Short Note

A note on HEAPSORT

In *The Computer Journal* 33 (3), 1990, Xunrang and Yuzhang presented a new algorithm for HEAPSORT to reduce the number of comparisons from $2n \lg n$ comparisons to $\frac{3}{4}n \lg n$ in the worst case†. By recursing on their technique we show how the cost can be reduced to $n \lg n + n \lg\lg n$ comparisons. Combining this with another known technique that has the same complexity, we shall achieve the optimal solution for deleting the maximum element from a heap. Using this for sorting we get slightly less than $n \lg n + n \lg^* n$ comparisons.

Received October 1990, revised January 1991

1. Introduction

In 1964 Williams[9] introduced a sorting algorithm called HEAPSORT that sorted in $O(n \lg n)$ time using only a constant amount of extra storage. This is asymptotically optimal for the sorting problem if we have a comparison-based algorithm. In fact the minimum number of comparisons to sort n elements is $n \lg n + O(n)$, while Williams's algorithm took $3n \lg n$. Later in 1964, Floyd[6] reduced the cost for heap creation so that the HEAPSORT algorithm runs in $2n \lg n$ comparisons. This is also the version that can be found in most text books. In 1987 Carlsson[2] gave an algorithm that used only $n \lg n + n \lg\lg n$ comparisons. Gonnet and Munro[7] showed that the crucial delete-max operation can be performed in $\lg n + \lg^* n$ comparisons, which will also affect the time for HEAPSORT. $\lg^* n$ is the iterative logarithm defined as 0 if $n \leqslant 1$ and $1 + \lg^*(\lg n)$ otherwise.

In 1990 Xunrang and Yuzhang gave a method to reduce the cost from $2 \lg n$ to $\frac{3}{4} \lg n$[10]. This result is, of course, superseded by the results of Carlsson and of Gonnet and Munro. However, there are some ideas in their article that help to explain the complexity of HEAPSORT. In this note we will exploit the ideas of Xunrang and Yuzhang and combine them with another idea to achieve the optimal algorithm. This presentation starts from a conceptually simple idea, using comprehensive transformations, to achieve a fairly complicated algorithm. Hopefully, this will be easier for the reader to understand than the original papers.

2. Data Structure and Algorithms

The (max-) heap is a data structure that can be viewed as a binary tree where all levels are full, except maybe for the last one. In the last level the leaves are stored as far to the left as possible. The tree can be stored implicitly in an array, where a node at position k has its children at positions $2k$ and $2k+1$, and its parent at $\lfloor k/2 \rfloor$. The height of such a heap is $\lfloor \lg n \rfloor$. Furthermore, any node has a key value that is at least as big as its children and not bigger than its parent. This fact makes all paths from a leaf to the root into a sorted list in which we can perform a binary search. More details on heaps can be found in almost all introductory text books on data structures and algorithms cf.[1]

† In this paper $\lg n$ will denote $\log_2 n$.

A HEAPSORT algorithm starts by building a heap from the elements that have to be sorted. This is done by first considering the last node that has a child. We can regard this as a heap where the children of the root are heaps, but the root itself might violate the heap property. Restore the heap property of the heap and repeat this for all elements that have any children level by level. This heap creation will take linear time. After that, the maximum element, which is now at the root of the heap, is swapped with the last leaf and the heap property is restored in logarithmic time. The extraction of the maximum element is repeated until all elements are removed in decreasing order. They will now be stored in ascending order in the array. The crucial part of the sorting is the restoring of the heap after a deletion. This is done by first swapping the largest element (the root) with the last element in the array. The largest element is at this point not at the root of the tree, so we exchange the root with its largest child. Now the heap property might be violated one level further down in the heap, and we have to determine if the element should stay or be swapped with its largest child. This will in the worst case give two comparisons on each level of the heap, which gives a total of $2 \lg n$ comparisons.

In the algorithm of Xunrang and Yuzhang they used the observation that it is not necessary to do two comparisons on every level. Instead, the children of the root are compared, and the largest is moved up to the root. This hole is filled by its largest child recursively k times. Then we compare the last leaf with the latest promoted element. If it is larger, we perform an insertion in the heap above this hole by repeatedly swapping it with its smaller ancestors. If it is smaller, we use Williams's algorithm for rearranging the heap rooted at the hole on level k.

The cost for this is first k comparisons to reach level k. If the last leaf goes in the top part of the heap (above the hole), we have at most k more comparisons. If the last leaf goes in the bottom part of the heap the extra cost will be at most 2 on each level, and there are $\lg n - k$ levels. The worst-case cost for each restructuring is thus not more than the maximum of $2k$ and $k + 2(\lg n - k) = 2 \lg n - k$, which is minimized to $\frac{4}{3} \lg n$ when $k = \frac{2}{3} \lg n$. This is clearly better than the cost for using Williams's algorithm.

The cost can be further reduced by using something better than Williams's algorithm in the bottom part of the heap. Xunrang and Yuzhang have devised such an algorithm, as we showed above. This will give us a worst-case cost for the delete max of $\max(2k, k + \frac{4}{3}(\lg n - k))$. The minimum for this is $\frac{8}{7} \lg n$ for $k = \frac{4}{7} \lg n$. Again, we can apply this new algorithm, recursively, to get an even better bound. By the algorithm, where the same algorithm is used recursively if the last leaf is in the bottom part, a delete-max from a heap of height h will cost at most:

$$T(h) = \begin{cases} 2 & \text{if } h = 1 \\ \max(2k, k + T(h-k) + 1) & \text{otherwise} \end{cases}$$

If we select k to be $h/2$ the cost to insert an element in the top part of the heap will be less expensive than to restructure the bottom part.

```
procedure RESTORE(i, n: integer; x: element-
type);
var j, t, k: integer; stop: boolean;
begin
  j:= 2*i; t:= 0; stop:= false;
  Levels:= ⌊lg(n div i)⌋;
  k:= Levels div 2;
  while (j < n) and not stop do
  begin
    t:= t+1;
    if field[j+1].key > = field[j].key then
      j:= j+1;
    if t < = Levels-k then
    begin
      field[j div 2].key:= field[j].key;
      j:= 2*j;
    end
    else begin
      if field[j].key < = x then stop:= true
      else begin
        field[j div 2].key:= field[j].key;
        j:= 2*j;
        k:= k div 2;
      end;
    end;
  end;
  if (j = n) and not stop then
  field[j div 2].key:= field[j].key
  else
    j:= j div 2;
  while j > i do
    if x > field[j div 2].key then
    begin
      field[j].key:= field[j div 2].key;
      j:= j div 2
    end
    else
      i:= j;
  field[j].key:= x;
end; (***RESTORE***)
Procedure HEAPSORT(field: elementarray;
n: integer);
vari levels, height, bound: integer;
    temp: elementtype;
begin
  for i:= n div 2 downto 1 do
    RESTORE(i,n,field[i].key);
  for i:= n-1 downto 2 do
  begin
    temp:= field[i+1].key;
    field[i+1].key:= field[1].key;
    RESTORE(1,i,temp)
  end;
  temp:= field[1].key;
  field[1].key:= field[2].key;
  field[2].key:= temp;
end; (*HEAPSORT*)
```

Fig. 1. The new efficient HEAPSORT algorithm with a worst case of $n \lg n + n \lg\lg n$ comparisons. The elements to be sorted must be stored in an array at positions 1–n, where the array and n are given as parameters to HEAPSORT. In the RESTORE procedure $\lfloor \lg(n \text{ div } i) \rfloor$ has to be computed. The best way of doing this is machine-dependent, and is left to the implementer.

The cost is in that case given by:

$$T(h) = \begin{cases} 2 & \text{if } h = 1 \\ \frac{h}{2} + T(\frac{h}{2}) + 1 & \text{otherwise} \end{cases}$$

which is $h+\lg h+O(1)$. This gives a total cost for sorting of $n\lg n+n\lg\lg n+O(n)$, which is the same as for the algorithm by Carlsson.[2]

Carlsson, however, uses a different technique to achieve this bound. By observing that each path from a leaf to the root is sorted, a binary search can be used in such a path. The algorithm, described in the terminology above, is to let $k = h$ but use a binary search upwards to find the place to insert the last leaf. That is, we find this special path of maximum children all the way down to the leaves at a cost of h and then a binary search is performed with the last leaf, costing $\lg h$. This gives a total cost of $\lg n+\lg\lg n+O(1)$ for the operation.

When both of these ideas are combined, find the path k steps and perform a binary search upwards, to give a cost that is at most:

$$T(h) = \begin{cases} 2 & \text{if } h = 1 \\ \max(k+\lg k, \\ \quad k+T(h-k)+1) & \text{otherwise} \end{cases}$$

If we select $k = h-\lg h$ the cost for searching upwards will be at most h, and thus the cost will be given by:

$$T(h) = \begin{cases} 2 & \text{if } h = 1 \\ h-\lg h+T(\lg h)+1 & \text{otherwise} \end{cases}$$

which has a solution of $h+\lg^* h+O(1)$. This is exactly the algorithm given by Gonnet and Munro.[7] This algorithm can be slightly refined by balancing the costs for insertion and rebalacing,[4] but this will only affect the constant term. Gonnet and Munro also showed that this is the optimal cost for deleting the maximum element in a heap.

It is interesting to note that for the average case it is best to find the path of maximum children all the way down to a leaf, and then compare upwards using a linear search. For a random heap the average cost for a delete-max operation will be slightly less than $h+1.3$ comparisons, as shown by Carlsson.[3] When this strategy is used for sorting, Carlsson also showed results indicating an average number of comparisons, that is, only $n\lg n+0.4n$. Wegener showed that this algorithm has a worst case of at most $1.5n\lg n$ comparisons,[8] which has been proven tight by Fleischer.

As can be noted, the worst case for sorting can be less than the sum of the worst cases for all different sizes. This depends on the fact that not all delete-max operations can be of maximal cost.

3. Conclusion

In this paper we have taken the result of Xunrang and Yuzhang for HEAPSORT and improved on their ideas. It has yielded a new, and hopefully more comprehensive, way to describe the best algorithms already published for the delete-max operation in a heap, and also for sorting using repeated deletions from a heap. One of the intermediate algorithms had a worst-case complexity of $n\lg n+n\lg\lg n$ without using an explicit binary search. It has proved to be much faster on average than the best previously presented algorithm with the same worst case. They have been implemented in PASCAL on a SUN-3/80 and the new algorithm is approximately 2.5 times faster than the old worst-case algorithm and only 50% slower than the best average-case algorithm on the average (see Fig. 1).

S. CARLSSON
Department of Computer Science,
Luleå Technical University,
S-951 87 Luleå, Sweden

References

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, Reading, Mass. (1983).
2. S. Carlsson, A variant of heapsort with almost optimal number of comparisons. *Information Processing Letters* **24** (1987), 247–250.
3. S. Carlsson, Average-case results on heapsort. *BIT* **27** (1987), 2–17.
4. S. Carlsson, An optimal algorithm for deleting the root of a heap. *Information Processing Letters* **37**, 317–320 (1991).
5. R. Fleischer, A tight lower bound for the worst case of Bottom-Up Heapsort. *Proceedings 2nd International Symposium on Algorithms, Tapei, Taiwan*, Lecture Notes in Computer Science 557, Springer-Verlag, pp. 251–262 (1991).
6. R. W. Floyd, Algorithm 245 –Treesort 3. *Comm. ACM* **7** (12), 701 (1964).
7. G. H. Gonnet and J. I. Munro, Heaps on heaps. *SIAM Journal on Computers* **15** (4), 964–971 (1986).
8. I. Wegener, Bottom-up-Heap Sort, a new variant of Heap Sort beating on average Quick Sort. *Proceedings, Mathematical Foundations of Computer Science 1990, Banská Bystrica, Czechoslovakia* pp. 516–522 (1990).
9. J. W. J. Williams, Algorithm 232. *CACM* **7** (6) 347–348 (1964).
10. G. Xunrang and Z. Yuzhang, A new HEAPSORT algorithm and the analysis of its complexity. *The Computer Journal* **33** (3), 281–282 (1990).

Announcements

14–16 October 1992

ICDT 92, International conference on Database Theory, Berlin, Germany

ICDT 92 is the successor of two series of conferences on theoretical aspects of databases that were formed in parallel by different scientific communities in Europe. The first series, known as the International Conference on Database Theory, was initiated in Rome in 1986, and continued in Bruges (1988) and Paris (1990). The second series, known as the Symposium on Mathematical Fundamentals of Database Systems, was initiated in Dresden in 1987, and continued in Visegrad (1989) and Rostock (1991). The merger of these conferences should enhance the exchange of ideas and cooperation within a unified Europe and between Europe and the other continents. In the future, ICDT will be organised every two years, alternating with the more practically oriented series of conferences on Extending Database Technology (EDBT). ICDT 92 is organised by Fachausschuss 2.5 of the Gesellschaft für Informatik, in cooperation with EATCS and ACM.

Topics

- Data models and design theory
  Dependencies and constraints
  Incomplete information
  Deductive databases
  Complex objects
  Distributed and heterogeneous databases
  Active database systems
  Parallelism in databases
- Query languages
  Updates and transactions
  Database programming languages
  Concurrency control and recovery
  Complexity and optimisation
  Data structures and algorithms for databases
  Fundamentals of security and privacy

For further information contact:

Joachim Biskup, ICDT 92, Institute für Informatik, Universität Hildesheim, Samelsonplatz 1, D-W-3200 Hildesheim, Germany. Tel: +49-5121-883 730. Fax: +49-5121-860475. e_mail: biskup@infhil.uucp (mcsun!unido!-infhil!biskup)

or

Richard Hull, ICDT 92, Computer Science Department, University of Southern California, Los Angeles, CA 90089-0782, USA. Tel: +1-213-740-4523. Fax: +1-213-740-7285. e-mail:hull@cse.usc.edu.

As before, the proceedings of ICDT 92 will be published by Springer-Verlag, and will be available at the conference.

15, 16, 17 October 1992

European Studies Conference, Omaha, Nebraska

Plan to attend the 17th annual European Studies Conference, sponsored by the University of Nebraska at Omaha's European Studies Committee and College of Continuing Studies. ECS 92 will be an interdisciplinary meeting with sessions devoted to the scholarly exchange of information, research methodologies and pedagogical approaches.

For more information call the University of Nebraska at Omaha – European Studies Conference Program Coordinators: Professor Bernard Kolasa (402) 554-3617, Professor Patricia Kolasa (402) 554-3484; or write to: University of Nebraska at Omaha, College of Continuing Studies, UNO's Peter Kiewit Conference Center, 1313 Farnam, Omaha, Nebraska 68182-3061.