Merging Algorithm for Relaxed Min-Max Heaps

Relaxed min-max 힙에 대한 병합 알고리즘

Yong Sik Min*
민 용 식*

ABSTRACT

This paper presents a data structure that implements a mergeable double-ended priority queue : namely, an improved relaxed min-max-pair heap. It suggests a sequential algorithm to merge priority queues organized in two relaxed min-max heaps : kheap and nheap of sizes k and n, respectively. This new data structure eliminates the blossomed tree and the lazying method used to merge the relaxed min-max heaps in [8]. As a result, the suggested method in this paper requires the time complexity of O(log(log(n/k))*log(k)) and the space complexity of O(n+k), assuming that $k \leq \lfloor \log(\text{size}(\text{nheap})) \rfloor$ are in two heaps of different sizes.

요 약

본 논문은 relaxed min-max heap을 병합시키기 위하여 이용된 새로운 자료구조인 개선된 relaxed min-max-pair 힙으로서, 두개의 relaxed min-max 힙 즉, 크기가 n인 relaxed min-max nheap과 크기가 k인 relaxed min-max kheap으로 구성된 우선 순위 큐를 병합시키기 위한 순차적 알고리즘을 제시하고자 한다. 본 논문에서 제시된 방법은 [8]에 제시된 방법에서 relaxed min-max 힙을 병합 시키기 위해서 이용된 blossomed tree와 lazying 방법을 제거하여도 병합이 되는 새로운 기법을 제시하였다. 결과적으로 본 논문에서 제시된 방법은 두개의 relaxed min-max 힙의 크기가 서로 다른 경우로서, 이때 크기 $k \leq \lfloor \log(\text{size}(\text{nheap})) \rfloor$인 경우, 시간 복잡도가 O(log(log(n/k))*log(k)) 이고 공간복잡도가 O(n+k)임을 볼 수가 있다.

I. Introduction

The priority queue is an important abstract data structure in computer science. It has been successfully used for applications such as sorting, network optimization, discrete-event simulation, and state-space searches [8].

The operations defined in priority queues are as follows :

(a) min : returns the element or the address of the element with the smallest priority ;
(b) insert(k) : adds the item containing the key k to the queue Q ;
(c) delete : removes the item containing the smallest key from the queue ;
and

*Hoseo Univ. Dept. of Computer Science
접수일자 : 1995년 1월 9일

(d) merge(q, q') : all elements of q' are added to q while q' is destroyed.

A data structure implementing a priority queue is often called a heap. The heap is considered an optimal implementation of a priority queue and has been the subject of almost three decades of research. In essence, a heap [1, 2, 7, 8] is a tree has the following properties : (a) it is heap-ordered ; that is, a key contained in any node is not greater than the keys of its offspring ; and (b) all leaves are on, at most, two adjacent levels, and all leaves on the last level are as far to the left as possible.

A heap is a min heap if it supports operation min which returns the element or the address of the element with the smallest priority and is a max heap if it supports operation max which returns the element or the address of the element with the largest priority. Operation (d) is supported by the leftist heaps proposed by Crane and by the binomial queues proposed by Vuillemen. Other priority queue implementations include the skew heap, the Fibonacci heap, the relaxed heap, and the pairing heap. Recently, Olariu et al. [4] suggested a double-ended priority queue implementation called the min-max-pair heap, which supports sublinear time merging. Y. Ding and M.A. Weiss [8] published a priority queue implementation called the relaxed min-max heap, which supports all the priority operations. The key idea of this method is that, by properly relaxing the order restrictions for min-max heaps, it can merge two regular-sized min-max heaps. To merge two regular-sized min-max heaps, however, this method uses the blossomed tree and the lazying merging method.

In this paper, we provide a priority queue implementation called an improved relaxed min-max-pair heap, which efficiently supports the priority queue operation (d). Throughout the improved relaxed min-max-pair heap, we solve the problem without the blossomed tree and the lazying merging method that are used to merge two relaxed min-max heaps in [8]. Then, we proposed a sequential algorithm to efficiently merge the relaxed min-max heaps. For purposed of our discussion, the relaxed min-max nheaps are split into two cases : (1) two perfect heaps of equal size : and (2) two heaps of different sizes. If size n of the relaxed min-max nheap and size k of the relaxed min-max kheap are the same, the merging algorithm is determined by the creation function. This algorithms is based on an array implementation of the relaxed min-max heaps. As a result, we require the time complexity of $O(\log(\log(n/k)) * \log(n))$.

The rest of this paper is organized as follows. Section Ⅱ presents the basic definitions related to the merging and use of relaxed min-max heaps. Section Ⅲ describes the basic algorithm for merging two heaps of size n and k. Section Ⅳ discuss the results of this method, and Section Ⅴ presents our conclusions.

Ⅱ. Basic Definition

We will first give some definitions related to the merging of relaxed heaps and then define a new data structure : namely, the improved relaxed min-max-pair heaps.

We defined a perfect heap as a heap with $2^i - 1$ elements, in which all leaves are on the same level ; otherwise, the heap is non-perfect. The pheap is rooted at p, similar to the subheap rooted at p. Let the size(heap) refer to the number of elements it contains, and the height be defined as $\lfloor \log(\text{size(heap)}) \rfloor$. We introduce a function h(heap) to return the height of a heap. We define slots of those leaf positions in nheap that are to be filled by merging processes [7]. In this paper, we regard the relaxed min-max nheap as an nheap, the relaxed min-max kheap, as a kheap and the relaxed min-max pheap as a pheap.

In order to support double-ended heap operations, we want the tree to hold min-nodes and

max-nodes alternately. We, therefore, label the nodes in a tree to characterize the expected distribution of max nodes and min nodes.

A min-max-labeled tree T of depth d is a pertect tree in which each node v has a label $l(v) \in \{max, min\}$ such that (1) for any two nodes v and w of the same level, $l(v) = l(w)$; and (2) for any node v, if $w \in children(v)$, then $l(v) \neq l(w)$. For any node v, if $l(v) = max$, then we say it is max-labeled ; otherwise, we say it is min-labeled.

Given a set S of values, a min-max heap on S is a binary tree T with the following properties : (1) T has the heap-shape ; and (2) T is min-max ordered : values stored at nodes on even(odd) levels are smaller(greater) than or equal to the values stored at their descendants (if any) where the root is at level 0. An example of a min-max heap is shown in Fig. 1.

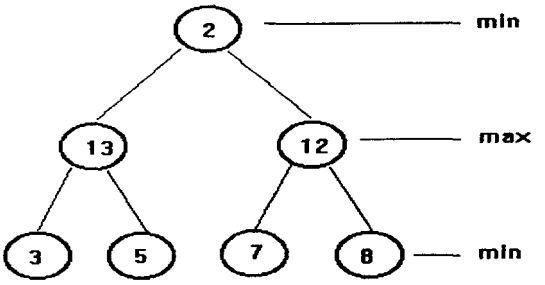

Fig. 1. Sample of a relaxed min-max heap

A relaxed min-max tree T is a min-max-labeled tree such that, for any node v in T, neither children(v) nor grandchildren(v) may contain more than one relaxed node. Clearly, and subtree of a relaxed min-max tree is also a relaxed min-max tree. This example is shown in Fig. 2.a. Min-max-pair heap is a binary tree H featuring the relaxed heap-shape property, such that every node in H 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 is stored in the min field of H[i] ; similarly, the value stored in the max field of H[i] is the largest key stored in the subtree of H rooted at H[i].

In order to easily merge the relaxed min-max heaps, we suggest the improved relaxed min-max-pair heap, which is better than the relaxed min-max heaps. This data structure's property is as follows :

(1) T has the heap-shape ;

(2) T is a relaxed min-max tree ; and

(3) T is the min-max-pair ordered.

But, in order to give the min-max value of each level, we proceed from the root to leaves. And so, in both the present level and the following level, we decide the min-max-pair. If the level that has not changed treats it as a new root level, we proceed to every leaf.

This new data structure is made during the merging of relaxed min-max heaps, and we use it. An example of the new improved relaxed min-max heap described above is shown in Fig. 2.b.

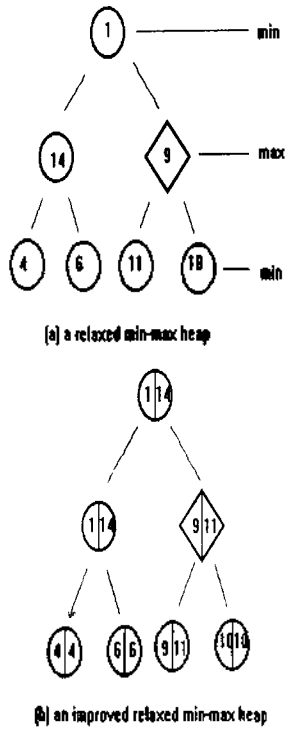

Fig. 2. An example of a improved relaxed min-max-pair heap

Ⅲ. Merging the Relaxed Min-Max Heaps

In order to efficiently merge the relaxed min-max heaps, we develop the algorithm by first showing how to merge two perfect heaps of equal size, and then by showing how to merge two heaps of different sizes.

(1) Merging Two Relaxed Min-Max Heaps of Equal Size

To merge two perfect relaxed min-max heaps, nheap and kheap, each of size k(=n), we first compare the roots of two perfect relaxed min-max-pair heaps. If the root of kheap is smaller than the root of nheap, we consider it the root of the newheap. Otherwise, we consider the root of nheap as the root of the newheap. Then, the rightson of the newroot is the relaxed min-max heap in which its root has the smaller value, and the leftson is the larger value after comparing the roots. As a result, the newheap produces a newheap with 2k(=2n) elements. But, at this time, the newheap has not satisfied the relaxed min-max heap's condition. To meet the relaxed min-max heap's condition, we must recreate half of the newheap. [see Fig. 3] The following pseudo-algorithm describes this.

```
procedure merge-equal-perfect-heaps(nheap, kheap)
begin
    Creation as a improved relaxed min-max-pair
    heaps for a relaxed min-max nheap or kheap
    call simple-merge-heaps(nheap, kheap)
    We made the relaxed min-max heap from the
    improved relaxed min-max-pair heap.
end

procedure simple-merge-heaps(nheap, kheap)
begin
    if(root(nheap) < root(kheap))
        copy kheap to leftson of newroot
        copy nheap to rightson of newroot
    else newroot = the root of kheap
        copy nheap to leftson of newroot
        copy kheap to rightson of newroot
    endif
    invoke the min-max heap's condition from the
    level which the root's value has the same to leaf.
    copy the last node of rightson to changed node.
end
```

Theorem 1. Two relaxed min-max heaps of equal size n can be merged with O(log(n)) comparisons.

Proof. The number of comparisons in the above algorithm to merge two relaxed min-max heaps of equal size is dominated by the creation operation, which requires O(log(n)) comparison [2, 7, 8]. Also, we need the time complexity of O(log(n)), which creates an improved relaxed min-max-pair heap. This algorithm, therefore, takes O(log(n)). ■

(2) Merging Two Heaps of Different Sizes

Here, we will consider the simple case of inserting a heap of k elements, kheap, into a heap of n elements, nheap. Without the loss of generality, assume that $k \leq \lfloor \log(\text{size}(\text{nheap})) \rfloor$.

We proceed with three phases as follows. In the first phase, we determine the level of the root of slots which has k by the merging process in nheap. Second, the pheap, (that is, the subheap of nheap that is allocated in the nheap) and the k'heap, (that is, the subheap of kheap that is allocated in the kheap) are merged. At this time, the merged new subheaps(pheap+k'heap) are satisfied with the relaxed min-max heap condition. In the last phase, we connect the newly merged heap to nheap, which is an original relaxed min-max nheap, and construct the relaxed min-max heap condition for nheap. This pseudo-algorithm is as follows.

```
procdure different-size-perfect-heap(nheap, kheap)
```

```
/*Initially we made first the relaxed min-max-
  pair heaps of nheap and kheap*/
begin
(1) /*we find the root of subhheap, that is location
      p which is allocated in each subheap of
      nheap, R(i)(i=1, 2, ..., p, where p is the
      number of 2^(h(kheap)-1)+1). Then we deter-
      mined the difference of height between
      nheap and kheap*/
    if(nheap is a perfect heap)
      then perfect-level-find(nheap, kheap)
      else nonperfect-level-find(nheap, kheap)
    (1.a) for(i=1 to p) do
        repeat
          move the subheap of nheap which is
          pointed to each subheap that is,
          the location p of each
          subheap and so we consturct the pheap.
        until(nheap's last node)
      endfor
    (1.b) for(i=1 to p) do
        while(the number of slots in each pheap
        is not equal to 0) do
          move the subheap of kheap which is
          equal to the number of slots in the each
          pheap and so we consturct the k'heap
        endwhile
      endfor
(2) for(i=1 to p) do
    /*pheap which is made from nheap and
      k'heap made from kheap are merged and
      we make the newheap*/
    union(pheap, k'heap)
    creation(subheap(pheap+k'heap))
  endfor
  if(the root of subheap(pheap+k'heap) is changed)
    then cv=1 else cv=0
(3) repeat
  while(cv in each subheap=1) do
    we construct the nheap by comparing the
    root of nheap to the previous node of p
    indicated by each subheap and then we main-
    tain the relaxed min-max heap's condition.
    (see procedure construct-two-heaps)
  endwhile
until(all cv in each subheap=0)
end
```

Ⅲ.1 Level-Find Algorithm

In the first phase of the merging process, to determine the location p's node, (that is, the root of the subheap to allocate the slots from kheap), we use the level of nheap. Then, we classify nheap as a perfect heap or a non-perfect heap.

(a) Perfect Heap

In the process of selecting the location p of the node, we must fill out a characteristic of the heap from the leftmost node of nheap in order to fill out the node of kheap, since nheap is a perfect heap. In this perfect heap, there are two steps in finding the location p of the node. In the first step, we determine the number of subheaps in nheap allocated all slots in order to find the location p in nheap. The number of subheaps, however, is equal to the number of leaves in kheap. In the second step, we select the locations as the number of subheaps determined. Since nheap is a perfect heap, the first subheap is located to the leftmost leaf or subheap of nheap. Also, we store the number of slots that are filled in each subheap. In order to do this, we use the variable S(R(i), i=1, 2, ..., 2^{l-1}, where l is the number of the level in kheap). The process of finding p is as follows :

```
procedure perfect-level-find
/*S(i) : the number of slots which the ith location
  has*/
(1) /*determine the number of subheaps*/
    p=the number of leaves in kheap
(2) /*determine the location of R(i) in nheap*/
    for(i=1 to p) do
      i=2^(h(nheap)-1)+i-1
      R(i)=the ith location of nheap
    endfor
```

(3) /*determine the number of slots which R(i) has*/
for(i = 1 to p) do
S(R(i)) = the number of slots in R(i)
endfor
end

Theorem 1. In the procedure perfect-level-find, it runs O(log(k)).

Proof. To determine the number of subheaps, we need the number of leaves in kheap ; that is, p. In step 2 and 3, it requires O(log(k)), since p means the height of kheap. Therefore, it takes O(log (k)). ■

(b) Non-perfect Heap

If nheap is a non-perfect heap, three steps are necessary to select the location p of the node. In the first step, we determine the number of subheaps in nheap in order to allocate all slots from kheap. We thus determine the location of the determined subheap's root. Then, using the difference of the height between nheap (h(nheap)) and kheap (h(kheap)), we find the location that is the root of the subtree that is not a first complete binary tree from each subtree of the level determined ; that is, if the difference between the size of the subtree determined and the slot is not less than 1, we find the lower subtree and select the non-perfect heap that has the difference of 1. We allocate the selected location to the first subheap in the kheap.

In the second step, we allocate the next location determined to the next subheap and so on. Then, the number of subheaps allocated is equal to the number of p determined in the first step. The number of slots allocated in each subheap is set to S(R(i)) such as in a perfect heap. The following pseudo-algorithm is the process of selecting the location p.

procedure nonperfect-level-find
begin
(1) /*determine the number of subheaps*/
$p = 2^{(h(kheap)-1)} + 1$
(2) /*determine the location of the first subheap*/
(2.a) level = h(nheap) − h(kheap)
if(level <= 0) then level = 0
$pl = 2^{(level)}$ /*calculate the current level*/
(2.b) if(nheap is not a leaf)
(a) if(the subheap of pl is a perfect heap) then pl = pl + 1 : go to step (2.b)
(b) LD = size(pl's subheap) - size(pl's slot)
(c) if(LD ≤ 1) then go to step 2.c
(d) if(the subheap of 2*pl is a perfect) then pl = 2*pl + 1 else pl = 2*pl
(e) go to step 2.b
(2.c) R(1) = the location pl of nheap
S(R(1)) = the number of slots in R(1)
(3) /*allocate the location from 2nd subheap to pth*/
for (i = 1 to p) do
R(i) = pl of nheap + (i − 1)'s location
S(R(i)) = the number of slots in R(i)
endfor
end

Theorem 2. To execute to find the location p of subheaps requires O(log(n/k)).

Proof. Step 1 of procedure nonperfect-level-find needs O(1), which is the height of kheap. In step 2, (2.a) requires O(1), which is the difference between the height of nheap and kheap, and (2.b) determines the location p of the root node of slots in nheap. The process to determine the path from the root of nheap to location p is log(n) − log(k) = log(n/k). (2.c) requires O(1). This procedure, therefore, requires O(log(n/k)). ■

Ⅲ.2 Merging Algorithm

Using subheaps allocated in the above section, we suggest the merging method between k'heap (which is the subheap of kheap) and pheap, (which is the subheap of nheap).

To select the k'heap which is merged with pheap, we move the number of slots in kheap that are assigned to subheaps in nheap. To execute

this, we point out the location of kheap, which has the ith location indicated by the total number of slots already assigned to the previous subheaps. Then, from the location of kheap determined, we create the merged k'heap, which constructs the number of slots to the subheap.

Fox example, in Fig. 4.c, we assume the circle represents the internal node and the square represents the slot. We store the number of slots of each subheap to S(R(i)). To determine the location of kheap indicated in the subheap, we select the location of kheap using X(R(i)). Then each X(R(i)) is initialized with 1. If S(R(i))={2, 2, 1}, it represents in Fig. 3 the value of S(R(i)). Since the number of slots in the first subheap is S(1)=2, and the location indicated in kheap of the first subheap is 1(X(1)=1), the number of slots indicated by the first subheap is 2(S(1)=2) for the first location of kheap.

Since the number of slots in the second subheap, R(2), is S(2)=2, and the location indicated in kheap of the second subheap is 3 (which is the sum of S(1)=2 and 1), it points out two nodes from the third location of kheap which has the same number of slots (that is, 2(S(2)=2)) as the second subheap. Also, the third subheap R(3) has the number of slots S(3)=1 and X(3)=5(2(S(1)+2(S(2))+1); that is, it indicates only one, node, since the third subheap pointed out the number of slots(S(3)=1) in the 5th location of kheap. The following pseudo-algorithm describes these points.

```
procedure selection-kheap's point
begin
int X[1 : n]
(1)/*determine the location in the kheap which
      each subheap has*/
   sum=0
   for(i=1 to p) do
     sum=sum+S[i-1]
     X[i]=X[i]+sum
   endfor
```

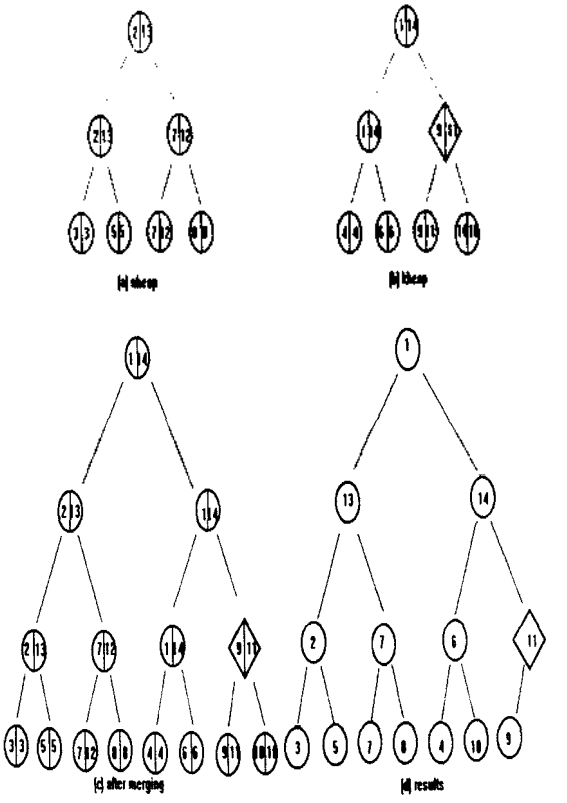

Fig. 3. The example of perfect heap which has a equal size

```
(2)/*move the number of slots determined from
the kheap*/
   (2.a) for (i=1 to p) do
   (2.b)   for (j=X[i] to (X[i]+S[i+1]-1)) do
             move the jth location of kheap to
             proper R(i)
           endfor
         endfor
end
```

Theorem 3. The above procedure runs O(log(k)*(the number of slots one subheap has)).

Proof. In the above procedure, (1) requires O(p) since it runs p-1 times using the number of p's subheap. The first step runs O(p), (2.a) requires O(p) and (2.b) requires O (the number of slots that one subheap has) since it occurs as the number of slots from the proper location of

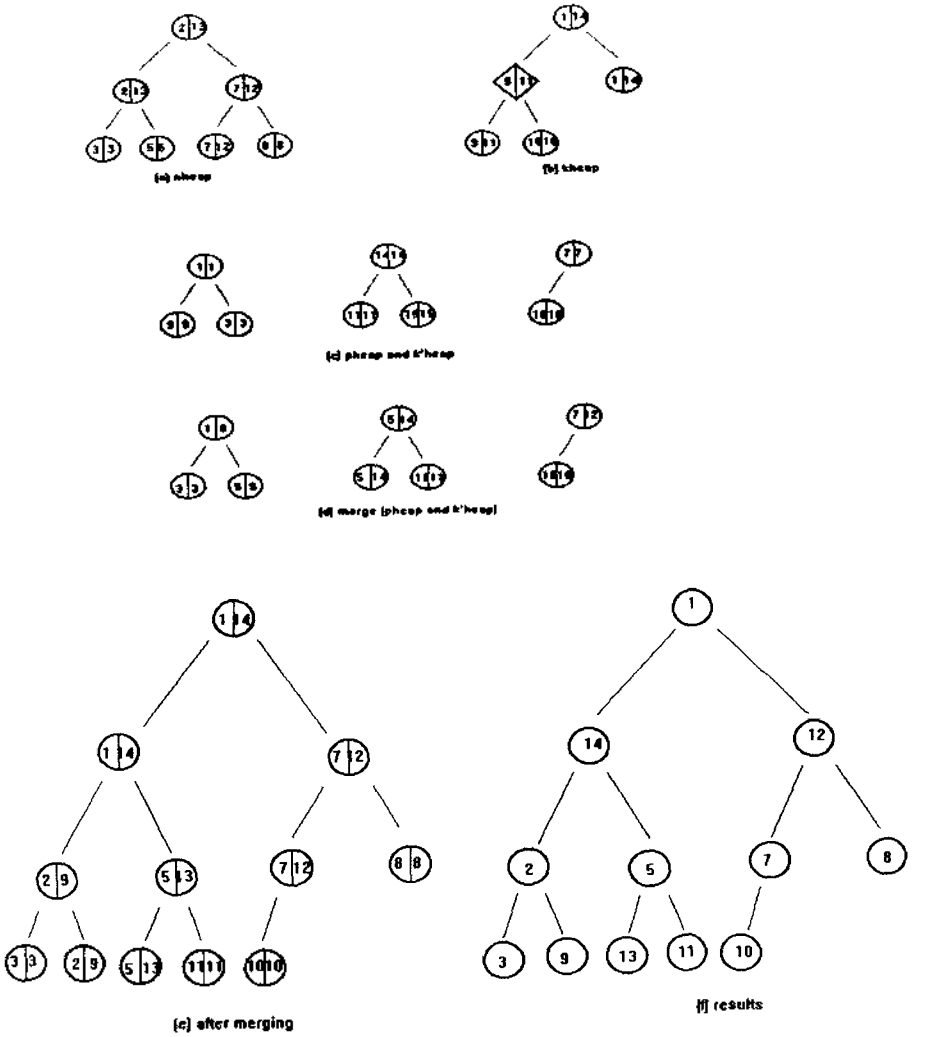

Fig. 4. The example of heaps with different sizes

kheap. Therefore, it runs O(log(k)*(the number of slots that one subheap has)), where p means log(k) since it is the height of kheap. ■

Next, we consider that the pheap and the k'heap are merging.[see Fig. 4d] The following pseudo-algorithm describes this.

```
procedure union-heaps
/* Pheap and k'heap is constructed by his own min-
   max values */
begin
if(the pheap and the k'heap are perfect heaps)
  then call merge-equal-perfect-heaps(pheap, k'heap)
  else { for (i = 1 to p) do
    (1) if(size(pheap) > size(k'heap))
          then newroot = {last element in pheap}
            change the location of pheap and k'heap
          else newroot = {last element in k'heap}
    (2) distribute pheap to temporary location t
        according to the rules which an improved
        relaxed min-max heap has
    (3) place newroot at pt
    (4) copy t to leftson of newheap(pt)
```

```
    (5) copy k'heap to rightson of newheap(pt)
    (6) creation(newheap) according to the orig
        inal rules which an improved relaxed min
        max pair heap has
    endfor }
end
```

In the above procedure, if the pheap and the k'heap are perfect heaps, they are merged using the same method as the perfect heap's case. Otherwise, this procedure constructs a heap as we treat the last node of the higher heap among two heaps : that is, the pheap and the k'heap as the root of the newheap. Since the newheap must satisfy the relaxed min-max heap's condition, we use the creation function. We acquire the resultant heap while the newheap moves the proper location of nheap. But, if the root of the newheap is changed, nheap does not satisfy the relaxed min-max heap's condition. To solve this problem, we use the variable cv in this paper. If the root of k'heap is changed after merging the pheap and the k'heap, we set the value of cv as 1. Otherwise, the value of cv is 0. Then, to meet the relaxed min-max heap's condition, we use the creation function, and we repeat this method untill all cvs that each subheap has are equal to 0.

```
procedure construct-twoheaps
begin
  (1) n = log⌊ size(nheap after merging)⌋
  (2) for (I = n downto 1) do
        k = 2^(i-1)
        s = 0
  (3)   m = min( size(nheap after merging)⌊/2, 2*k-1)
  (4)   for (j = k to m) do
          p = 2*j
          if(p < ⌊size(nheap after merging)⌋ and
            nheap(p) > nheap(p+1)) then p = p+1
          if(nheap(p) < nheap(j))
            then exchange(nheap(p), nheap(j))
        endfor
      endfor
  (5) for (i = 1 to p) do
        creation(R(i)'s subheap)
        if(the root node is exchanged)
          then cv = 1 else cv = 0
      endfor
  (6) for (all cv in R(i) = 0) do
        return(relaxed min-max heap)
      endfor
  endfor
end
```

Theorem 4. The time complexity of the above procedure is $O(\log(n/k)*\log(n))$.

Proof. In the above procedure, step 4 requires $O(\log(n/k))$ since it exchanges the subheaps from the location which is the difference between h (nheap) and h(kheap), to the root. Steps 2 through 4 require $O(\log(n)*\log(n/k))$.

Further, since step 5 requires log(n) as a creation function, it runs $O(\log^2(p+k'))$ because the subheap is pheap and k'heap. Steps 1 through 6 need the value of cv≠0 : that is, it does not change the root node of the subheap in all subheaps. This is what we indicate from the root of nheap to location p, which the subheap pointed out. This time complexity is $O(\log(n/k))$ as seen in theorem 1. Therefore, it runs $O(\log(n/k)*\log(n))$. ■

Ⅳ. The Analysis of Merging Relaxed Heaps

The time complexity can be computed in the non-perfect heap as follows. The first phase of merge finds the location p of a node allocated in each subheap in nheap. If nheap is a perfect heap, it requires O(p). Otherwise, it requires $O(\log(n/k))$. The second phase takes $O(\log(k)*$(the number of slots that one subheap has)). The third step requires $O((\log(n)*\log(n/k))+\log^2(p+k'))$. Therefore, the total time complexity is $O((\log(n/k)*\log(n))$.

The space complexity is computed as follows : First, regarding the size of nheap and kheap,

which include the min and max field, 2n and 2k, respectively. O(n+k) space is required. Second, since each subheap needs the pheap and kheap, it needs 2p and 2k', (that is, O(p+k')) since it needs pheap(size 2p) and k'heap(size 2k'). Therefore, the total space complexity is O(n+k), which means O(n+k+p+k').

To run the algorithm on a practical machine using C-language, we use from 0.1 million to 8 million data which were randomly generated and have no equal values. Also, we choose each data point, which was obtained as the average of 20 program executions, each on a different set of test data. As a result, two relaxed min-max heaps yield the 627.6686 seconds for 8 million data : 297.13 seconds for 4 million, 66.3 seconds for 1 million, 24 seconds for 0.4 million : and 5.3 seconds for 0.1 million.

V. Conclusion

This paper presents a new data structure that efficiently merges relaxed min-max heaps. This structure implements a mergeable double-ended priority queue to support very efficient merging. The improved relaxed min-max-pair heaps has a disadvantage, however. It requires more storage because each node in an array has two fields, a min field and a max field. Despite the problem of space utilization, we can efficiently merge two relaxed min-max heaps without the blossomed tree and the lazying method used in [8]. This result shows that, in two perfect heaps, the time complexity takes O(log(k)), but, in two heaps of different sizes, the time complexity requires O(log(n/k)*log(k)), assuming $k \leq \lfloor \log(\text{size}(\text{nheap})) \rfloor$ and the space complexity takes O(n+k). When we ran the algorithm on a machine using C-language, it required 627.6686 seconds for 8 million data, which consisted of two relaxed min-max heaps of different sizes. This practical time represents the average time of running 20 programs. Also, we think about that this method may be applicable to the parallel machine.

Acknowledgement

I thanks Dr. Zheng for his insighful comments, which has helped to improve the results and the presentation of this paper. And I thanks Dr. Prasad who supplies me more data to complete this paper.

References

1. Aho, A.V., Hopcroft, J.E. and Ullman, J. D., "The Design and Analysis of Computer Algorithm," Addision-Wesely, 1974.
2. Atkison, M., Sack J., Santoro, N. and Strothotte, T., "Min-max Heaps and Generalized Priority Queue," Commu. ACM, Vol.29, No. 10, pp. 996-1000, 1986.
3. Gonnet, G.H. and Munro, J. I., "Heaps on Heaps," SIAM Journal of Computing, Vol. 15, No. 4, pp. 964-971, Dec. 1986.
4. S. Olariu, C.M. Overstreet and Z. Wen, "A Mergeable Double-Ended Priority Queue," Computer Journal, Vol. 34, No. 5, pp.423-427, 1991.
5. Stasko, J. T. and Vitter, J.S., "Pairing Heaps : Experiments and Analysis," Commu. ACM, vol. 30, no. 2, pp.234-249, Mar. 1987.
6. Strothotte, Thomas and Sack, J.R., "An Algorithm for Merging Heaps," Acta informatica 22, pp. 171-186, 1985.
7. Y. Ding, M.A. Weiss, "The Relaxed Min-Max Heap, A Mergeable Double-Ended Priority Queue," Acta informatica 30, pp. 215-232, 1993.

▲Yong Sik Min(Regular Member)

1981년 2월 : Dept. of Computer Science, Kwangwoon Univ.(B. S.)
1984년 2월 : Dept. of Computer Science, Kwangwoon Univ.(M. S.)
1991년 2월 : Dept. of Computer Science, Kwangwoon Univ.(Ph. D)
1984년 3월 ~ 1987년 2월 : Full-time lecturer, Songwon Junior College Dept. of Computer Science
1987년 3월 : present : Associate Professor, Hoseo Univ. Dept. of Computer Science
1993년 8월 ~ 1994년 8월 : Visiting Professor, Louisiana State Univ. Dept. of Computer Science