Assignment 4

Fu Guanshujie, Ge Yuhao, Lou Haina, Qiu Xiaomin

March 2021

1 Problem 1

1.1

1.The running time of **SiftUp(n)** is $O(log(n)$.
Proof:

Algorithm 1 SiftUp

Input: i the index
Output: heap after SiftUp

```
1: function SiftUp(i)
2:     child ← i
3:     father ← child/2
4:     temp ← Maxheap[i]
5:     while father >= 1 And child >= 2 do
6:         if Maxheap[father] < temp then
7:             Maxheap[child] ← Maxheap[father]
8:             child ← father
9:             father ← child/2
10:        else
11:            break
12:        end if
13:    end while
14:    Maxheap[child] ← temp
15: end function
```

Assume the element's index is i and we need to shift it up, then its fathers' layers is less than $log(i)$. If the father is smaller than the element, we swap them **(line 8)**, and then continue to compare it with its father **(line 5)**. Thus, we need to perform less than **log(i)** times swap operations, which is in **O(log(n))**.

2.An **insert** into a heap takes time in $O(\log(n))$.
Proof: Thus, an insert into a heap takes about **c+log(n)** time, which is in **O(log(n))**.

Algorithm 2 Insert

Input: $value$ the value to be inserted
n the number of the elements in the heap
Output: heap after Insertion

```
function Insert(value)
    n++
    Maxheap[n] ← value
    SiftUp(n)
end function
```

1.2

Suppose that we have an element at the root to be sifted down. Normally, we need to compare it with all its child nodes in every layer of a heap which requires two comparisons and hence results in **2log(n)** for $SiftDown$. To modify it, we just compare its child nodes to denote the larger child and continue comparison of the child nodes until the leaf nodes. **This process need $log(n)$ comparisons: line 5.** We store the larger child nodes' index into $Patharray$, which comprise a path along which elements need to be swapped. Then, we perform a binary search on this path to find the proper position for the root element and modify the heap for $SiftDown$. The number of elements in the $Patharray$ is about **log(n)**. **This process need $O(loglog(n))$ comparisons: line 15.** We can prove it through following equations: As new data size is less than half each time with one additional comparison, we get:

$$T(log(N)) = 1 + T(\frac{log(N)}{2}) = 2 + T(\frac{log(N)}{2^2}) = \cdots = n + T(\frac{log(N)}{2^n})$$

Assume $log(N) = 2^i + k$, then $i = loglog(N)$.

$$T(log(N)) <= \lfloor loglog(N) \rfloor + 1 <= loglog(N) + 1 = O(loglog(N))$$

Thus, the comparisons required in the $SiftDown$ algorithm are reduced to about **log(n) + O(log log(n))**.

Algorithm 3 SiftDown

Input: i the index of the root
n the number of the elements in the heap
Output: heap after SiftDown

1: **function** SiftDown(i, n)
2: $m \leftarrow i$
3: $root \leftarrow Maxheap[i]$
4: **for** $j \leftarrow 2 * i$, $k \leftarrow 1$; $j <= n$; $j \leftarrow j * 2$, $k++$ **do**
5: **if** $j < n$ **and** $Maxheap[j] < Maxheap[j+1]$ **then**
6: $j++$
7: **end if**
8: $Patharray[k] = j$
9: **end for**
10: **if** $root >= Patharray[1]$ **then return**
11: **end if**
12: $left \leftarrow 1$, $right \leftarrow k$
13: **while** $left < right$ **do**
14: $mid \leftarrow left + \frac{right-left}{2}$
15: **if** $Patharray[mid] > root$ **then**
16: $left \leftarrow mid + 1$
17: **else**
18: $right \leftarrow mid$
19: **end if**
20: **end while**
21: $right--$
22: $Maxheap[m] \leftarrow Maxheap[Patharray[1]]$
23: **for** $k \leftarrow 2$; $k < right$; $k++$ **do**
24: $Maxheap[Pathheap[k]] \leftarrow Maxheap[Patharray[k+1]]$
25: **end for**
26: $Maxheap[Patharray[right]] \leftarrow root$
27: **end function**

---

2 Problem 2

2.1

For pair (a, b), use $(ab)\%10$ as hash function, and each value correspond to the index of an array. Each element in the array points to another list which contains all the pairs that share the same hash value.

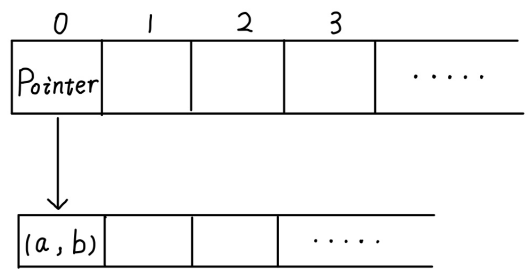

Figure 1: Hashing

3 Problem 3

3.1

For pairing heaps with 3 pointers for each item, we first define these pointers as:
ptr_c: pointer to oldest child
ptr_y: pointer to younger sibling
ptr_o: pointer to older sibling or its parent
To execute $deleteMin$ for a pairing heap, we consider following steps to execute $deleteMin$ and rebalance the heaps:
1. Use $H.min$ to point at the min root in the root sequence when building the heaps.
2. Delete the minimum root pointed by $H.min$ by setting the ptr_o of its child to NULL.
3. Add all the child nodes of deleted node into the root sequence.
4. Combine the remaining roots nodes from right to left pair by pair and then repeat until we get a single root.

We will explain the details in step 4, mainly on how to combine the root nodes. Let ptr_t and ptr_n stand for two neighboring root nodes. By comparing these two nodes, we add the larger node into the child of the smaller node, for which we need to change four pointers as shown below.

Algorithm 4 Pairing

if $ptr_t.value > ptr_n.value$ **then**
  $ptr_n.ptr_c.ptr_o \leftarrow ptr_t$
  $ptr_t.ptr_y \leftarrow ptr_n.ptr_c$
  $ptr_t.ptr_o \leftarrow ptr_n$
  $ptr_n.ptr_c \leftarrow ptr_t$
end if

By continuing comparing and pairing two root nodes, we will finally combine all roots into a single heaps which satisfies the property of the heap. Figure 1 shows how two roots pairing.

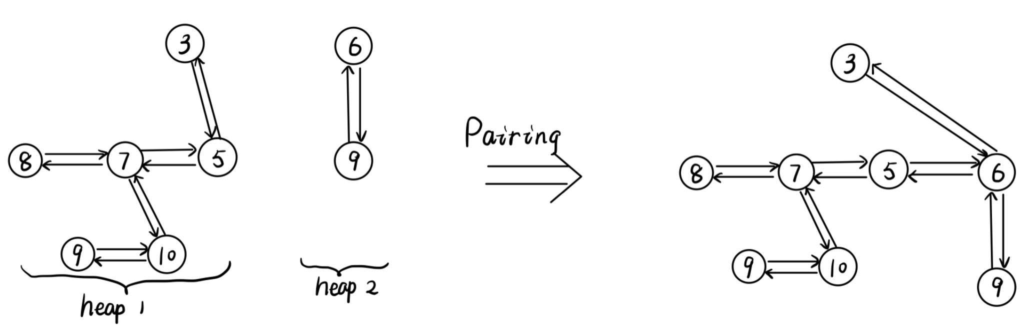

Figure 2: Pairing

3.2

For pairing heaps with 2 pointers for each item, the steps for $DeleteMin$ and rebalance is similar to above but with differences when change the pointers.

ptr_y: pointer to younger sibling or parent

ptr_c: pointer to oldest child

Step 1-3 are similar to above and we will mainly focus on the details of steps to explain how to pair with two pointers. Let ptr_t and ptr_n stand for two neighboring root nodes.

Algorithm 5 Pairing

if $ptr_t.value > ptr_n.value$ **then**

  $ptr_t.ptr_y \leftarrow ptr_n.ptr_c$

  $ptr_n.ptr_c \leftarrow ptr_t$

end if

Clearly, only two pointers need to be changed to add the root into the child nodes. Figure 2 shows how two roots with 2 pointers are paired.

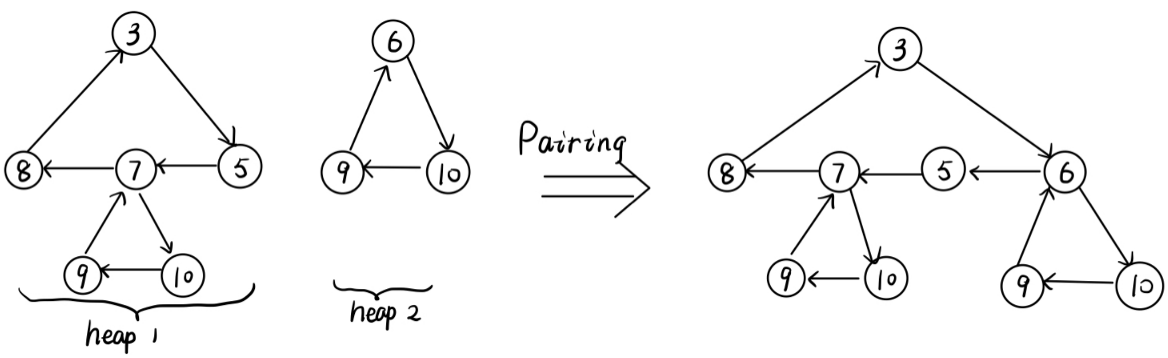

Figure 3: Pairing

4 Problem 4

4.1 Insert

Firstly, we do the insert step. $t(H') = t(H) + 1andm(H') = m(H)$, therefore the actual cost should be $((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1 = O(1)$. Then we do consolidate step. The size of the root list before calling consolidate is at most $D(n)+2$, also $t(H)+1$, every time through while loop, one of roots is linked to another, thus the total amount of work is $O(D(n))$. The potential before inserting is $t(H)+2m(H)$, and the potential afterward is at most D(n)+1+2m(H)). So worst-case running time is $O(D(n))+((D(n)+1)+2m(H))-(t(H)+2m(H)) = O(D(n))$. Because $D(n) < \log n$, time complexity should be $O(\log n)$.

4.2 Find Minimum

Because H is given by $min[H]$ and potential is the same, the cost is $O(1)$

4.3 Union

Firstly, the unit step. The change of potential should be $(t(H)+2m(H))-((t(H1)+2m(H1))+(t(H2)+2m(H2))) = 0$, so it's $O(1)$. Then we do consolidation step. The size of the root list before calling consolidate is at most $t(H1) + t(H2)$, In each iteration step two trees are combined, which takes constants time, so the total work should be O(t(H1)+t(H2)-1). The potential before is $t(H1) + t(H2) + 2m(H1) + 2m(H2)$, potential after is at most

$t(H1)+t(H2)+2m(H1)+2m(H2)$, so the potential difference is 0. Because $t(H1)+t(H2) <= t(H) <= D(n)-1$. Time complexity should be $O(\log n)$.

4.4 Extract Minimum

Same as lecture, time complexity should be $O(\log n)$.