $2i \leq n$) and $2i+1$ (if $2i+1 \leq n$).
The procedure *rearrange*(m, i) considers only the array positions $1, ..., m$ and looks at the subtree with root i. It transforms this subtree into a heap. We now give the Heapsort algorithm, following the notations of [W89].

Heapsort
1) For $i = \lfloor n/2 \rfloor, ..., 1$: rearrange (n, i). (Heap creation phase).
2) For $m = n, ..., 2$:
   interchange $a[1]$ and $a[m]$;
   if $m \neq 2$ then rearrange$(m-1, 1)$. (Selection phase).

The *rearrange* procedure looks as follows.

Procedure rearrange(m, i)
1) If $i > m/2$, STOP.
2) If $i < m/2$, compute min, the minimum of $a[i]$, $a[2i]$ and $a[2i+1]$ (2 comparisons).
   If $i = m/2$, $min = \min\{a[i], a[2i]\}$.
3)
   If $min = a[i]$ then STOP
   else if $min = a[2i]$ then interchange $a[i]$ and $a[2i]$, rearrange$(m, 2i)$
   else interchange $a[i]$ and $a[2i+1]$, rearrange$(m, 2i+1)$.

Bottom–Up–Heapsort, also denoted by BUH, works like Heapsort, but *rearrange* is replaced by the following. We first search for the leaf that we can reach by starting at the root and going always to the child containing the smaller element. Let us call this leaf the *special leaf* and the corresponding path *special path.* Procedure *leaf–search*, as defined below, does this.

Procedure leaf–search(m, i, j)
1) $j := i$.
2) while $2j < m$ do begin
   if $a[2j] < a[2j+1]$ then $j := 2j$
   else $j := 2j+1$
   end.
3) if $2j = m$ then $j := m$.

We now climb up the special path and look for the destination position of the former leaf element which is the same position as computed by the *rearrange* procedure given above.

This is done by the procedure *bottom–up–search* as defined below.

Procedure bottom–up–search(i, j)
(*j is the output of *leaf–search*(m, i, j)*)
1) while $a[i] < a[j]$ do $j := \lfloor j/2 \rfloor$.

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* given below.

Procedure interchange(i, j)
1) $l := \lfloor \log(j/i) \rfloor$, $x := a[j]$, $a[j] := a[i]$.
2) while $j > i$ do begin
interchange $a[\lfloor j/2 \rfloor]$ and x
$j := \lfloor j/2 \rfloor$
end.

Thus for Bottom–Up–Heapsort, we replace *rearrange*(m, i) in Heapsort by *bottom–up–rearrange*(m, i) as defined below.

Procedure bottom–up–rearrange(m, i)
1) *leaf–search*(m, i, j);
2) *bottom–up–search*(i, j);
3) *interchange*(i, j).

III. The Upper Bound

Let n be the number of elements in the heap with $n = 2^k - 1$. H_k is a heap for n elements. The elements in the heap are the numbers $1, ..., n$.

The argument for the upper bound ([W89]) is as follows.
Let us call the elements $1, ..., 2^{k-1}$ *small* and the elements $2^{k-1} + 1, ..., 2^k - 1$ *big*. (In sections V and VI respectively we will give other definitions for small and big.) We consider only the 2^{k-1} leaf elements of the heap. Since the tree is a heap, at most 2^{k-2} leaf elements are small. All large leaf elements are still contained in the heap created after the first 2^{k-1} deletions. Since the number of these elements is at least 2^{k-2}, their average depth in the created H_{k-1} is at least $k - 4$. Hence the number of comparisons for the first 2^{k-1} calls of *bottom-up-search* is bounded by

$$4 \times 2^{k-2} + k \times 2^{k-2} = \frac{1}{4} n \log n + n$$

and the time bound follows by summing over all levels.

IV. An Example for Worst Case Construction

Let $T_L(i)$ ($T_R(i)$) be the left (right) subtree of the node at position i. We call the movement of a leaf element along a special path as a *trip*.
Let T be a tree and S a set of neighbouring leaves in the tree. Then *subtree*(S) denotes the subtree rooted at the nearest common ancestor of the nodes in S, *nca*(S). It will always be clear from the context which T is meant.
The *subtree ancestors* of a set S of neighbouring leaves are the internal nodes of subtree(S). *Exclusive ancestors* of a set S denote the ancestors that only have leaf descendants in S.
Let further m be an integer which we choose later and $p = 2^m - 1$.

Let us consider only the first 2^{k-2} deletions. Our goal is to send as many leaf elements as possible to some positions near the root of the heap, for these elements have to perform a lot of comparisons. Especially, we want to send the leaf elements of $T_R(1)$ to the topmost region of $T_L(1)$. Let us denote this region – we choose it to be a complete binary tree with exactly p nodes – by $TT_L(1,p)$ (Toptree of $T_L(1)$ with p nodes) (Fig. 1).

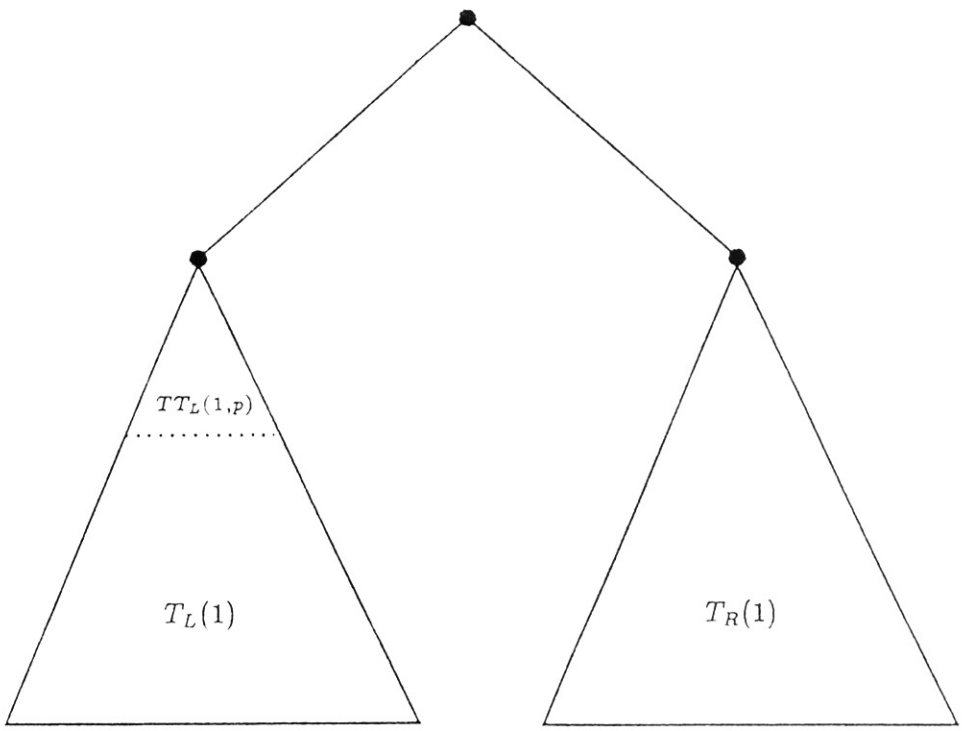

Figure 1

We first consider a small example as in Fig. 2 where $k = 5$ and $p = 1$. The numbers beside the nodes are the stored elements. The contents of the nodes are immaterial at those positions where no values are shown.

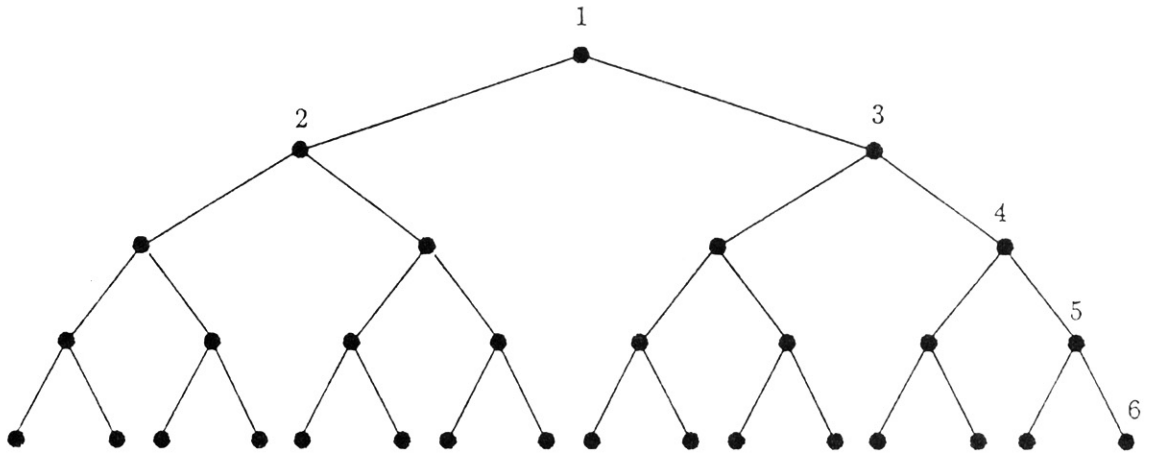

Figure 2

In this heap, during the first deletion step of BUH element 6 will be placed at position 2, where it replaces element 2. But note that before any other element will be placed at some position in $TT_L(1,1)$, all ancestor elements of the rightmost leaf have to be eliminated from the heap. In our example the next 3 trips (in general the next $k-2$ trips) have to go through $T_R(1)$. Thus it is cleverer to choose p bigger and thereby ensure that we can send more than one element in consecutive trips to $TT_L(1,p)$. Therefore we divide the deletions of the first 2^{k-2} elements into phases P_i, where each phase consists of a pair $P_i = (S_i, G_i)$ of sets of leaf nodes, so that the leaf elements in S_i all make their trips through $T_L(1)$ followed by the trips of all nodes in G_i through $T_R(1)$.

Consider the following example for $k = 14$ and $p = 7$ in Fig. 3.

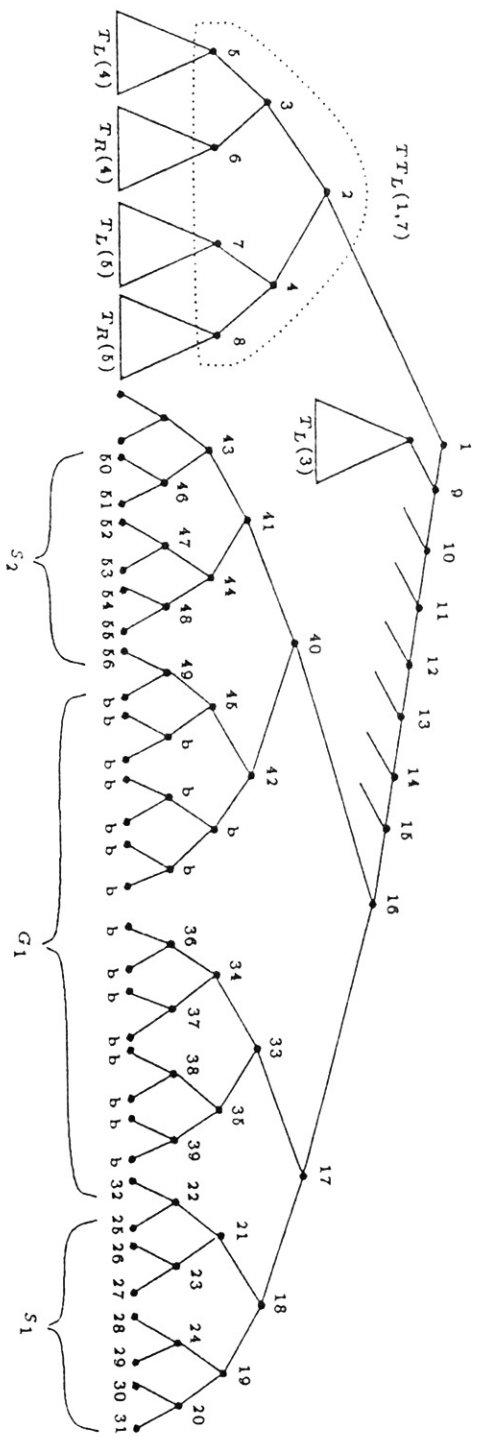

Figure 3

The elements $2, ..., 8$ are placed in $TT_L(1,7)$. The next 16 smallest elements $(9, ..., 24)$ are all placed at the ancestor nodes of S_1. Then we place the next 7 elements $(25, ..., 31)$ in S_1 in descending order from right to left. Our aim is that the elements in the nodes of a S_i will all make their trips through $T_L(1)$ to a position in $TT_L(1,p)$. This is certainly fulfilled in our example for the elements in S_1.

All S_i have the same number of leaf nodes, namely p. The elements in the nodes of G_i (G stands for 'gap' between two successive sets of p trips through $T_L(i)$) make all their trips consecutively through $T_R(1)$. They do the job of eliminating all ancestor elements of S_i from the heap. We count only the comparisons necessary for elements in nodes of a S_i. Since we want to send the next elements in S_{i+1} as soon as possible, we essentially choose the elements in G_i and in their exclusive ancestors as big (marked as b in Fig. 3).

But in the example there are also some elements in G_1 and their ancestors which are small. This has the following reason. We have to eliminate all ancestor elements of S_i. Some of them are also ancestors of S_{i+1} (e.g., the elements $9, ..., 16$ in Fig. 3). These elements must be replaced by small elements, the others are replaced by big elements. The trips which replace the small ancestor elements by small elements should not start in S_{i+1} since then they would destroy the whole structure. We force them to start in G_i by placing some small elements in the exclusive ancestors of G_i and, if necessary, in G_i itself. In the example these are the elements $32, ..., 39$.

This strategy is now repeated until we have no further empty leaf in $T_R(1)$. We formally describe the construction process in the next section.

V. The Construction

Let us assume that k is very large and that we have already chosen m (and so p). By saying 'place some elements' we would always mean that the concrete placement does not matter but that the heap property is always preserved.

Remark: In this section, an element is called *small*, if it is placed in one of the steps 0) to 14). If it belongs to the remaining elements handled in step 15), it is called *big*.

Heapconstruction

0) Place the element 1 at the root.

1) Place the p smallest elements $2, ..., p+1$ in the topmost positions of $T_L(1)$.

Remark: In section VI we will specify the exact placement.

2) Identify the p rightmost leaves (the elements of S_1) and all their ancestors, excluding the root.

Remark: The number of these ancestors is $k-m-2+p$, since S_1 has exactly p subtree ancestors and there are $k-m-2$ nodes between the root of H_k and the root of subtree(S_1).

3) Place the next $k-m-2+p$ smallest elements starting from the value $p+2$ at these ancestors of S_1.

4) Place the next p smallest elements at the leaf position of S_1 from right to left in descending order.

Remark: All these elements will make their trips through $T_L(1)$ and will be placed in $TT_L(1,p)$. If we place them in S_1 from right to left in descending order, they will need at least as many comparisons as in any other case, but for the asymptotic time bound this does not matter. In section VI, however, it will be necessary to place the elements in this order.

5) $i := 1$.

6) Let g denote the number of elements in G_i. $g := k-m-2+p$.

7) Identify all leaves in G_i, the next p leaves (that is S_{i+1}) and all ancestors of S_{i+1}.

8.) Let r be the rightmost leaf of S_{i+1}. Identify the nearest common ancestor of S_i and r, denoted by $nca(S_i \cup \{r\})$.

Remark: Note that as long as there are small elements in ancestors of S_i below this nca, the trips will not start from a leaf to the left of G_i.

9) Count the number of ancestors of S_i below nca$(S_i \cup \{r\})$, denoted by c.

10) Place the next $g-c$ smallest possible elements in the exclusive ancestors and, if required, also in the leaf positions of G_i.

Remark: These are the elements that are necessary to eliminate the ancestors of S_i above the nca$(S_i \cup \{r\})$ without destroying the structure of the heap to the left of G_i. Because of these elements, no trip will start from a node of S_{i+1} during the deletions of the nodes in G_i.

'Smallest possible' (instead of smallest) takes care of the fact that we eventually may have to place some small elements at some non–exclusive ancestors of G_i in order to reach the exclusive ancestors of G_i under preservation of the heap property.

11) Fill up the ancestral positions of S_{i+1} that have not received any value so far.

12) Compute g as the number of elements of G_{i+1} as follows: $g = g - c + |A(S_{i+1})|$ where $A(S_{i+1})$ denotes the set of all ancestors of S_{i+1} below $\mathrm{nca}(S_i \cup \{r\})$.

13) $i := i + 1$.

14) If there are still more than $g+p$ leaves in $T_R(1)$ then place the next p smallest elements at the positions of S_i from right to left in descending order and go to 7.).

15) Fill up all empty positions of H_k with the remaining elements and STOP.

VI. The Extension to the whole tree

We now want to change the construction so that it can be extended to $T_L(1)$, that means, we now consider the deletions of leaves from the left subtree $T_L(1)$ and we want to send small leaf elements of $T_L(1)$ to $TT_R(1,p)$.

Remark: In this section, an element that is placed in one of the steps 1) to 20) is called *small*; if it belongs to the remaining elements handled in step 21), it is called *big*.

In the construction algorithm of section V, we created as many complete S–sets as possible. Let now S_l be the last created S–set and $r \geq 0$ be the number of the remaining leaves in $T_R(1)$.
Note that when BUH begins with the deletions in S_{l+1} (which is in $T_L(1)$), we want to have the following situation:
The elements actually placed in $TT_L(1,p)$ are the former leaf elements of S_l. Since the elements of S_{l+1} shall replace the elements in $TT_R(1,p)$, the elements in $TT_R(1,p)$ have to be the smallest in the heap. In particular, they have to be smaller than those in $TT_L(1,p)$. Thus they have to be bigger than the elements initially placed in S_{l+1} and smaller than those in S_l.

So there are essentially two problems to solve:

1) Before the bottom–up–heapsort algorithm reaches the deletions of S_{l+1} (which is in $T_L(1)$), the p smallest elements of the then heap have to be placed in $TT_R(1,p)$.

2) At the same time, the elements in $TT_L(1,p)$ have to fit in the aimed structure so that the leaves in a G_i of $T_L(1)$ do not start their trips from left of G_i. So the last elements inserted in $TT_L(1,p)$ have to be chosen carefully.

In order to solve the mentioned problems, we replace step 15) in *Heapconstruction* by the following steps 15) to 21).

15) If there are more than $g+p$ leaves in $T_R(1)$ then place the next p smallest elements in S_i and go to 7.).

16) Compute r. If $r \leq g$, we identify the g leaves (some of them may be in $T_L(1)$) and go to 18).

17) Now $r > g$ holds. Let all the r remaining leaves build G_l; place the next smallest elements in the ancestors of G_l and the leaves, if necessary, so that all trips of the elements in G_l go through $T_R(1)$.

Remark: The trips in G_l are not allowed to destroy the actual structure of $T_L(1)$.
At this point the nodes in S_l are yet to receive any concrete value.

18) This is the step that solves problem 1) mentioned above, that means we now want to place the p elements which have to be placed in $TT_R(1,p)$ before the deletions of the nodes in S_{l+1} take place. But before we give the placing rule, we want to motivate it.
Let now k and m be chosen such that each subtree of a leave of $TT_R(1,p)$ contains at least one set S_i completely. Then each node in $TT_R(1,p)$ initially contains certainly a small element. Let us first suppose that H_k is constructed according to *Heapconstruction* of section V. Under this assumption we now consider the steps of BUH on H_k.

For each node v of $TT_R(1,p)$ there is a step, in which v receives for the first time a big element. If we consider the chronological order of these steps we can recognize a pattern.

Lemma: For all nodes v of $TT_R(1,p)$:
a) When v receives a big element for the first time, its right neighbour already contains a big one.

b) If v then is a left child, its direct ancestor will receive a big element during the next deletion step.

Proof: Note that we delete the S–sets from right to left. During the deletions of the nodes in a G_i we replace all those ancestor elements of S_i that are not also ancestor elements of S_{i+1}; an ancestor element of both S_i and S_{i+1} is replaced by a small element (because of step 10)).

a) Each subtree of a leaf of $TT_R(1,p)$ contains at least one S_i completely. Then the same also holds for each internal node of $TT_R(1,p)$. Let S_i be such a S–set for the node v. If BUH starts with the deletions of the nodes in S_i, the right neighbour of v cannot be an ancestor of a still existing S–set (the S_j for $j < i$ are all deleted then) and so it contains already a big element.

b) When a node v which is a left child receives a big element for the first time, its former small element is placed in its direct ancestor. Since its right neighbour already contains a big element and its direct ancestor can not be an ancestor of a still existing S–set, this node will receive a big element during the next deletion step.

■

The lemma shows the following. The replacement of the small elements in $TT_R(1,p)$ takes place in rounds where in each round the elements in a group of nodes are replaced. In the first round the rigthmost leaf in $TT_R(1,p)$ receives a big element. In the i–th round, first the rightmost leaf of $TT_R(1,p)$ that still contains a small element (that is the i–th leaf counted from right to left) gets a big one. Then if the node is a left child, its direct ancestor will get a big element and so on.

We give an example for $p = 15$; **the numbers above the nodes give the node positions (and not the contained elements).**

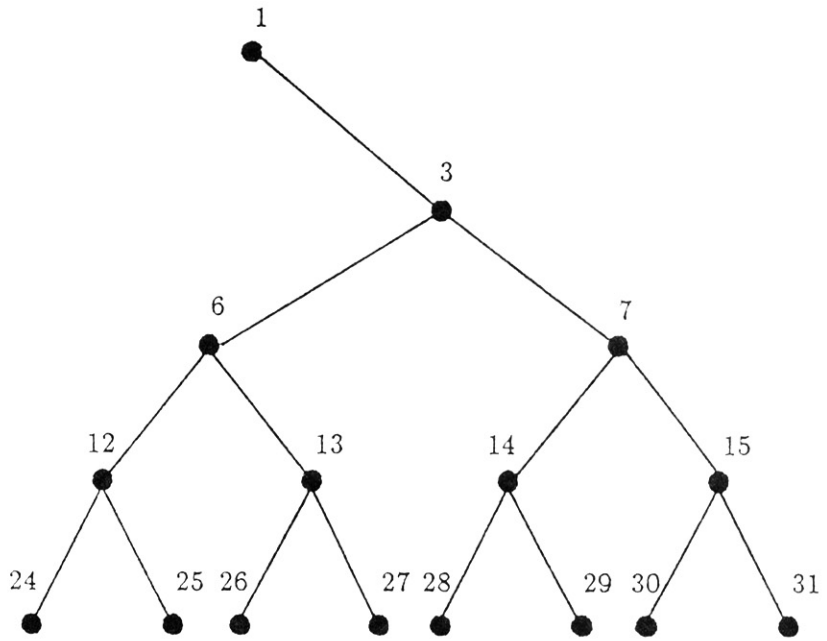

Figure 4

The mentioned groups in the example are the sets: $\{31\}$, $\{30, 15\}$, $\{29\}$, $\{28, 14, 7\}$, $\{27\}$, $\{26, 13\}$, $\{25\}$, $\{24, 12, 6, 3\}$. Note, that the nodes in each group build a path. Each leaf of $TT_R(1, p)$ represents a group and each group has a group root, namely the node with the smallest index within the group. The maximal cardinality of a group is m.
Now let us return to the *Heapconstruction* of this section. Instead with big elements as in the lemma we want to fill up $TT_R(1, p)$ with the next p smallest elements available after step 17). Especially we want that after the replacement of all small elements of a group by some of these p elements, the contents of the nodes in the group remains unchanged till BUH reaches S_{l+1}. Therefore we order the groups from right to left according to the positions of the corresponding leaves. We now place the p smallest elements available after step 17) in the subtrees of these groups. In each subtree are so many elements placed as the number of those contained in the corresponding group. Furthermore the elements are placed instead of some big elements, which were placed by *Heapconstruction* of section V, and they are only placed at internal nodes, since leaf elements could go into wrong subtrees. Finally the elements are placed in descending order from the right group to the left, that means the elements placed in the subtree of a group are all bigger than all elements placed in any subtree of a group to the left of it. So when a group is filled by the steps of BUH with these elements, no further trip will go through a node of this group until the deletions of S_{l+1} take place.
This solves problem 1).

19) Now fill the nodes in $T_L(1)$ according to the same strategy as used by *Heapconstruction* given in section V, that means create as many S–sets as possible by steps similar to the steps 6) to 14) but in $T_L(1)$ instead of $T_R(1)$. Ignore the fact, that the nodes of $TT_L(1,p)$ already have some elements (namely the elements $2,...,p+1$). Treat them as if they were empty.

Remark: Note, that in the case $r \leq g$ (in step 16)) we eventually have to start in $T_L(1)$ with some leaves in G_l.

20) Having finished the construction place the elements that, according to step 19), are in the nodes of $TT_L(1,p)$, in the so far empty nodes of S_l in descending order from right to left, since these are the elements which must be placed in $TT_L(1,p)$ before we delete S_{l+1}. The nodes in $TT_L(1,p)$ now again contain the elements placed in step 1).

21) Fill up all the empty positions of H_k with the remaining elements and STOP.

Note, that each element in S_l has a definite destination node in $TT_L(1,p)$. From the construction algorithm (the elements in S_l are those being placed in step 19) in $TT_L(1,p)$) follows that these elements are not consecutive numbers and that they have to be placed in $TT_L(1,p)$ in reverse preorder, that means in preorder from right to left.

This fact induces some restrictions on the placement of elements in $TT_L(1,p)$ right at the very beginning (step 1)) of *Heapconstruction*. This is explained below.
Let us suppose, that we have a heap whose elements are arranged in preorder. If we replace the elements by bigger ones where these are inserted in descending order, it is easy to prove, that the resulting tree is organized in reversed preorder. The same holds vice versa. Let us consider an example for $p = 7$.

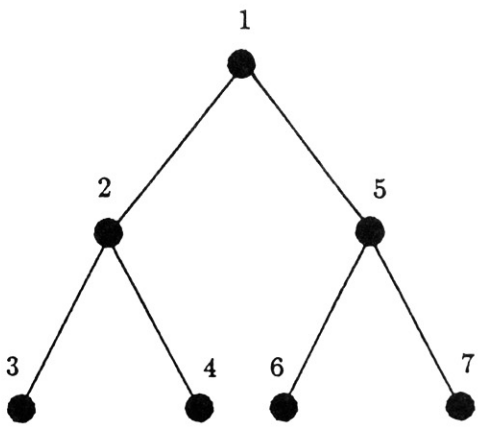

Figure 5

Let us replace the elements by $8, ..., 14$ in descending order. The resulting tree is

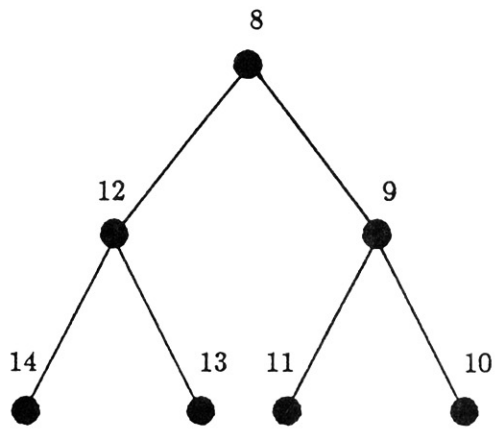

Figure 6

Now we replace the elements by $15, ..., 21$ and we get

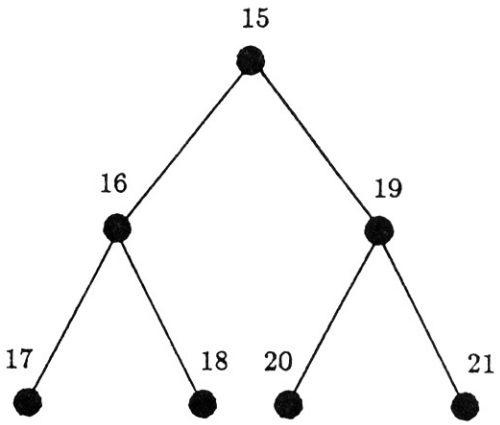

Figure 7

It is also easy to see, that an inserted element is placed directly at its destination node. So we get an alternating sequence of preordered and reverse preordered trees.
Thus, if l is odd, we start at the beginning (that is step 1) in *Heapconstruction*) with the elements $2, ..., p+1$ in preorder and if l is even in reverse preorder in $TT_L(1,p)$. The elements in S_l will then be placed directly in their destination nodes.
This solves problem 2).

VII. Analysis

In order to analyze the running time, we prove the following lemma.

Lemma: For all suitable i:

a) The deletions of the elements placed in G_i have no influence on the contents of the leaves to the left of G_i.

b) All elements of S_i in $T_R(1)$ and $T_L(1)$ will be inserted in $TT_L(1,p)$ and $TT_R(1,p)$ respectively.

Proof: We use induction on i:
$i = 1$:
a) According to step 3) of the construction algorithm, the first c elements are necessarily placed at nodes in $A(S_1)$. If any of the next $g - c$ leaf elements would start its trip to the left of G_1, this would contradict step 10) of the construction algorithm, as one easily sees.

b) This is obvious.

$i \rightarrow i + 1$:
a) The argument is the same as in the case $i = 1$.

b) Suppose, we just deleted the last node of S_i. Following our construction algorithm, the only elements, that are smaller than those now placed in $TT_L(1,p)$ (resp. $TT_R(1,p)$) are the elements in the ancestors of S_i. But all these elements will be eliminated from the heap by deleting the nodes of G_i. The elements in S_{i+1} remain unchanged (because of part a)). So the next p trips have to go through $T_L(1)$ (resp. $T_R(1)$) and since the elements in S_{i+1} are smaller than those below $TT_L(1,p)$ ($TT_R(1,p)$), they usually replace the former elements of S_i (The only exception is when there is a transition from $T_R(1)$ to $T_L(1)$).

∎

Now in order to prove a lower bound on C_k, we prove an upper bound on the length of G_i. Note, that $|G_1| = p + k - m - 2$ and $|A(S_i)| = p + a_i$ for some $a_i \geq 0, a_0 = 0$. Let $x = k - m - 2$.
According to step 13) of our construction algorithm, we find that $|G_{i+1}| = |G_i| - c + |A(S_{i+1})|$. Now note, that x is an upper bound for $|G_i| - c$, since $|G_i| - c$ is the number of nodes on the path from the root to $\mathrm{nca}(S_i \cup \{r\})$, excluding the root. So we have

$$|G_i| \leq x + |A(S_i)|.$$

Now let us compute an upper bound for $|A(S_i)|$.

Lemma: Let T be a complete binary tree with $2^k - 1$ nodes and S a set of p neighbouring leaves. The maximum number of ancestors of S in T is not bigger than $p + 2k$.

Proof: Let p_i be the number of ancestors of S of height i. Since all ancestors of the same height i are neighbours, $p_{i+1} \leq \lfloor p_i/2 \rfloor + 2$; let h be the height of subtree(S). Then

$$\sum_{i=1}^{h} p_i \leq p + 2h$$

.

Together with the ancestors on the path from the root of T to subtree(S), this number is smaller than $p + 2k$. Since $|S_i| = p$ for all i and $x \leq k$, it follows that for all i: $|G_i| + |S_i| \leq 2p + 3k$.

■

Each S_i contributes at least $p(k-m)$ comparisons to C_k and so

$$C_k \geq 2\lfloor \frac{2^{k-2}}{2p+3k} \rfloor p(k-m)$$

If we choose m such that $k^l \leq p \leq k^{l+1}$ for some fixed $l \gg 1$, the asymptotic lower bound is $\frac{1}{4} n \log n - O(n \log \log n)$.

Finally, note that the construction of the sections V and VI is always possible. Concerning step 10), there are always enough nodes in G_i and its exclusive ancestors to place the $g - c$ small elements, because $|G_i| = g > g - c$.

Furthermore, if we choose k big enough, the number of phases in a subtree rooted at a leaf of $TT_R(1,p)$ is at least 1 and all nodes in $TT_R(1,p)$ of the constructed heap will contain small elements.

The maximal cardinality of a group mentioned in section VI, step 18) is m. If we observe, that in each G_i at least $c \geq p$ leaves contain big elements, we can be sure that there are at least m internal nodes in each subtree at a leaf in $TT_R(1,p)$ suitable to place the elements of step 18).

VIII. Conclusions

The lower bound proof presented in this paper is optimal in the sense, that if we consider only the first 2^{k-1} deletions, we cannot get a better bound. We didn't say anything about the concrete placement of the big elements, but it seems to be difficult to iterate the heap structure for the resulting heaps H_{k-1}, H_{k-2}, ... Nevertheless we conjecture that the upper bound of $\frac{3}{2}n \log n$ is sharp.

Acknoledgements

The authors would like to express their sincerest thanks to Kurt Mehlhorn for first pointing out the problem and also for his constant inspiration and continuous support to carry out this work. We also would like to thank Miklos Santha for helpful discussions.

IX. References

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

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

[MDR89] C.J.H. Mc Diarmid and B.A. Reed: Building heaps fast. *Journal of Algorithms 10, 352 - 365*, 1989

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

[W89] I. Wegener: BOTTOM–UP–HEAPSORT, a new variant of HEAPSORT beating on average QUICKSORT (if n is not very small). *Submitted to Journal of Algorithms*, 1989

[W90] I. Wegener: The worst case complexity of Mc Diarmid and Reed's variant of BOTTOM–UP–HEAPSORT is less than $n \log n + 1.1n$. *??*, 1990

[Wi64] J.W.J. Williams: Algorithm 232. *Communications of the ACM 7, 347 - 348*, 1964