Triangular Heaps

Henk J.M. Goeman
Walter A. Kosters

Department of Mathematics and Computer Science, Leiden University,

P.O. Box 9512, 2300 RA Leiden, The Netherlands

Email: {goeman,kosters}@wi.LeidenUniv.nl

Abstract

In this paper we introduce the *triangular heap*, a heap with the special property that for every father node its right child (if present) is smaller than its left child. We show how triangular heaps can be applied to the traditional problem of sorting an array in situ in ways quite similar to well-known methods using ordinary heaps. An average case analysis is presented for the construction and for the sorting process of both ordinary and triangular heaps.
Keywords: data structures, analysis of algorithms, heaps, heapsort.

1 Introduction

In this paper we propose the *triangular heap*, a heap with the special property that for every father node (the key of) its right child —if present— is smaller than (the key of) its left child. Whereas in a heap the largest node always takes the first position (i.e. the root), in a triangular heap both the largest and the second largest node can always be found in first and second position (i.e. in the root and in the left child of the root).
The heap and its use for sorting an array in situ were first introduced by Williams and Floyd, see [11] and [3]. An important part of the algorithm —the procedure siftdown— builds a heap from a rootnode and two smaller heaps, represented in its two subtrees, by repeatedly interchanging the node with the largest of its children until the node eventually reaches a position where it is larger than its children (if any). This requires two comparisons for each level encountered.
The heap construction algorithm consists of a bottom-up series of these operations. Repeatedly removing the largest node and re-establishing the heap structure for the remaining nodes, again with the procedure siftdown, gives rise to a sorting method called heapsort.

Efficient and improved implementations of this method were, for instance, given in [1], [7], [9] and [10]. These implementations contain more and more efficient realisations of the procedure siftdown (effectively resulting in the same heap) where first a vacant position is sent down to the lowest level at a cost of only one comparison per level, and then the new node is inserted in its proper place in a bottom-up fashion, again at a cost of only one comparison for each level. The name trickledown is commonly used to refer to these improved realisations of siftdown.
Note that in a triangular heap more relations between the nodes of the tree are —in a sense— remembered in the tree structure. Thus the triangular heap structure reveals more information on the ordering of its nodes than the ordinary heap structure. Yet it turns out that this extra information can be easily maintained.
The central question here is: is it possible to exploit the extra available information in applications of the triangular heap in such a way that this outweighs the extra effort to maintain the stronger structure? In this paper we shall consider that question applying the triangular heap to the traditional problem of sorting an array in situ. This will result in several new sorting algorithms using procedures quite similar to those often used for the ordinary heap structure.

2 Definitions and Prerequisites

A *heap* is a finite binary tree on a totally ordered set such that each node is larger than its children. Whenever a finite binary tree is a heap we will say that the tree satisfies the *heap property*.
Note that for a heap the underlying complete binary tree corresponds directly to the Hasse diagram of the partial order of known relations between the nodes as implied by the heap structure. To obtain the Hasse diagram one should just ignore the partitioning of outgoing edges in left and right ones.
When applying the heap structure to the problem of sorting in situ an array A with indexset $\{1 \ldots n\}$, the heaps are usually represented within the array itself where node i ($1 \leq i \leq n$) has $2i$ as its left child (if $2i \leq n$) and $2i + 1$ as its right child (if $2i + 1 \leq n$). For the whole array to be a heap we should have $A[i] < A[i \text{ div } 2]$ for every i with $1 < i \leq n$. Indeed, for every node i with $1 < i \leq n$ its parent is node i div 2.
With this representation a heap is always a *complete* binary tree. Here complete means that nodes may be absent only at the bottom level of the tree and there only as far to the right as possible.

Now, a *triangular heap* is a heap with the special property that for every father node, having two children, its right child is smaller than its left child. Whenever a finite binary tree is a triangular heap we will say that the tree satisfies the *triangular heap property*.

For a triangular heap the underlying complete binary tree *does not* correspond directly to the Hasse diagram of the partial order of known relations between the nodes as implied by the triangular heap structure. However, as we shall see shortly, we can easily construct another heap for the same set of nodes, whose corresponding Hasse diagram will also be the Hasse diagram of implied relations for the original triangular heap.
There is a well-known and important construction, that establishes a 1-1 correspondence between ordered forests and binary trees. To get the corresponding binary tree from a given ordered forest one should just remove all the original edges and then install a left outgoing edge for each original parent node to its original left-most child and a right outgoing edge for each node to its original next sibling (if it had any). This construction is sometimes called the left-child, right sibling representation.
If we apply this construction to a forest, containing just one binary tree T, we will get another binary tree, which will be called its *stretched form*. The new binary tree will be a heap if and only if the original tree is a triangular heap, and its Hasse diagram is just the Hasse diagram for the original triangular heap.
So, to be a triangular heap, the complete binary tree, represented as above in the array A with indexset $\{1 \ldots n\}$, should have for every i with $1 < i \leq n$, $A[i] < A[i \text{ div } 2]$ if even(i), and $A[i] < A[i-1]$ if odd(i). Its stretched form is then easily recognised to be a binary tree, again satisfying the heap property, where node i $(1 \leq i \leq n)$ has $2i$ as its left child (if $2i \leq n$), and $i+1$ as its right child (if $i+1 \leq n$ and even(i)). And, indeed, every node i with $1 < i \leq n$ now has node i div 2 as its parent if even(i) and it has node $i-1$ as its parent if odd(i). Of course, for $n > 2$ the resulting tree is not complete any more, although it still occupies just the indexset $\{1 \ldots n\}$.

Throughout this paper lg will denote the binary logarithm. The height of a tree is defined in a usual way: the height of an empty tree equals 0, and the height of a non-empty tree is equal to 1 plus the maximum of the heights of the subtrees of the root.

3 Implementations and Algorithms

In this section we will present our algorithms in a procedural notation using a call by value parameter mechanism. The algorithms will be given in a clear and compact form. Several obvious variants and optimisations could have been chosen as well, without a significant effect on the analysis. For instance, when we count the number of swaps in the procedure SiftDown below, it can also be read as counting the number of data movements in an obvious variant for this procedure SiftDown, where a cyclic exchange is done instead of the repetition of swaps.

Let an integer m and an array

$$A : \textbf{array } 1 \ldots m \textbf{ of } SomeType$$

be given, where $m > 0$ and where *SomeType* is a totally ordered set. We will assume that elements in different positions of the array are always different.

3.1 Procedures for HeapSort

We first present and explain the program *HeapSort* which sorts an array in situ using ordinary heaps.

Obviously, the following procedure Swap(k, j) interchanges two elements of the array A, whenever $1 \leq k \leq m$, $1 \leq j \leq m$ and $k \neq j$.

```
Swap(k, j) :
  A(k), A(j) := A(j), A(k)
```

For $1 \leq k \leq n \leq m$, the procedure SiftDown(k, n) reorders the subtree in the array A, rooted at node k and restricted to the indexset $1 \ldots n$, into a heap, whenever both of its subtrees rooted at node $2k$ and at node $2k + 1$, again restricted to the indexset $1 \ldots n$, are heaps already (they can be empty).

```
SiftDown(k, n) :
  while k ≤ n div 2 do
    j := 2 * k ; if j < n then if A(j) < A(j + 1) then j := j + 1 fi fi ;
    if A(k) < A(j) then Swap(k, j) ; k := j else k := n fi
  od
```

Then, the procedure MakeHeap establishes the heap property for the array A. This will also be called the construction phase of the sorting process.

```
MakeHeap :
  k := m div 2 ; while k > 0 do SiftDown(k, m) ; k := k − 1 od
```

Now, finally, the procedure HeapSort sorts the array A in situ.

```
HeapSort :
  n := m ; MakeHeap ;
  while n > 1 do Swap(1, n) ; n := n − 1 ; SiftDown(1, n) od
```

The procedure TrickleDown(k, n), a trickledown version of SiftDown, may replace SiftDown everywhere above, and can be given as follows:

```
TrickleDown(k, n) :
  t := A(k); l := k;
  while k ≤ n div 2 do
    j := 2 * k; if j < n then if A(j) < A(j + 1) then j := j + 1 fi fi;
    A(k) := A(j); k := j
  od;
  i := k;
  while k > l do
    j := k div 2; if t > A(j) then A(k) := A(j); k := j; i := k else k := l fi
  od;
  A(i) := t
```

3.2 Procedures for Triangular HeapSort

We can now present and explain the program *TriangularHeapSort* which sorts an array in situ using triangular heaps.

For $1 \leq k \leq n \leq m$, the procedure TriangularSiftDown(k, n) reorders for the *stretched form* of the tree in array A, the subtree rooted at node k into a heap, whenever its subtrees are heaps already. As we explained earlier, now every node involved is larger than its right sibling and its left-most child (if present) as seen in the original *unstretched form* of the tree in array A.

```
TriangularSiftDown(k, n) :
  while k ≤ n div 2 ∨ (even(k) ∧ k < n) do
    if k ≤ n div 2 then
      j := 2 * k; if even(k) then if A(j) < A(k + 1) then j := k + 1 fi fi
    else
      j := k + 1
    fi;
    if A(k) < A(j) then Swap(k, j); k := j else k := n fi
  od
```

Then, the analogon of the procedure MakeHeap is the procedure MakeTriangularHeap establishing the triangular heap property for the array A. Note especially that it is necessary here to start at $m - 1$, instead of m div 2.

```
MakeTriangularHeap :
  k := m − 1; while k > 0 do TriangularSiftDown(k, m); k := k − 1 od
```

Again, finally, the procedure TriangularHeapSort sorts the array A in situ. This algorithm is an easy analogon of the well-known algorithm for HeapSort.

```
TriangularHeapSort :
  n := m ; MakeTriangularHeap;
  while n > 1 do Swap(1, n); n := n − 1 ; TriangularSiftDown(1, n) od
```

We also have a TrickleDown version of TriangularSiftDown, namely the procedure TriangularTrickleDown(k, n), which can be given as follows:

```
TriangularTrickleDown(k, n) :
  t := A(k); l := k;
  while k ≤ n div 2 ∨ (even(k) ∧ k < n) do
    if k ≤ n div 2 then
      j := 2 ∗ k; if even(k) then if A(j) < A(k + 1) then j := k + 1 fi fi
    else
      j := k + 1
    fi;
    A(k) := A(j); k := j
  od;
  i := k;
  while k > l do
    if odd(k) then j := k − 1 else j := k div 2 fi;
    if t > A(j) then A(k) := A(j); k := j; i := k else k := l fi
  od;
  A(i) := t
```

All these algoritms will be compared and analysed in the following sections.

4 Counting Heaps and Triangular Heaps

If $f(n)$ denotes the number of heaps on n distinct keys, then (see [8], p. 79)

$$f(n) = n! \,/ \prod_{1 \le k \le n} S(k, n),$$

where for $1 \le k \le n$, $S(k, n)$ is the size of the subtree rooted at k.
Analogous to the formula for $f(n)$ we can compute $g(n)$, the number of triangular heaps on n distinct keys. It is easily seen that

$$\frac{g(n)}{(n-1)!} = \frac{1}{(n-1)s_2} \frac{g(s_1)}{(s_1-1)!} \frac{g(s_2)}{(s_2-1)!},$$

where s_1 and s_2 are the sizes of the subtrees of the root, $S(2, n)$ resp. $S(3, n)$. They satisfy $s_1 + s_2 = n - 1$. This formula leads to

$$g(n) = (n-1)! \,/ \prod_{1 \le k \le (n-1) \operatorname{div} 2} (S(k, n) - 1)\ S(2k + 1, n).$$

If we take the tree in its *stretched* form and interpret $S(k,n)$ as the size of the subtree rooted at k in that tree, then we have again, as should be expected,

$$g(n) = n! \,/ \prod_{1 \le k \le n} S(k,n).$$

If $n = 2^t - 1$ for some integer t then $g(n) = f(n)/2^{(n-1)/2}$. This last result also follows easily if one notes that an ordinary heap with $2^t - 1$ elements can be turned into a triangular heap in a unique way by interchanging subtrees whenever the roots of these subtrees are not in correct order. If $n = 2^t - 1$ one can not lose completeness here.

5 An Analysis of the Algorithms

In this section we will analyse the algorithms we have given before. Sorting an array in situ using heaps consists of two parts: the construction phase and the sorting phase. We will examine these phases in separate sections.

5.1 The Construction Phase

Since every triangular heap is an "ordinary" heap, it is clear that in general the construction requires more comparisons of array elements, and also more data movements. It is easy to show that the number of comparisons needed to construct a heap from a random permutation of $\{1, 2, \ldots, n\}$ (where $n = 2^k - 1$ for some integer $k > 0$), using the siftdown algorithm, is always between $n - 1$ and $2n - 2\lg(n+1)$. Using the trickledown version of this algorithm, the number of comparisons is between $\frac{3}{2}n - \lg(n+1) - \frac{1}{2}$ and $2n - 2\lg(n+1)$. The number of data movements (or swaps) is between 0 and $n - \lg(n+1)$, both for the siftdown version and the trickledown version (the arrays elements are only moved when the correct place for the "new node" —which is the same for both algorithms— is found).

In the following the next lemma will be used frequently.

Lemma. (i) Suppose that real numbers a, b and c, and a function G are given. The solution of the recurrence

$$M(n) = 2\,M(\frac{n-1}{2}) + a\lg(n+1) + b + G(n), \quad M(3) = c,$$

where $n = 2^k - 1$ for integer $k \ge 3$, is given by

$$M(n) = \left\{ \sum_{l=3}^{\lg(n+1)} \frac{G(2^l - 1)}{2^l} + a + \frac{b+c}{4} \right\} (n+1) - a\lg(n+1) - (2a+b).$$

(ii) Suppose that real numbers a, b, c and d are given. Let, for $n = 2^k - 1$ with integer $k > 0$,

$$L(n) = \frac{\lg(n+1)}{n}(an + b) + c + \frac{d}{n}$$

and

$$G(n) = (a - b)\frac{\lg(n+1)}{n} + \frac{-a + 2b + c - d}{n} + a.$$

Then $L(n)$ satisfies the recurrence

$$L(n) = G(n) + \frac{n-1}{n}L(\frac{n-1}{2}).$$

Proof. Straightforward. □

For triangular heaps we have:

Theorem. The number of comparisons needed to construct a triangular heap from a random permutation of $\{1, 2, \ldots, n\}$ (where $n = 2^k - 1$ for some integer $k > 0$), using the staightforward algorithm, is between $n - 1$ and $\frac{13}{4}n - 3\lg(n+1) - \frac{3}{4}$, and using the trickledown version between $\frac{3}{2}n - \lg(n+1) - \frac{1}{2}$ and $\frac{13}{4}n - 3\lg(n+1) - \frac{3}{4}$. If $n = 1$ both upper bounds are 0.

Proof. For the siftdown algorithm, a siftdown at a node that is the root of a subtree of height $h > 1$ takes between 2 and $3h - 2$ comparisons if the node is a left child in the original tree, and between 1 and $3h - 4$ comparisons if the node is a right child in (or the root of) the original tree. For leaves the values are 1, 1, 0 and 0 respectively. For the trickledown version the lower bounds are h for left children and $h - 1$ for right children; the upper bounds are the same as those for the siftdown algorithm.
The lower bound for the number of comparisons needed for the construction of a triangular heap using the siftdown algorithm follows by solving the recurrence

$$B(n) = 2\,B(\frac{n-1}{2}) + 2, \quad B(1) = 0.$$

The upper bounds for the construction of a triangular heap using the two algorithms coincide. Let $U(n)$ denote this upper bound. In order to construct a triangular heap, the left and right subtrees of the root have to be triangular heaps (requiring at most $U(\frac{n-1}{2}) + 2$ resp. $U(\frac{n-1}{2})$ comparisons; remember that a siftdown at the left child may involve the right subtree), and we should add $3\lg(n+1) - 4$ comparisons for the siftdown at the root. We are led to the recurrence

$$U(n) = 2\,U(\frac{n-1}{2}) + 3\lg(n+1) - 2,$$

with boundary value $U(3) = 3$, from which the formula for the upper bound follows, using the Lemma.
The lower bound for the trickledown version is derived in a similar way. (As we will see shortly, it is also possible to proceed in a more general manner.) □

Doberkat (see [2] or [4], p. 213) proved that the number of comparisons for the siftdown algorithm is approximately $1.88\,n$ in the average case. Using this result Wegener (see [9]) showed that for the trickledown version it is approximately $1.65\,n$. We shall now give similar results for triangular heaps. Our methods are also applicable to ordinary heaps, where they confirm the results of Doberkat and Wegener.
We need a result that holds in a more general setting. Let us consider ordered oriented trees (this means that the children of a node are ordered from left to right). In order to establish the heap property for such a tree we use an obvious generalisation of the siftdown algorithm: in a bottom-up fashion all subtrees are heapified by repeatly interchanging (as long as necessary) the root of the subtree with its largest child. In a similar way we have an obvious generalisation of the trickledown version.
We let $M_1(T)$ resp. $M_2(T)$ denote the average number of comparisons needed to establish the heap property for the ordered oriented tree T using the generalisation of the siftdown version resp. the trickledown version. Here we suppose that all permutations of $\{1, 2, \ldots, t\}$ are equally likely (where t is the number of nodes in T). Let $L_1(T)$ resp. $L_2(T)$ denote the average number of comparisons during the final siftdown resp. trickledown at the root for the siftdown version resp. the trickledown version, and let $H(T)$ be the average number of comparisons needed (for the trickledown version, again during the final trickledown) when the root of T already contains t, the largest number in T. For empty trees T we define $M_j(T) = L_j(T) = 0$ $(j = 1, 2)$. We have:

Proposition. Let T be an ordered oriented tree with $t > 1$ nodes, where the root has $w \geq 1$ (non-empty) subtrees T_1, T_2,..., T_w. Suppose that T_i has t_i nodes $(i = 1, 2, \ldots, w)$. Then, for $j = 1, 2$, we have

$$M_j(T) = \sum_{i=1}^{w} M_j(T_i) + L_j(T),$$

$$L_j(T) = G_j(T) + \sum_{i=1}^{w} \frac{t_i}{t} L_1(T_i),$$

$$G_1(T) = w \quad \text{and} \quad G_2(T) = w - 1 + \frac{1}{t} + \frac{1}{t} H(T),$$

where $H(T)$ satisfies

$$H(T) = w + \sum_{i=1}^{w} \frac{t_i}{t-1} H(T_i)$$

with boundary value $H(T) = 0$ if T has one node.
If T has one node we have $M_1(T) = M_2(T) = L_1(T) = L_2(T) = 0$.

Proof. The probability that just before the final siftdown t is in the root of T equals $1/t$. If a node has w children, it takes $w - 1$ comparisons to find its largest child. If t is in the root, the siftdown algorithm needs w comparisons, whereas the trickledown version needs —by definition— $H(T)$ comparisons. Key t is with probability t_i/t in

subtree T_i, just before the final siftdown. In this case the siftdown algorithm takes (in the average case) $w + L_1(T_i)$ comparisons. In fact, the numbers in T_i turn out to be random.
For the trickledown version the argument is a little more intricate. In this case, if t happens to be in subtree T_i, we need $w - 1 + L_2(T_i)$ comparisons, except for the case when the original root of the tree turns out to be the largest number in T_i (after the removal of t from this tree); this happens with probability $1/t_i$. We then have one extra comparison, leading to a total of

$$L_2(T) = \frac{1}{t}H(T) + \sum_{i=1}^{w} \frac{t_i}{t}(L_2(T_i) + w - 1 + \frac{1}{t_i}) =$$

$$\frac{1}{t}H(T) + \sum_{i=1}^{w} \frac{t_i}{t}L_2(T_i) + \frac{(w-1)(t-1)+w}{t}$$

comparisons; the desired formula easily follows.
The recursion for $H(T)$ can be proved in the following way. At the root level we need $w - 1$ comparisons to find the largest child, which is the root of subtree T_i with probability $t_i/(t-1)$ (note that the denominator is $t-1$ instead of t, since the root value t is already fixed). In this subtree we use $L_2(T_i)$ comparisons on average. Finally, key t returns to the root using one extra comparison. □

One should be aware of the fact that all these quantities depend on the shape of T. As a degenerate case we have lists: all father nodes in T have exactly one child. Then one can compute

$$L_1(T) = L_2(T) = \frac{1}{2t}(t-1)(t+2)$$

and

$$M_1(T) = M_2(T) = \frac{1}{4}(t-1)(t+4) - H_t + 1 = \frac{1}{4}t^2 + O(t),$$

where the harmonic numbers H_t are defined by $H_t = \sum_{i=1}^{t} 1/i$. This result is not surprising: for lists both algorithms boil down to insertion sort.
Note that in a similar way formulas can be given for the average number of data movements or swaps.

Now we define A_k and B_k for integer $k \geq 0$ by

$$A_k = \sum_{l=1}^{k} \frac{l}{2^l(2^l-1)} \quad \text{and} \quad B_k = \sum_{l=1}^{k} \frac{1}{2^l(2^l-1)},$$

so $A_0 = B_0 = 1$, $A_1 = B_1 = 1/2$, $A_2 = 2/3$ and $B_2 = 7/12$.
Let $A = \lim_{k\to\infty} A_k \approx 0.744$ and $B = \lim_{k\to\infty} B_k \approx 0.607$. In [2] and [9] the numbers α_1, α_2 and β are defined by:

$$\alpha_1 = \sum_{l=1}^{\infty} \frac{1}{2^l - 1}, \quad \alpha_2 = \sum_{l=1}^{\infty} \frac{1}{(2^l-1)^2}$$

and $\beta = B - \frac{1}{2}$. They can also be defined as certain basic hypergeometric series. Note that

$$\beta = \sum_{l=2}^{\infty} \frac{1}{2^l(2^l-1)} = \sum_{l=2}^{\infty} \frac{1}{2^l-1} - \sum_{l=2}^{\infty} \frac{1}{2^l} = \alpha_1 - \frac{3}{2}.$$

Furthermore $A = \alpha_1 + \alpha_2 - 2$, since

$$\alpha_1 + \alpha_2 = \sum_{l=1}^{\infty} \frac{2^l}{(2^l-1)^2} = \sum_{l=1}^{\infty} \frac{1/2^l}{(1-1/2^l)^2} = \sum_{l=1}^{\infty}\sum_{j=1}^{\infty} j(1/2^l)^j = \sum_{j=1}^{\infty} j \sum_{l=1}^{\infty}(1/2^j)^l$$

$$= \sum_{j=1}^{\infty} j\frac{1/2^j}{1-1/2^j} = \sum_{l=1}^{\infty} \frac{l}{2^l-1} = \sum_{l=1}^{\infty} \frac{l}{2^l(2^l-1)} + \sum_{l=1}^{\infty} \frac{l}{2^l} = A + 2,$$

where we used

$$\sum_{l=1}^{\infty} x^l = \frac{x}{1-x} \quad \text{and} \quad \sum_{l=1}^{\infty} l\,x^l = \frac{x}{(1-x)^2}$$

for real x with $-1 < x < 1$.

As a consequence of the proposition we have

Theorem. The construction of a heap on $n = 2^k - 1$ (integer $k > 0$) elements using the siftdown algorithm takes

$$(1 + 2A_k - B_k)(n+1) - 2\lg(n+1) - 1$$

comparisons in the average case. For the trickledown version it takes

$$(3 - A_k - B_k)(n+1) - \lg(n+1) - 3$$

comparisons in the average case. For triangular heaps the siftdown algorithm takes

$$(\frac{9}{8} + \frac{5}{2}A_k - \frac{3}{4}B_k)(n+1) - \frac{5}{2}\lg(n+1) - \frac{7}{4}$$

comparisons in the average case (if $n \geq 3$) and the trickledown version between

$$(\frac{77}{24} - \frac{3}{2}A_k)(n+1) - \lg(n+1) - \frac{25}{6}$$

and

$$(\frac{37}{12} - \frac{3}{2}A_k - \frac{9}{40}B_k)(n+1) - \lg(n+1) - \frac{503}{120}$$

comparisons (again for $n \geq 3$).

Proof. Let us first examine ordinary heaps, so all trees considered are complete binary trees. We write $L_j(n) = L_j(T)$ and $M_j(n) = M_j(T)$ $(j = 1, 2)$, where $n = 2^k - 1$ is the number of nodes in T. Using the proposition, we get

$$L_1(n) = 2 + \frac{n-1}{n} L_1(\frac{n-1}{2}), \quad L_1(1) = 0,$$

with solution

$$L_1(n) = \frac{2}{n} \lg(n+1)\ (n+1) - 3 - \frac{1}{n}.$$

We arrive at

$$M_1(n) = 2\, M_1(\frac{n-1}{2}) + \frac{2}{n} \lg(n+1)\ (n+1) - 3 - \frac{1}{n}, \quad M_1(1) = 0.$$

In case of the trickledown version we have to compute $H(T)$, but for a complete binary tree T of height h this is easy: $H(T) = 2(h-1)$. We are led to

$$L_2(n) = 1 + \frac{1}{n}(1 + 2(\lg(n+1) - 1)) + \frac{n-1}{n} L_2(\frac{n-1}{2}), \quad L_2(1) = 0,$$

with solution

$$L_2(n) = (\lg(n+1) + 1)(1 - \frac{1}{n}),$$

leading to

$$M_2(n) = 2\, M_2(\frac{n-1}{2}) + (\lg(n+1) + 1)(1 - \frac{1}{n}), \quad M_2(1) = 0.$$

Using the Lemma we get the solutions mentioned above.
These formulas support the intuition that most keys have to sink down to one of the lowest levels, thereby favouring the trickledown version: $L_1(n) \approx 2 \lg(n+1) - 3$, whereas $L_2(n) \approx \lg(n+1) + 1$.

For triangular heaps we have to do more. We apply the proposition to the stretched form of the tree. In the following we will examine $M_j(T)$, where T is now the stretched form of the original tree. We will also write $M_j(n)$ instead of $M_j(T)$, if T has n nodes; remember that in general the shape of the tree cannot be inferred from the number n, but this will be clear from the context. For the moment we drop the index j. In the following we shall derive different recurrences for $M(n)$ and $M(n-1)$ $(n = 2^k - 1)$, reflecting the fact that in triangular heaps left and right children play an asymmetric role. We shall always try to rephrase our recurrences in such a way that they are restricted to left children only: we eliminate the terms corresponding to right children. We have $M(n) = M(n-1) + L(n)$ and $M(n-1) = M(\frac{n-3}{2}) + M(\frac{n-1}{2}) + L(n-1)$ $(n \geq 3)$. Of course, $M(1) = L(1) = 0$. From these equations it follows that

$$M(n) = 2M\,(\frac{n-1}{2}) + (L(n) + L(n-1) - L(\frac{n-1}{2})).$$

For $L(n)$ we have

$$L(n) = G(n) + \frac{n-1}{n} L(n-1)$$

and

$$L(n-1) = G(n-1) + \frac{n-3}{2(n-1)} L(\frac{n-3}{2}) + \frac{1}{2} L(\frac{n-1}{2}).$$

From these equations it follows that

$$L(n) = G(n) + \frac{n-1}{n} \left\{ G(n-1) - \frac{1}{2} G(\frac{n-1}{2}) \right\} + \frac{n-1}{n} L(\frac{n-1}{2})$$

and

$$L(n-1) - L(\frac{n-1}{2}) = G(n-1) - \frac{1}{2} G(\frac{n-1}{2}).$$

Now we look at the triangular siftdown algorithm. For $L = L_1$ we have $G(n) = 1$ and $G(n-1) = 2$. Note that $G(2) = 1$. We find

$$L_1(n) = \frac{5n-3}{2n} + \frac{n-1}{n} L_1(\frac{n-1}{2}), \quad L_1(3) = 5/3.$$

Using the Lemma we get

$$L_1(n) = \frac{5 \lg(n+1)}{2n}(n+1) - \frac{19}{4} - \frac{3}{4n},$$

so (note the term $-\frac{13}{4}$):

$$M_1(n) = 2M_1(\frac{n-1}{2}) + \frac{5 \lg(n+1)}{2n}(n+1) - \frac{13}{4} - \frac{3}{4n}, \quad M_1(3) = 8/3,$$

and finally, again using the Lemma, we arrive at

$$M_1(n) = (\frac{9}{8} + \frac{5}{2} A_k - \frac{3}{4} B_k)(n+1) - \frac{5}{2} \lg(n+1) - \frac{7}{4}.$$

For $L = L_2$ we have to compute $H(T) = H(n)$, since here

$$G(n) = \frac{1}{n}(H(n)+1) \quad \text{and} \quad G(n-1) = 1 + \frac{1}{n-1}(H(n-1)+1).$$

The $H(n)$ satisfy the recurrence

$$H(n) = \frac{5}{2} + \frac{1}{2(n-2)} + H(\frac{n-1}{2})$$

(use $H(n) = 1 + H(n-1)$ and $H(n-1) = 2 + \frac{n-1}{2(n-2)} H(\frac{n-1}{2}) + \frac{n-3}{2(n-2)} H(\frac{n-3}{2})$), leading to

$$H(n) = \frac{5}{2} \lg(n+1) - 3 + \frac{1}{2} \sum_{i=0}^{k-3} 1/(2^{k-i} - 3)$$

(remember $n = 2^k - 1$), with $H(1) = 0$ and $H(3) = 2$. So we have

$$L_2(n) = \left\{ \frac{5\lg(n+1)}{2n} + 1 + \frac{1}{n}\left(\varepsilon_n + \frac{1}{2(n-2)} - \frac{3}{2}\right)\right\} + \frac{n-1}{n} L_2(\frac{n-1}{2}) \quad (n \geq 7),$$

where

$$\varepsilon_n = \frac{1}{2}\sum_{i=0}^{k-3} 1/(2^{k-i} - 3).$$

The boundary value is $L_2(3) = 5/3$. Now for $n \geq 7$

$$1/6 \leq \varepsilon_n + \frac{1}{2(n-2)} \leq 1/5.$$

This leads to a lower bound recurrence and an upper bound recurrence for $L_2(n)$:

$$L_2(n) = \left\{ \frac{5\lg(n+1)}{2n} + 1 - \frac{\gamma}{n}\right\} + \frac{n-1}{n} L_2(\frac{n-1}{2}) \quad (n \geq 7), \quad L_2(3) = 5/3,$$

where $\gamma = 4/3$ for the lower bound and $\gamma = 13/10$ for the upper bound, with solutions (again using the Lemma)

$$L_2^-(n) = \frac{\lg(n+1)}{n}(n - 3/2) + \frac{7}{6} - \frac{3}{2n}$$

as a lower bound and

$$L_2^+(n) = \frac{\lg(n+1)}{n}(n - 3/2) + \frac{47}{40} - \frac{61}{40n}$$

as an upper bound. Again we can see the positive effect of the trickledown operation: $L_1(n) \approx \frac{5}{2}\lg(n+1) - \frac{19}{4}$, whereas $L_2(n) \approx \lg(n+1) + 1.2$.
In a similar way we can handle M_2. We get

$$M_2(n) = 2\, M_2(\frac{n-1}{2}) + \left\{ L_2(n) + 1 + \frac{\frac{1}{2(n-2)} + 3/2}{n-1} \right\}$$

with boundary value $M_2(3) = 8/3$. Now we can use

$$\frac{3}{2n} \leq \frac{\frac{1}{2(n-2)} + 3/2}{n-1} \leq \frac{1}{60} + \frac{7}{4n} \quad (n \geq 7),$$

the bounds for L_2 and the Lemma to show that

$$(\frac{77}{24} - \frac{3}{2}A_k)(n+1) - \frac{25}{6} \leq M_2(n) + \lg(n+1) \leq (\frac{37}{12} - \frac{3}{2}A_k + \frac{9}{40}B_k)(n+1) - \frac{503}{120},$$

thereby finishing the proof. □

Note that, as $k \to \infty$,

$$1 + 2A_k - B_k \to 1 + 2A - B = \alpha_1 + 2\alpha_2 - 2 \approx 1.88$$

(Doberkat's result),

$$3 - A_k - B_k \to 3 - A - B = \frac{9}{2} - \alpha_1 - \alpha_2 - \beta \approx 1.65$$

(Wegener's result) and

$$\frac{9}{8} + \frac{5}{2}A_k - \frac{3}{4}B_k \to \frac{9}{8} + \frac{5}{2}A - \frac{3}{4}B \approx 2.53.$$

Thus it takes approximately $2.53\,n$ comparisons to construct a triangular heap on n nodes using the siftdown algorithm. For the trickledown version,

$$\frac{77}{24} - \frac{3}{2}A_k \to \frac{77}{24} - \frac{3}{2}A \approx 2.09,$$

whereas

$$\frac{37}{12} - \frac{3}{2}A_k + \frac{9}{40}B_k \to \frac{37}{12} - \frac{3}{2}A + \frac{9}{40}B \approx 2.10,$$

so it takes approximately $2.1\,n$ comparisons to construct a triangular heap using this algorithm. Again, formulas for the average number of data movements or swaps can be given in a similar way.

We conclude that with little extra work a much stronger structure can be accomplished.

5.2 The Sorting Phase

So let us now pay attention to the sorting phase. In [8] Schaffer and Sedgewick show that the average number of data movements (in fact, assignments between array elements) required to sort a random permutation of n distinct keys using ordinary heapsort is approximately $n \lg n$ (for large n). Our implementation of the ordinary heapsort uses swaps, but these can easily be replaced with data movements. Then, counting the number of swaps in our version is exactly the same as counting the number of data movements in the terminology of [8]. This justifies a direct comparison of our results with those of [8].

As is easily seen, the siftdown algorithm for triangular heaps never does more than $2n \lg n$ swaps. As usual we assume that all permutations are equally likely. We will show:

Theorem. The average number of swaps needed to sort a random permutation of n distinct keys (where $n = 2^k - 1$ for some integer $k > 0$) using the siftdown algorithm for triangular heaps is bounded from below by $(1/\lg \phi)\, n \lg n$, where $\phi = \frac{1}{2}(1 + \sqrt{5})$; $1/\lg \phi \approx 1.44$. This bound holds for sufficiently large n.

Proof. The proof proceeds analogous to that of Theorem 3 in [8], where so-called pull-down sequences are used; for triangular heaps they can be defined in a similar way. As we will see, it is also possible to argue without these sequences. We already know that the construction phase takes a linear number of swaps; indeed, in Section 5.1 we proved that the number of comparisons is at most $\frac{13}{4}n$, and the number of swaps is at most equal to the number of comparisons. So we may restrict our attention to the sorting phase.

The next observation is that in a triangular heap the number of keys that may have reached their position (during a fixed siftdown at the root) at the cost of l swaps is bounded from above by ϕ^l. (For "ordinary" heaps this upper bound is 2^l, the maximal number of keys at level $l+1$.) This follows from the fact that in a triangular heap with height larger than $l+1$ there are exactly $\mathrm{fib}(l)$ of those keys; here $\mathrm{fib}(l)$ denotes the lth Fibonacci number, defined by $\mathrm{fib}(l) = \mathrm{fib}(l-1) + \mathrm{fib}(l-2)$ for $l \geq 2$ and $\mathrm{fib}(0) = \mathrm{fib}(1) = 1$. It is well-known that $\mathrm{fib}(l) = \mathrm{Round}(\frac{\phi}{\sqrt{5}}\phi^l) \leq \phi^l$. Any possible value for l is bounded from above by $2\lfloor \lg n \rfloor$, being the length of the longest "search path" in a triangular heap with n nodes.

Now suppose that the sorting phase requires l_i swaps during the ith siftdown ($i = 1, 2, \ldots, n$). These numbers are uniquely determined by the key that is sifted down. Summing over all sequences $l_1, l_2, \ldots, l_n$ with $\sum_i l_i < M$ (this sum being the total number of swaps) we may conclude that there are at most $(2\lg n)^n \frac{1}{\phi-1}\phi^M$ triangular heaps that require fewer than M swaps to sort. In fact, from our observations it follows that there are at most $2\lg n\, \phi^{l_i}$ triangular heaps requiring exactly l_i swaps during the ith siftdown.

Now let $M = n(\log_\phi n - \log_\phi(2\lg n) - 4)$. We have seen that the number of triangular heaps that require fewer than M swaps to sort is bounded from above by $A(n) = \frac{1}{\phi-1}(n/\phi^4)^n$. Therefore the number of swaps required by the siftdown algorithm in the average case is bounded from below by

$$\frac{g(n) - A(n)}{g(n)} M = M - n\frac{A(n)}{g(n)}(\log_\phi n - \log_\phi(2\lg n) - 4).$$

In order to complete the proof it is sufficient to show that $g(n)$, the number of triangular heaps on n keys, is exponentially larger than $A(n)$. We have $g(n) = f(n)/2^{(n-1)/2}$, which is larger than $(n/4e\sqrt{2})^n$ (use the fact that $f(n) > (n/4e)^n$), and finally note that $4e\sqrt{2} < \phi^4$. $\square$

Experiments suggest that this lower bound is rather tight. Therefore, the number of swaps is somewhat larger than that for ordinary heapsort.

6 Discussion

In this paper we introduced the triangular heap and examined some of its properties. We have applied triangular heaps to the traditional problem of sorting an array in situ. Several interesting and efficient sorting algorithms are described.
The stronger structure of the triangular heap can be established and maintained with only little extra work. Good bounds for the average number of comparisons for the construction phase and for the average number of data movements for the sorting phase are derived.
Still many questions can be asked and remain to be investigated. It would be interesting, for instance, to find applications of heaps, where the stronger structure of the triangular heap is particularly useful. Conjectures like those in [9] can also be formulated for triangular heaps: experiments suggest that the average number of comparisons used by the trickledown algorithm for triangular heaps is approximately $n \lg n + 0.30\, n$, outbeating its ordinary heap counterpart (that uses less swaps). It would also be interesting to examine the expected height (or position) of a given number in a triangular heap (cf. [5]).

References

[1] S. Carlsson, *A Variant of Heapsort with Almost Optimal Number of Comparisons,* Information Processing Letters **24** (1987), 247–250.

[2] E.E. Doberkat, *An Average Case Analysis of Floyd's Algorithm to Construct Heaps,* Information and Control **61** (1984), 114–131.

[3] R.W. Floyd, *Algorithm 245: Treesort 3,* Communications of the ACM **7** (1964), 701.

[4] G.H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures,* Second edition, Addison-Wesley, 1991.

[5] J.M. de Graaf and W.A. Kosters, *Expected Heights in Heaps,* BIT **32** (1992), 570–579.

[6] D.E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching,* Addison-Wesley, 1973.

[7] C.J.H. McDiarmid and B.A. Reed, *Building Heaps Fast,* Journal of Algorithms **10** (1989), 352–365.

[8] R. Schaffer and R. Sedgewick, *The Analysis of Heapsort,* Journal of Algorithms **15** (1993), 76–100.

[9] I. Wegener, *Bottom-up-heap Sort, a New Variant of Heap Sort Beating on Average Quick Sort,* p. 516–522 in Proceedings Mathematical Foundations of Computer Science, LNCS **452**, Springer-Verlag, 1990.

[10] I. Wegener, *A Simple Modification of Xunrang and Yuzhang's HEAPSORT Variant Improving its Complexity Significantly,* The Computer Journal **36** (1993), 286–288.

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

Leiden, December 1994