International Journal of Computer Mathematics
Vol. 80, No. 9, September 2003, pp. 1121–1129

IMPROVEMENTS IN DOUBLE ENDED PRIORITY QUEUES*

M. ZIAUR RAHMAN[a,†], REZAUL ALAM CHOWDHURY[b] and M. KAYKOBAD[c]

[a]*Department of Computer Science and Engineering, Ahsanullah University of Science and Technology, Dhaka, Bangladesh;*
[b]*Department of Computer Science, University of Texas at Austin, USA;*
[c]*Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh*

(Received 4 March 2002)

In this paper, we present improved algorithms for min–max pair heaps introduced by S. Olariu *et al.* (A Mergeable Double-ended Priority Queue – *The Comp. J.* 34, 423–427, 1991). We also show that in the worst case, this structure, though slightly costlier to create, is better than min–max heaps of Strothotte (Min–max Heaps and Generalized Priority Queues – *CACM*, 29(10), 996–1000, Oct, 1986) in respect of deletion, and is equally good for insertion when an improved technique using binary search is applied. Experimental results show that, in the average case, with the exception of creation phase data movement, our algorithm outperforms min–max heap of Strothotte in all other aspects.

Keywords: Algorithm; Heap; Min–Max pair heap; Min–Max heap; Priority queues

C.R. Categories: D.4.8, E.2. H.3.3

1 INTRODUCTION

A (single-ended) priority queue is a “largest in, first out” list. It is used to perform some or all of the following operations.

- Create the structure (*Create*(*Queue*))
- Find the Maximum (*FindMax*)
- Delete the Maximum (*DeleteMax*)
- Add a new element (*Insert*(*x*))
- Merge two queues (*Merge*(*Queue1*, *Queue2*))

“Smallest in, first out” priority queues can be defined analogously. These are called priority queues since the key to each item reflects its relative ability to get out from the list quickly.

* An earlier version of this paper has been published in the proceedings of *International Conference on Computer and Information Technology, 1999*.
† Corresponding author. E-mail: bapy_bd@yahoo.com

ISSN 0020-7160 print; ISSN 1029-0265 online © 2003 Taylor & Francis Ltd
DOI: 10.1080/207160310001599079

Several data structures exist for priority queue implementation. A sorted list is an obvious choice, But it requires $O(n)$ insertion cost. Though for small n (<20), sorted list is acceptable [3], for larger values of n a more efficient data structure is required.

Max-heap is one such efficient data structure. It is a complete binary tree having the property that root element is the maximum and its left and right sub heaps are also max-heaps. A max-heap of size n can be constructed in linear time, and can be stored in an n-element array. Hence it is referred to as an implicit data structure. Using max-heap *FindMax* can be performed in constant time and both *DeleteMax* and *Insert*(x) in logarithmic time. A double-ended priority queue is similar with the exception that both maximum and minimum can be sought. Implementation of double ended priority queues using max-heap requires linear time for *FindMin* (Find the Minimum element) operation.

A more efficient algorithm [12] was devised using a min-heap back-to-back with the max-heap. This method leads to constant time find and logarithmic time insertion and deletion operations but requires double ($2n$) space and is somewhat trickier to implement.

The MinMax Heap structure overcomes these limitations. A min–max heap is based on heap structure under the notion of min-max ordering: values stored at nodes on even (odd) levels are smaller than or equal to (respectively, greater than) values stored at their descendants. This structure can be constructed in linear time. *FindMin*, *FindMax* operations are performed in constant time and *Insert*(x), *DeleteMin* and *DeleteMax* in logarithmic time using this structure. Also sub-linear merging algorithm is given with relaxation of strict ordering [5].

The min–max pair heap was introduced by Olariu *et al.* [9]. It has the benefit that this double-ended priority queue supports merging in sub-linear time. The algorithms for the above data structure is improved in our paper.

2 MIN–MAX PAIR HEAPS

DEFINITION *A min–max pair heap is a binary tree H featuring the heap-shape property, such that every node in $H[i]$ has two fields, called the min field and the max field, and such that H has a min–max ordering: for every $i(1 \leq i \leq n)$, the value stored in the min field of $H[i]$ is the smallest of all values in the sub-tree of H rooted at $H[i]$; similarly the value stored in the max field of $H[i]$ is the largest key stored in the sub-tree of H rooted at $H[i]$.*

However we can consider those two heaps separalely by taking min (max) elements. We name the min heap as A and max heap as B. Then we can show their relationship by a Hasse diagram.

For example a min–max pair heap is shown in Figure 1. Its corresponding Hasse diagram is shown in Figure 2.

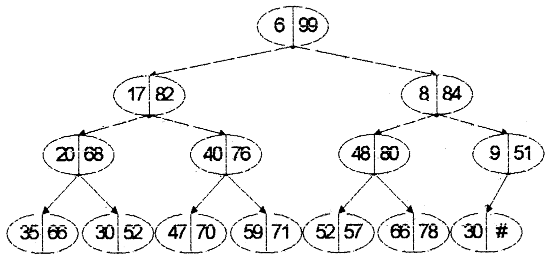


FIGURE 1 A sample min–max pair heap.

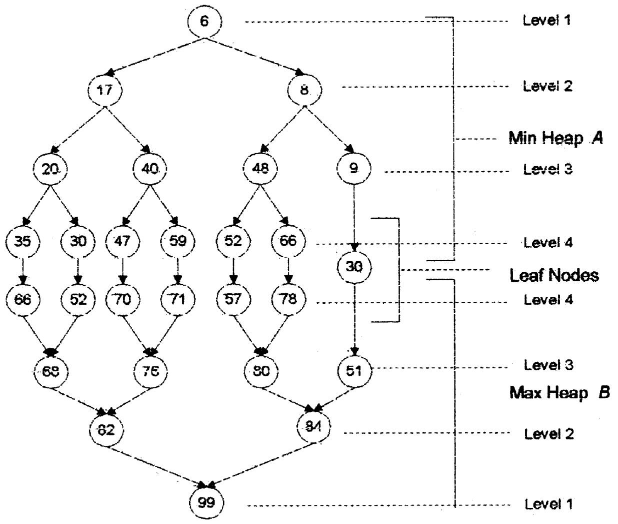


FIGURE 2 The Hasse diagram of min–max pair heap of Figure 1.

3 ALGORITHMS

The creation phase of min–max pair heap is similar to general heap in the respect that it also proceeds from leaf to root. However, in contrast to general heap, here we cannot regard the leaf nodes as already min–max pair heap and so adjustment is also applied to leaf nodes. This is illustrated using the Hasse diagram.

Starting from level h in Hasse diagram (h is the height of the min–max pair heap) we proceed towards level 1, adjusting min heap element followed by the adjustment of max heap element. This is illustrated elaborately using an example of adjustment in level 2.

All the elements of level 3 are heapified before the adjustment of level 2 elements. Now to adjust a level 2 min heap element we obtain chain of younger sons and its extension to max heap ranging from a level 3 element in min heap to level 3 element in max heap. We insert the level 2 min heap element at proper place in this chain. When we are done, we adjust corresponding level 2 max heap element. In this case we similarly obtain a chain of elder sons and its extension to min heap ranging from level 3 element in max heap to level 2 element in min heap. We insert the level 2 max-heap element in this chain. This is repeated for all level 2 elements.

The exact routines are as follows:

```
procedure Create_Min_Max_Pair_Heap
  //Adjust the elements in $H$ (1:$n$) to form a min–max pair heap//
  for $i \leftarrow n/2 + 1$ to $n$ do
    if $H[i].min > H[i].max$
        $H[i].min \leftrightarrow H[i].max$
    end if
  end for
  for $i \leftarrow n/2$ downto 1 do
      Siftdown($H[i]$)
  end for
End //Create_Min_Max_Pair_Heap
```

```
procedure Siftdown(node $H[i]$)
        Trickle_Down_Min_Field($H[i]$)
        Trickle_Down_Max_Field($H[i]$.)
End //Siftdown

procedure Trickle_Down_Min_Field(node $H[i]$)
 $b \leftarrow true$
 $k \leftarrow i$
 $p \leftarrow H[i]$
 while ($2 \times k \leq n$)          //There is more child to consider
    $k \leftarrow 2 \times k$              //Select the minimum child node
    if($K + 1 \leq n$ and $H[k].min > H[k+1].min$)
         $k \leftarrow k + 1$
    endif
    if($b = true$)               //This is used to save the first child of $H[i]$ upto which
      $b \leftarrow false$                adjustment will be required
      $f \leftarrow k$
    end if
 end while
 if($p.min < H[k].min$)
         Check_Up_Min($p, i, k/2$)
 else
         Move_Up_Min($i, k$)
         $j \leftarrow k$
         if($H[k].max = \#$)          //The max filed may be absent wheen $n$ is odd
              $k \leftarrow k/2$
         end if
         if($k = i$)
              return
         end if
         if($p.min \leq H[k].max$)
              $H[j].min \leftarrow p.min$
              return
         else
              $H[j].min \leftarrow H[k].max$
              $H[k].max \leftarrow p.min$
              Bubble_UP_Max($f, k$)
         end if
 end if
end //Trickle_Down_Min_Field
```

Trickle_Down_Max_Field routine is almost similar to the *Trickle_Down_Min_Field*. There are exceptions due to two reasons. The first reason is, it needs to adjust up to min field of $H[i]$ and second there are some variation in the handling of the null max field whenever odd element is handled.

Procedure *Check_Up_Min*(p, *start_index*, *end_index*) finds proper place for p in the chain ranging from *start_index* to *end_index* and place p in that position. Procedure *Move_Up_Min*(*start_index*, *end_index*) moves all elements on the chain ranging from *start_index* to *end_index* one place up in the chain. Procedure *Bubble_Up_Min*(*start_index*, *end_index*) adjusts H[*end_index*].*min* bottom up at the proper position in the chain ranging from *start_index* to *end_index*.

The *Check_Up_Max*, *Move_Up_Max* and *Bubble_Up_Max* are identical to their Min partners.

The *Bottom_up_Min*(*start_index*, *end_index*) (resp. *Bottom_up_Max*(*start_index*, *end_index*)) routine works in the same way. But this routine is used when a transition occurs from min-heap to max-heap or vice-versa. In this case $H[end\_index].Min$ ($H[end\_index].Min$) is the element for which we find a proper place in the chain ranging from *start_index* to *end_index*. Then it is inserted at that place.

If the new element goes to the other heap, we use *Move Up Min*(*start_index*, *end_index*) (resp. *Move_Up_Max*(*start_index*, *end_index*)) routine. This routine moves all elements on the chain ranging from *start_index* to *end_index* one place up in the chain.

The Insertion operation is of logarithmic complexity. We copy the element at the last position $H[n]$. We refer to Hasse diagram for easier understanding of insertion operation. Since insertion occurs in a leaf node element in the Hasse diagram, we compare it with adjacent parent element in min heap. If it is smaller we insert it bottom up to the min heap. Otherwise, we compare it with adjacent parent element on max heap and insert it bottom up to max heap if required.

The Deletion from min-max pair heap is easy, just pick the minimum-$H[1].min$ (resp. $H[1].max$) and copy last element to $H[1].min$ ($H[1].max$). Then perform *Trickle_Down_Min_Fieid* (*Trickle_Down_Max_Field*) for reheapification.

4 WORST CASE COMPLEXITY ANALYSIS

We will calculate the number of comparisons required in the worst case by the min–max pair heap creation algorithm described above.

Let there be n elements in the tree. If h is the height of the min–max pair heap then $2(2^h - 1) = n$. *i.e.* $h = \lg(n+2) - 1$.

The worst case number of comparisons required by the heap creation algorithm is

$$\begin{aligned}
&= \sum_{i=1}^{h-1} 2^{i-1}[2(h-i) + 1 + 2(h-i) + 2] + 2^{h-i} \\
&= 4h\sum_{i=1}^{h-1} 2^{i-1} - 2\sum_{i=1}^{h-1} i2^i + 3\sum_{i=1}^{h-1} 2^{i-1} + 2^{h-1} \\
&= 4h(2^{h-1} - 1) - 2[(h-2)2^h + 2] + 3(2^{h-1} - 1) + 2^{h-1} \\
&= 3.2^{h+1} - 4h - 7 \\
&= 3(n+2) - 4\lg(n+2) - 3 \\
&= 3n - 4\lg(n+2) + 3
\end{aligned}$$

However, if we calculate the worst case complexity for heap creation by inserting elements into the corresponding chain of younger (elder) sons using binary search, similar to Gonnet and Munro [7], the number of comparisons becomes

$$\begin{aligned}
&= \sum_{i=1}^{h} 2^{i-1}[(2(h-i) + \lg(2(h-i)+1) + \lg(2(h-i)+2)] \\
&= 2h\sum_{i=1}^{h} 2^{i-1} - 2\sum_{i=1}^{h} i2^{i-1} + \sum_{i=1}^{h} 2^{i-1}[\lg(2(h-i)+1) + \lg(2(h-i)+2)]
\end{aligned}$$

$$
\begin{aligned}
&= 2h(2^h - 1) - 2[(h-1)2^h + 1] + 1.566421n \\
&= n - 2\lg(n+2) - 1 + 1.566421n \\
&= 2.566\ldots n - 2\lg(n+2) - 1
\end{aligned}
$$

In the heap creation phase, worst case number of movements is as follows

$$
\begin{aligned}
&= \sum_{i=1}^{h-1} 2^{i-1}[2(h-i) + 2 + 2(h-i) + 3] + 3.2^{h-1} \\
&= 4h\sum_{i=1}^{h-1} 2^{i-1} - 2\sum_{i=1}^{h-1} i2^i + 5\sum_{i=1}^{h-1} 2^{i-1} + 3.2^{h-1} \\
&= 4h(2^{h-1} - 1) - 2[(h-2)2^h + 2] + 5(2^{h-1} - 1) + 3.2^{h-1} \\
&= 4.2^{h+1} - 4h - 9 \\
&= 4(n+2) - 4\lg(n+2) - 1 \\
&= 4n - 4\lg(n+2) + 3
\end{aligned}
$$

5 COMPARISONS BETWEEN MIN-HEAPS, MIN–MAX HEAPS AND MIN–MAX PAIR-HEAPS

The worst-case complexities of improved algorithms for min-heaps, min–max heaps and min–max pair heaps are shown in Table I.

In Table I the function $g(x)$ is defined as follows: $g(x) = 0$ for $x \leq 1$ and $g(n) = g(\lceil \lg(n) \rceil) + 1$.

6 EXPERIMENTAL RESULTS

Experimental results on the average number of comparisons and data movements in the heap creation phase for the two algorithms are plotted in Figure 3.

From the above chart we observe that on the average comparison cost of min–max pair heap is slightly lower than min–max heap. The result is justified since we have used bottom up algorithms, which requires 1.299 comparisons on the average case [4]. Also we observe that data movement cost is nearly $2.94\,n$ for min–max pair heap – which is far better than worst case $4\,n$.

The insertion and deletion costs are shown in Figure 4. We perform x number of insertions in a heap of size x and perform x deletions from a heap of size $2x$. From the above charts we observe that, though in the worst case number of comparison required is twice in min–max pair heap than min–max heap on average it is about 1.025 times and data movement is seems to be 0.91 times as n increases. And if we consider the combined cost we found that the performance is better in our algorithm than min–max heap.

In case of deletion, we observe that our algorithm shows further improvement in average case. Min–max pair heap requires less than half of the comparisons than that of min–max heap and almost same number of data movement.

TABLE I Worst-Case Complexities of Improved Algorithms for Min-heaps, Min–Max Heaps and Min–Max Pair Heaps.

	Data movement cost			*Comparison cost*		
	Min heaps	*Min–Max heaps*	*Min–Max pair heaps*	*Min-heaps*	*Min–Max heaps*	*Min–Max pair heaps*
Create	N	n	$4n$	$1.625n$	$2.15\ldots n$	$2.566\ldots n$
Insert	$\lg(n+1)$	$0.5\lg(n+1)$	$\lg(n+2)$	$\lg(\lg(n+1))$	$\lg(\lg(n+1))$	$\lg(\lg(n+2))$
DeleteMin	$\lg(n)$	$\lg(n)$	$2\lg(n+2)$	$\lg(n)+g(n)$	$1.5\lg(n) + \lg(\lg(n))$	$\lg(n+2)+\lg(\lg(n+2))$
DeleteMax	$\lg(n)$	$\lg(n)$	$2\lg(n+2)$	$0.5n+\lg(\lg(n))$	$1.5\lg(n)+\lg(\lg(n))$	$\lg(n+2)+\lg(\lg(n+2))$

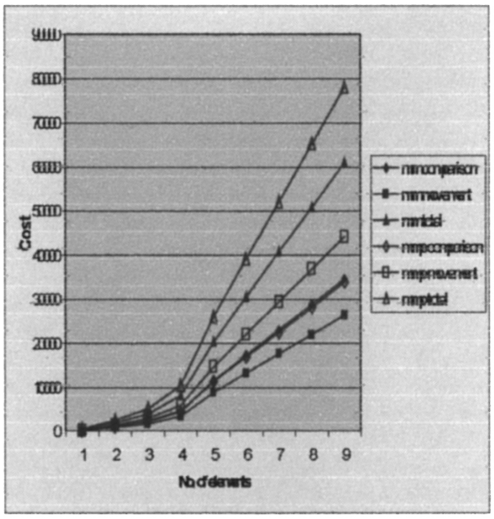
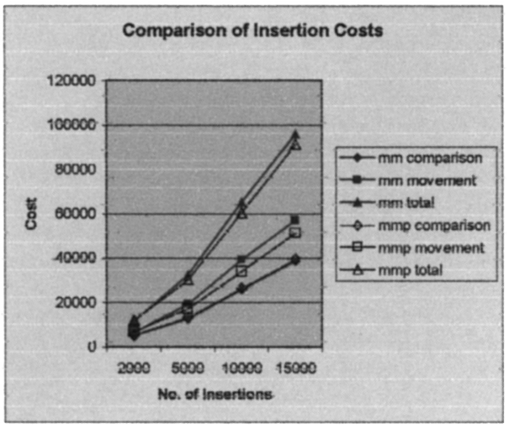


FIGURE 3 Comparison of creation cost (mm = min–max, mmp = min–max pair).



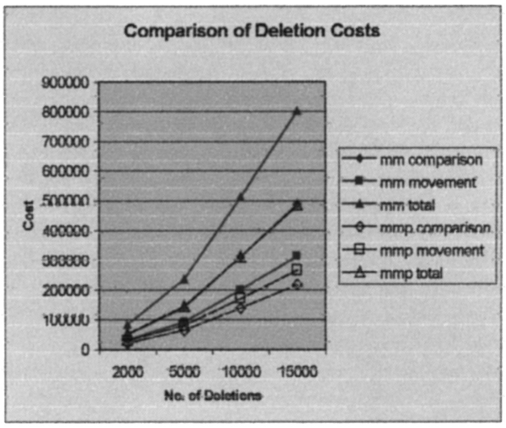


FIGURE 4 Insertion and deletion costs (mm = min–max, mmp = min–max pair).

7 CONCLUDING REMARKS

We have presented efficient algorithms for the implementation of implicit double-ended priority queues using min–max pair-heap. The deletion cost for the above structure is a significant improvement over min–max heap. All other costs for double-ended priority queues are comparable to that of a min–max heap. We have shown that the min–max pair heap structure is very much similar to a conventional max-heap – there are just two heaps in it. This similarity opens up possibilities for applying known applications and optimizations of max-heap to double-ended priority queues. For example, the concept of fine heap can be introduced here for further optimization of the above algorithm. The heap merging algorithms [12] have been revised [9] to merge min–max pair heaps. However, using our efficient algorithms the complexity coefficient will improve.

Acknowledgement

We would like to thank M. Manzur Murshed of the Australian National University for his generous support.

References

[1] Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA.
[2] Atkinson, M. D., Scak, J.-R., Santoro, N. and Strothotte, Th. (1986). Min–max heaps and generalized priority queues. Programming techniques and data structures. *Comm. ACM*, **29**(10), 996–1000.
[3] Brown, M. R. (1980). The analysis of a practical and nearly optimal priority queue. Garland Publishing, New York.
[4] Carlsson, S. (1987). Average-case results on heapsort. *BIT*, **27**, 2–17.
[5] Ding, Yuzheng, Weiss and Mark Allen. (1993). The relaxed min–max heap – A mergeable double-ended priority queue. *Acta Informatica*, **30**, 215–231.
[6] Gonnet, G. H. (1984). *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, MA.
[7] Gonnet, G. H. and Munro, J. I. (1982). Heaps on heaps. In: *Proceedings of the ICALP*, Aarhus, 9, July, pp. 282–291.
[8] Knuth, D. E. (1973). *The Art of Computer Programming, Vol III: Sorting and Searching.* Addison-Wesley, Reading MA.
[9] Olariu, S., Overstreet, C. M. and Wen, Z. (1991). A mergeable double-ended priority queue. *Computer Journal*, **34**, 423–427.
[10] Sack, J.-R. and Strothotte, Th. (1985). An algorithm for merging heaps. *Acta Informatica*, **22**, 171–186.
[11] Strothotte, Th. and Sack, J.-R. (1985). Heaps in heaps. *Congressus Numerantium*, **49**, 223–235.
[12] Williams, J. W. J. (1964). Algorithm 232. *CACM*, **7**(6), 347–348.