Information Processing Letters 26 (1987/88) 33–36
North-Holland

15 September 1987

THE DEAP—A DOUBLE-ENDED HEAP TO IMPLEMENT DOUBLE-ENDED PRIORITY QUEUES

Svante CARLSSON

Department of Computer Science, Lund University, Box 118, S-221 00 Lund, Sweden

Communicated by L. Kott
Received 4 June 1986
Revised 5 February 1987

This paper presents a symmetrical implicit double-ended priority queue implementation, which can be built in linear time. The smallest and the largest element can be found in constant time, and deleted in logarithmic time. This structure is an improvement of the MinMaxHeap presented by Atkinson et al. (1986).

Keywords: Priority queue, priority deque, double-ended priority queue, heap

1. Introduction

A priority queue is a data type where the element with the smallest key value can be found and deleted, and new elements can be inserted. It is also called a priority queue when the element with the largest key value is wanted, instead of the smallest.

In some cases, it is interesting to find and delete both the smallest and the largest element, as well as to be able to insert new elements. A structure with these operations may be called a double-ended priority queue, or priority deque for short.

A common way of implementing priority queues is through a heap—an implicit data structure first introduced by Williams [7]. Some attempts to implement double-ended priority queues with heaps or heap-like structures have been made. Some examples of this are the MinMaxHeap [1] and the structure with two heaps placed 'back-to-back' in a suitable way, proposed by Williams.

In this paper, a double-ended heap, called Deap, is presented. This structure is a mixture of the MinMaxHeap and the priority deque proposed by Williams. Also, the use of binary search of a path in a heap is used. This binary search technique is presented in [3,5].

The asymptotic results of all three implementations are the same. That is:

(1) finding the largest and the smallest element takes constant time,

(2) deleting the maximum or minimum element, or inserting a new element takes O(log n) time,

(3) the structure can be constructed in linear time, and

(4) no additional pointers are required.

2. The data structure

A deap is an implicit data structure with much resemblance to a heap. It can be regarded as a heap with a never referred element at the root. The left subtree of the root is a minheap, a heap with the smaller element at the root, and the right is a maxheap, a heap with the largest element at the root. Any leaf in the minheap is also smaller than the corresponding leaf in the maxheap, i.e., any node k in the minheap is smaller than the element at node k + a, if it exist, else (k + a) div 2,

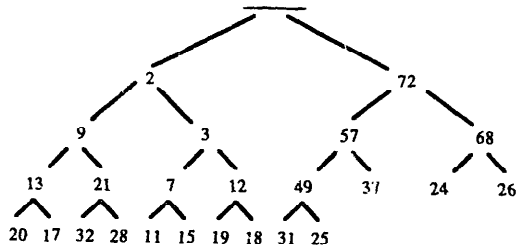

Fig. 1. Sample of a Deap.

where a is $2^{\lfloor \log k \rfloor - 1}$. Otherwise, the Deap has the same properties as the common heap. The Hasse diagram for a Deap is retrieved by turning the maxheap 'upside-down' and placing it back-to-back with the minheap. An example of a deap is shown in Fig. 1, and the corresponding Hasse diagram in Fig. 2.

As can be observed in the Hasse diagram, the Deap is a symmetrical structure, which makes it easy to transform so that the left subtree is a maxheap and the right is a minheap, without using any extra space.

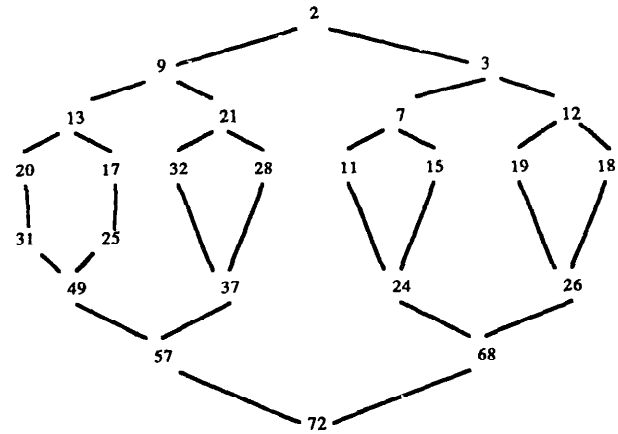

Fig. 2. Hasse diagram for Deap of Fig. 1.

Insert:
call the element to be inserted X
If the first free position is in the minheap **then**
compare X with the corresponding element in the maxheap (indexed I)
If X is the larger **then**
move the element in position I to the freed position
perform a binary search in the path from I to the root of the heap we are working on [3]
move all elements smaller than X one level down in the maxheap
store X in the freed position
If X is not the larger **then**
perform the binary search and data movements in the corresponding way in the minheap
If the first free element is in a maxheap **then**
proceed in a corresponding way as above

DeleteMin:
let X be the last element of the Deap
let I = 2
while element I is not a leaf **do**
replace element I with the smallest of its sons
let I be the index of the smallest son
perform an insert operation of element X in position I—the created hole

Creation:
for all positions—J—with any information about an element with a larger index, starting with the last **do**
If J is an index in the minheap **then**
do the same as for the DeleteMin operation but with X equal to element J
If J is an index in the maxheap **then**
do as in the DeleteMax operation

Fig. 3. Algorithms for the Deap operations.

3. Algorithms for the Deap operations

Since the minimum element in a Deap is stored in position two and the maximum in three it is easy to find them in constant time.

If an element is to be inserted in a place in the minheap, then it has to be compared with the corresponding leaf in the maxheap. If the new element is the larger, then the two have to change places, and the new element is inserted in the maxheap with the binary insertion technique described in [3,5], which is easy to do after observing that a node at position i has its ancestor k levels up at position $i \operatorname{div} 2^k$. If the new element is smaller than its corresponding leaf in the maxheap, it is inserted in the minheap by the same technique. If an element must be inserted in a place in the maxheap, a similar algorithm is used.

If the minimum element has to be deleted, we have to replace it with its smallest son. This son has to be replaced by its smallest son and so on, until we reach a leaf. Then, the element stored in the last place of the Deap is inserted into the empty space. The maximum element is deleted in a similar way.

If a Deap has to be constructed, we do it in the same way as we construct a heap, by repeated delete operations but instead of inserting the element in the last place we insert the root of the subDeap that we are currently working on.

For further implementation details, see Fig. 3 or [2].

4. Complexity analysis

An insertion uses $\lfloor \log \lfloor \log n \operatorname{div} i \rfloor \rfloor + 1$ comparisons, when we do a binary search in a path from element i to element n. A deletion uses $\lfloor \log(\text{size} + 1) \operatorname{div} i \rfloor$ comparisons plus the number of comparisons by the insertion. This gives us the total number of comparisons per operation of

Insert: $1 + \lfloor \log \lfloor \log(n \operatorname{div} 2) \rfloor \rfloor + 1 \leqslant \log \log n + 2,$

DeleteMin: $\lfloor \log((n+1) \operatorname{div} 2) \rfloor + \lfloor \log \lfloor \log((n+1) \operatorname{div} 2) \rfloor \rfloor + 1 \leqslant \log n + \log \log n,$

DeleteMax: $\lfloor \log((n+1) \operatorname{div} 3) \rfloor + \lfloor \log \lfloor \log((n+1) \operatorname{div} 2) \rfloor \rfloor + 1 \leqslant \log n + \log \log n,$

MakeDeap: $\sum_{i=1}^{3n/4} (\lfloor \log((n+1) \operatorname{div} i) \rfloor + \lfloor \log \lfloor \log((n+1) \operatorname{div} i) \rfloor + 1 \rfloor + 1) \leqslant 2.07 \ldots n.$

These results are to be compared with those for the MinMapHeap, which are

Insert: $\log(\log(n+1)),$

DeleteMin: $\frac{3}{2} \log n + \log \log n,$

DeleteMax: $\frac{3}{2} \log n + \log \log n,$

Create: $2.15 \ldots n.$

5. Concluding remarks

Algorithms and data structures for double-ended priority queues have been presented earlier, but none of them had the symmetry as well as the simple storage requirement of the Deap. In addition to this, it uses fewer comparisons than any other data structure for this purpose. The symmetry should also simplify the merging of priority deques, even though it must be further investigated.

Acknowledgment

I would like to thank Christian Collberg, Sten Henriksson, Rolf Karlsson together with the rest of the group for algorithm theory in Lund for their help in preparing this paper.

References

[1] M.D. Atkinson, J.-R. Sack, N. Santoro and Th. Strothotte, Min-Max heaps and generalized priority queues, Comm. ACM 29 (10) (1986) 996–1000.

[2] S. Carlsson, Deap—a double-ended heap to implement double-ended priority queues, Tech. Rept. CODEN: LUNFD6/(NFCS-3011)/(1–7), 1986.

[3] S. Carlsson, A variant of HEAPSORT with almost optimal number of comparisons, Inform. Process. Lett. 24 (4) (1987) 247–250.

[4] R.W. Floyd, Algorithm 245—Treesort3, Comm. ACM 7 (12) (1964) 701.

[5] G.H. Gonnet and J.I. Munro, Heaps on heaps, in: Proc. ICALP, Aarhus (1982) 282–291; also: SIAM J. Comput. 15 (4) (1986) 964–971.

[6] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973).

[7] J.W.J. Williams, Algorithm 232, Comm. ACM 7 (6) (1964) 347–348.