Submit to 2002 International Computer Symposium (ICS2002), **Workshop on Algorithms and Computational Molecular Biology**, December 18-21, 2002, National Dong Hwa University, Hualien, Taiwan, R.O.C.

Heap Traversal and Its Applications

Abstract

The heap is a very fundamental data structure used in many computer software applications as well as in operating systems. In this paper, we propose a heap traversal algorithm which visits the nodes of a binary heap in ascending order of the key values stored in the nodes. We demonstrate two implementations of the heap traversal algorithm, using a binary search tree or a binary heap as the auxiliary structure that facilitates the traversal process. We also report the experimental results comparing the efficiency of heap-traversal based sorting algorithms vs. other well-known sorting algorithms. Our experimental results show that heap traversal can be applied to many traditional applications with improved performance.

Li-Jen Mao* and Sheau-Dong Lang**

*Department of Information Management, Fortune Institute of Technology

Chi-Shan, Taiwan, R.O.C. Email: larry.mao@msa.hinet.net Tel: 886-918-054-121

**School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816-2362, U.S.A.

Email: lang@cs.ucf.edu Tel:407-823-2474 Fax:407-823-5419

(Contact author: Li-Jen Mao*)

Keywords: Data Structures, Binary Search Tree, Heap Sort, Heap Traversal.

1. Introduction

A *binary heap* is a fundamental yet important data structure that provides an efficent implementation of priority queue operations [1, 2, 6, 10, 11]. A binary (min-)heap is also called a *heap ordered binary tree*; it is a method of storing a binary tree in an array where the binary tree maintains two properties: (1) the *heap property* in which the value stored in every node is smaller than the values of its two children (if exist); and (2) the *heap shape* property in which the binary tree must be a *complete tree*; that is, every level of the tree is complete except possibly for the last level which is filled in from left to right. Since a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes, this implies that the height of a complete binary tree is O(lg n). When a binary heap is stored in an array, the root of the tree has array index one (the element with index zero is usually not used), the children of the root are at indexes two (left child) and three (right child), etc. In general, the children (if exist) of the tree node with index k have indexes $2k$ (left child) and $2k+1$ (right child) and the parent (except for the root) is in position $\lfloor k/2 \rfloor$.

The basic operations of a (min-)heap are: *build-heap, heapify, insert, find-minimum,* and *delete-minimum*. Figure 1 gives a pseudo-code description of the *heapify* operation, which rearranges the heap to restore its heap property at the node with position *index*, assuming its two subtrees satisfy the heap property prior to the adjustment. This procedure takes O(lg n) time, where n is the number of nodes.

```
Procedure heapify(Hp[], index, n) {
/* n : number of nodes, Hp [1..n]: an array storing a heap of size n,
   index: index of a tree node where the heap property may be violated */
1.   pos   = 2 * index;
2.   item = Hp[index];
3.   not_done = true;
4.   while (pos < n and not_done) do
5.            if (pos < n – 1 and HP[pos] > Hp[pos+1])
6.                     pos++;
7.            if (item <= Hp[pos])
8.                     not_done = false;
9.            else   Hp[pos/2] = Hp[pos];
10.           pos = 2*pos;
11.   end while
12.   Hp[pos/2] = item;
13.  }
```

Figure 1: The Heapify procedure to restore the heap property at node *HP[index]*

The *build-heap* operation converts an input array to a heap by executing the *heapify* operation repeatedly from the last non-leaf node towards the root node. A pseudo-code description of *build-heap* is given in Figure 2. The procedure takes O(n) time [1, 6, 10]. It is also well known that the *insert* and *delete-minimum* operations take O(lg n) time in the worst case; *delete-minimum* takes O(1) time.

```
Procedure  build-heap (Hp [], n) {
/*   n : number of nodes, Hp[1..n]: an array to be converted into a heap */

1.  for   (i = n/2;   i > 0;   i--)
2.         heapify(HP[], i, n);
```

Figure 2: The Build-heap procedure which converts an array into a min-heap

There are many variants of the basic heap data structure reported in the literature [2, 6, 7]. Some implementations also support efficient merge operation (of two heaps), deletion of an arbitrary item, and increasing the priority of a item (the decrease-key operation). Although these variants may outperform the standard binary heap asymptotically, the binary heap has a very small constant in its O(lg n) time complexity and will often outperform other more complex heap implementations for typical cases [5, 10, 11, 12].

In this paper, we propose a novel algorithm which traverses a min-heap by visiting the nodes in ascending order of the values stored in the heap. The algorithm can be applied to any implementation of heaps although our discussions will be limited to the binary min-heap only. The heap traversal algorithm explores the heap property to facilitate the traversal process, and it produces the same output as that of the in-order traversal of a traditional binary search tree. However, because the heap property is *weaker* than that of a binary search tree, our heap traversal algorithm uses an auxiliary data structure for bookkeeping purposes. We will prove that the size of the auxiliary storage required is at most $\lceil n/2 \rceil$, and the time for locating the next node for traversal is O(lg n) in the worst case if the auxiliary structure is organized as a binary heap. Thus, the entire traversal process takes O(nlg n) time in the worst case. The main advantage of the proposed heap traversal technique is that locating the next node is fast due to the small size of the auxiliary data structure, which makes heap traversal a potentially useful subroutine for other priority-based applications [9]. However, this is accomplished at the expense of the additional space required.

The remainder of the paper is organized as follows. Section 2 describes the heap traversal algorithm and provides theoretical analysis of its space and time complexities. In Section 3, we will present experimental results comparing two heap-traversal based sorting algorithms with some traditional sorting algorithms (heapsort and two versions of Quicksort). Section 4 concludes the paper and points out some directions for further research.

2. Heap Traversal

Our heap traversal algorithm takes a (min-)heap as the input, and traverses the heap in ascending order of the values stored in the nodes without making any changes to the structure or contents of the heap. The nodes that are being traversed are copied into another list, producing an ordered listing of the values stored in the original heap. Thus, the heap traversal is similar to an in-order traversal of a binary search tree; the only difference is that since the heap structure does not provide as much order information as in a binary search tree, attempting to efficiently traverse a heap is a more challenging problem.

2.1 The Heap Traversal Algorithm

We now describe the general heap traversal algorithm **HeapTraversal** as shown in Figure 3. Suppose the input heap is stored in an array ***Hp***[1..N]. Let the output items be stored in the array ***OrderList***[1..N]. (We should comment that the results stored in ***OrderList***[1..N] are identical to the array ***Hp***[1..N] if heapsort is applied, due to the similarity between the traversal procedure and heapsort.) Our algorithm first initializes an auxiliary data structure ***AuxDS*** as empty. The structure ***AuxDS*** will be used to store the indexes of the elements of heap ***Hp***[] for selecting the next minimum element. The ***AuxDS*** can be implemented as a binary search tree or another binary heap. More details on the implementation of ***AuxDS*** will be given later.

```
Algorithm HeapTraversal(Hp[], AuxDS, OrderList[], N) {
/* traverse a heap Hp[1..N] using an auxiliary data structure AuxDS which can be a binary
  search tree, or another binary heap; the results of traversal are copied into
  OrderList[1..N] in order */

1.  ListIdx = 1; HpIdx = 1
2.  OrderList[ListIdx++] = Hp[HpIdx]   // we know the heap top is the smallest
3.  InsertAuxDS(max{Hp[2], Hp[3]});   // put larger item in AuxDS
4.  OrderList[ListIdx++] = min{Hp[2], Hp[3]}
5.  HpIdx = index(min{Hp[2], Hp[3]})
6.  repeat
7.      HpIdx = HpTraverse(Hp[], AuxDS, HpIdx, N) // visit an new item
8.      OrderList[ListIdx++] = Hp[HpIdx]              // copy in ordered list
9.  until (HpIdx == N) // until all items has been traversed
```

Figure 3: The heap traversal procedure

The algorithm **HeapTraversal** initializes the two index variables *ListIdx* (for ***OrderList***[]) and *HpIdx* (for ***Hp***[]) to 1, in Step 1 of Figure 3. The *ListIdx* will be increased each time a newly traversed element of ***Hp***[] (indexed by *HpIdx*) is copied to the ***OrderList***[]. After the variable initialization, since we already know that the smallest element is on top of the heap, it is copied to ***OrderList***[]. The larger value of the two children (i.e. ***Hp***[2] and ***Hp***[3]) of the root is copied into the auxiliary structure ***AuxDS*** by calling InsertAuxDS (Step 3); the smaller value (indexed by *HpIdx*) of the root is copied to ***OrderList***[]. The algorithm **HeapTraversal** then gets into a loop (Steps 6 to 9) which repeatedly calls the a subroutine **HpTraverse** using *HpIdx* as the index of the next item of ***Hp***[] for traversal. The loop continues until all nodes have been traversed and values being copied to ***OrderList***[]. It is easy to see that the number of iterations of this loop is exactly N – 2.

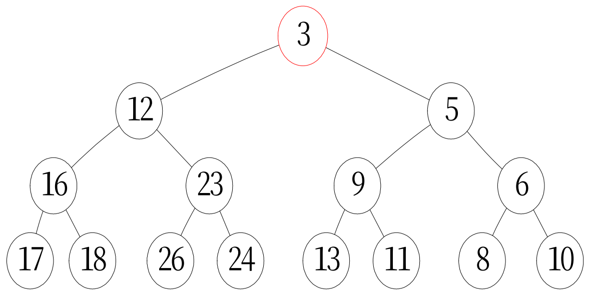
The subroutine **HpTraverse** (shown in Figure 4) is called by the algorithm **HeapTraversal** and will return the next smallest element that has traversed so far to the caller. **HpTraverse** first gets the index of last traversed element (i.e. argument ***Idx***) from its caller **HeapTraversal**, and computes the indexes of two child nodes of ***Idx*** in Step 1-2. Their values are compared to the smallest element of the auxiliary structure ***AuxDS***; the smallest of all these will be returned as the next element for traversal and be copied to ***OrderList***[]. The two indexes of two remaining elements will then be put into ***AuxDS*** by calling the procedure InsertAuxDS (Steps 4 and 9 of Figure 4). Note that there is a little difference between the ***if*** part (Steps 4-6) and the ***else*** part (Steps 7-11) of the algorithm **HpTraverse**: the ***if*** part involves only one call to the procedure InsertAuxDS; the ***else*** part involves two calls to the procedure InsertAuxDS, plus one extra call to the DeleteMin(***AuxDS***) (Step 8). Thus the ***if*** part will use rightly 2/3 of the execution time of the ***else*** part, this may be worth attention for further improvement.

```
Algorithm HpTraverse(Hp[], AuxDS, idx, N) {
/* traverse a heap Hp[1..N] using the auxiliary data structure AuxDS, the idx is the index of
   item last visited in heap Hp[], this procedure returns the index of next smallest item
   traversed so far */
1.   LeftChild = Hp[idx*2];
2.   RightChild = LeftChild + 1;
3.   if (min{Hp[LeftChild], Hp[RightChild]} < Hp[min(AuxDS)]) {
4.      InsertAuxDS(max{Hp[LeftChild], Hp[RightChild]}); // put larger
5.      return index of min{Hp[LeftChild], Hp[RightChild]} // smaller for process
6.   }
7.   else { /* the min item in the auxiliary data structure is smallest */
8.        MinIndex = DeleteMin(AuxDS) // remove smallest from AuxDS
9.        InsertAuxDS({Hp[LeftChild], Hp[Righthild]});    // put both in AuxDS
10.       return MinIndex;
11.   }
```

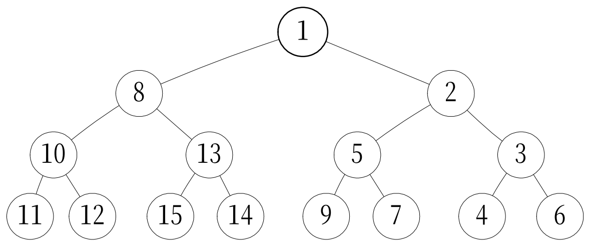
Figure 4: The HpTraverse subroutine

2.2 A Heap Traversal Example

Figure 5 uses an example to demonstrate the heap traversal procedure **HeapTraversal** of Figure 3. In Figure 5 (a), a binary min-heap of 15 nodes is given as the input, where the values of the nodes are shown in the figure. When applying the algorithm **HeapTraversal** to the heap, the order in which each node is visited during heap traversal is shown in Figure 5 (b), where a value *k* inside a node means that node is the *k*th value visited during traversal. Note the similarity between heap traversal and heapsort, the main difference is that heap traversal doesn't modify the input heap (Figure 5 (a)), and uses an auxiliary structure to maintain information of the potential locations for selecting the next smallest item for traversal. Also, heap traversal copies the output to another list without modifying the input heap, whereas heapsort does sorting in-place, i.e., without using auxiliary storage. Also note that an auxiliary structure for efficient heap traversal must keep some order information of its values; thus, a binary search tree or a binary heap would be good candidates for the auxiliary structure ***AuxDS***. We will report the experimental results based on these two implementations of the auxiliary structure in Section 3. Since the height of the auxiliary structure ***AuxDS*** (if implemented as a tree) tends to be lower than that of the original heap, the search and insertion time of the auxiliary structure is better than that based on the original heap. A complexity analysis is given in the next section.



(a) An input heap of 15 nodes, where the numbers are the values stored in the nodes.



(b) The heap traversal order of the above heap, where the numbers represent the sequencing order.

Figure 5: A Heap traversal example

2.3 Complexity Analysis of Heap Traversal

We first analyze the space requirement of the auxiliary structure ***AuxDS*** used in heap traversal. Notice that whenever the next smallest item is removed from ***AuxDS***, the children (if exist) are inserted into ***AuxDS*** (Step 4 or 9 of Figure 4). Thus, the size of ***AuxDS*** increases by one if the removed node has two children; otherwise, the size either remains the same (if there is one child) or will decrease by one (if there are no children). Since in an input heap of size *n* to the heap traversal algorithm, there are exactly $\lceil n/2 \rceil$ non-leaf nodes, so the size of the auxiliary structure can increase at most $\lceil n/2 \rceil$ times. Thus, we have proved the following result.

Theorem 1: The required size of the auxiliary structure used in the heap traversal algorithm is at most $\lceil n/2 \rceil$, where *n* is the size of the input heap.

To analyze the time complexity of the heap traversal algorithm, we will concentrate on the number of heap element comparison operations. (Thus, we will ignore data movement, index comparison, initialization operations in this analysis.) However, the result of comparison operations will provide a Big-O measure of the overall time complexity of the algorithm.

We assume a binary min-heap is used for the implementation of ***AuxDS***. According to Theorem 1, the size of this heap will increase from 0 to at most $\lceil n/2 \rceil$, then decrease eventually back to 0. If we apply the bottom-up version of the heap siftdown function [3, and 8, p.192], which essentially uses one comparison per tree-level during the InsertAuxDS operation (Steps 4 and 9 of Figure 4), the total number of heap element comparisons during heap traversal is bounded by the following summation:

$$2\sum_{k=1}^{n/2-1} \lg k \le 2\int_{1}^{n/2} (\lg e)\ln x\,dx = 2(\lg e)\left(\frac{n}{2}\ln\frac{n}{2} - \frac{n}{2}\right) = n\lg\frac{n}{2} - (\lg e)n = n\lg n - 2.443n$$

Theorem 2: If a binary heap is used to implement the auxiliary structure employing the bottom-up version of the heap siftdown function, the total number of heap element comparisons of the heap traversal algorithm is bounded by $n\lg n - 2.443\,n$. As a result, the time complexity of heap traversal is $O(n \lg n)$.

This result indicates that the number of element comparisons of the heap traversal algorithm is optimal, based on the $n\lg n$ lower bound for comparison-based sorting algorithms. However, the additional term $2.443\,n$ in the complexity indicates potential advantage comparing other results published in the literature. However, it should be noted that the additional storage overhead required by the heap traversal algorithm could make the algorithm unsuitable for some applications.

3. Experimental Results

In this section, we report the experimental results comparing sorting algorithms based on the heap traversal technique vs. other well-known sorting algorithms. In this study, we implemented two versions of the heap traversal algorithm: one uses a binary search tree and the other uses a binary heap as the auxiliary structure for heap traversal. However, the binary heap didn't use the bottom-up siftdown technique used in the analysis reported in the previous section. The corresponding sorting algorithms are named *heaptrav-bt* (using a binary search tree) and *heaptrav-hp* (using a binary heap). We also implemented two versions of *Quicksort* and the traditional heapsort [12], for our comparative study. The first version of *Quicksort* is a generic version without recursion; the second version of *Quicksort* uses the median-of-three partitioning and a cutoff of array size ten for recursive calls to optimize its performance [10, 11]. All these algorithms were implemented using the C language and were compiled using the BSD compatibility package C compiler (i.e. cc). The data in the input arrays were generated by the random number generator; the sizes of arrays were 300, 3000, and 30000. We ran the experiments using a Sun SPARC Server with 2048 megabytes of memory running SunOS 5.6, where each algorithm was run 3000 times using randomly generated numbers for each array size.

Table 1 shows the experimental results of the average execution times of 5000 runs for each array size. It can be seen that the optimized Quicksort performs the best. The algorithm *heaptrav-bt* outperforms Heapsort and outperforms the generic Quicksort, but is about 20% worse than the optimized Quicksort. The binary-heap based heap traversal sorting algorithm, *heaptrav-hp*, on the other hand, has a performance very close to that of Heapsort, using our current implementation of the binary heap for the auxiliary structure.

Table 1: Average execution time (5000 Runs) in microseconds (μs) of sorting algorithms with problem size range from 300 to 30000.

Sorting Algorithms / **Size**	Quicksort (Generic)	Quicksort (Optimized)	Heapsort	*heaptrav-bt*	*heaptrav-hp*
300	1127 μs	742 μs	1331 μs	933 μs	1388 μs
3000	14294 μs	9941 μs	17962 μs	11703 μs	17880 μs
30000	180181 μs	126946 μs	234942 μs	157534 μs	231502 μs

4. Conclusion

In this paper, we proposed a novel technique which traverses a heap in ascending order of the values stored in the heap. The heap traversal technique can be applied to any implementation of the heap although our studies were limited to binary heaps. The heap traversal technique can be used in applications that involve processing items in a priority-based order; for example, locating the *k*th smallest values in an array for multiple values of *k*, without sorting the array. Our theoretical analysis showed that a binary-heap based implementation of heap traversal uses an optimal number of element comparisons, but at the expense of additional storage required by heap traversal. We implemented two versions of heap sorting algorithm with this heap traversal approach, namely; *heaptrav-bt* (heapsort using the heap traversal technique with a binary search tree as the auxiliary structure) and *heaptrav-hp* (heapsort using the heap traversal technique with a binary heap as the auxiliary structure). These two sorting algorithms were compared with the well-known Quicksort and Heapsort algorithms. The experimental results showed that the new variation of heapsort, *heaptrav-bt*, could be competitive although was slower than the optimized Quicksort.

For further research, we plan to apply other auxiliary structures for heap traversal to improve its performance. We also plan to perform an average-case analysis of the heap traversal algorithm, to better understand the theoretical aspects of the algorithm. We also like to study other applications of heap traversal, and consider the possibilities of combining heap traversal with other sorting techniques such as heap-mergesort [4].

REFERENCES

[1] V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, New York, 1974.

[2] S. Baase and A.V. Gelder. *Computer Algorithms*, 3rd ed., Addison Wesley Longman, 2000.

[3] S. Carlsson. A variant of heapsort with almost optimal number of comparisons, *Inform*ation *Processing*. *Letters*, 24(4), 247-250, 1987.

[4] R. A. Chowdhury, S. K. Nath, and M. Kaykobad. The heap-mergesort, *Computers and Mathematics with Applications*, 39, 193-197, 2000.

[5] R. Cole. An optimally efficient selection algorithm, *Inform*ation *Processing*. *Letters*, 26, 295-299, 1988.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd ed., MIT Press, 2001.

[7] Paul F. Dietz and Rajeev Raman. Small-rank selection in parallel, with applications to heap construction, *Journal of Algorithms*, 30, 33-51 1999.

[8] R.W. Floyd. Algorithm 245 - Treesort3, *Communications of the ACM*, 7 (12), p.701, 1964.

[9] D. W. Jones. An empirical comparison of priority-queue and event-set implementations, *Communications of the ACM*, 29 (4), 300-311, 1986.

[10] D. E. Knuth. *The Art of Computer Programming*, Vol. III: *Sorting and Searching*, Addison-Wesley, Reading, MA. 1973.

[11] M. A. Weiss. *Data structures and algorithm analysis in C.* 2nd edition, Addison-Wesley, 1997.

[12] J. W. J. Williams. Algorithm 232 - Heapsort, *Communications of the ACM*, 7 (12), 347-348, 1964.