CSC263 A2
January 31, 2019
Xu Wang, Jiatao Xiang, Huakun Shen

Question 1

a. Increase the key of a given item x in a binomial max heap H to become k
Increase(H, x, k):

1) Change the key value of x to k, assume $k > x$
2) While the *x has a parent (x is not a root)* and *the key of x's parent is smaller than k*:
Do step 3
3) switch the position of x and its parent

Worst Case Runtime:
Assume H has n nodes, $n =< b_t, b_{t-1}, ..., b_0 >_2$, where $t = \lfloor log_2 n \rfloor$
$H = F_n :$ $<$ all trees B_i such that bit $b_i = 1 >$
The largest binomial tree in H is B_t, whose number of node is 2^t, and height is t.
Then, in the worst case, x needs to be switched at most t times to become the root of its binomial tree.
Since $t = \lfloor log_2 n \rfloor$, x would be switched at most $\lfloor log_2 n \rfloor$ times.
$RT_{WC} = \mathcal{O}(log_2 n)$

b. Delete a given item x from a binomial max heap H.
Remove(H, x):

1. While x is not the root of its binomial tree:
do step 2
2. $Increase(H, x, \text{key of } x's \text{ parent} + 1)$
3. Locate the maximum node m of H, which is one of the roots of the binomial trees in the binomial heap
Let's say B_i is the binomial tree that contains m, isolate B_i and create a new binomial heap U
$U = H - B_i$
4. Delete the root node which was located in step 3, and make S a new binomial tree of the result
$S = B_i - m$ (m is the root of B_i, the max node in H)
5. $H \leftarrow Union(U, S)$

Worst Case Runtime:

- The goal of the step 1, 2 is to move and make x the root of its binomial tree. Every time x's key is increased to (x's parent's key +1), x switches with its parent. As explained in (a), a binomial heap with n nodes has a maximum binomial tree of height $\lfloor log_2 n \rfloor$, thus it takes at most $\lfloor log_2 n \rfloor$ basic operations to make x the root of its binomial tree. $RT_1 = \mathcal{O}(log_2 n)$
- Step 3 searches through the root of every binomial tree in the binomial heap to locate the maximum node in H
For a binomial heap with n nodes, it has $\lfloor log_2 n \rfloor$ binomial trees. Thus it takes at $\lfloor log_2 n \rfloor$ steps to locate the binomial tree with the maximum node. $RT_2 = \mathcal{O}(log_2 n)$
- Step 4 deletes the root of a binomial tree, which takes constant time. $RT_3 = \mathcal{O}(1)$
- Step 5 makes H the union of the results from step 3 and step 4, which takes $RT_3 = \mathcal{O}(log_2 n)$ of time
- $RT_{WC} = RT_1 + RT_2 + RT_3 + RT_4 = \mathcal{O}(log_2 n)$

In brief, the algorithm of $remove(H, x)$ is:

(a) $increase(H, x, \infty)$
(b) $extract\_max(H)$ (The sum of step 3 - 5)

Question 2

1. Our ***SuperHeap*** is based on *Binomial Max Heap and Min Heap*, with a little modification.
   Binomial Max Heap is basically symmetric to *Binomial Min Heap*, they have the same but inverse implementation, and we will build our ***SuperHeap*** based on it.
   We will also make use of our solution from **Question 1**, the *Remove(H, x)* function (which is also built on *Binomial Max Heap*.
   Idea: We use both max heap and min heap in our data structure to make super heap, which can trace both min and max values.
   How: We store an extra information (attribute) called **twinValue** in each node,"a pointer to the twin node in the other heap which has the same key."

2. Implementation of methods
   Let's call the SuperHeap SH, and the max heap $MaxH$, the min heap $MinH$.

   (a) *Merge(D, D'):* Similar to the algorithm of *Union* we have discussed in class, which takes $\mathcal{O}(log_2 n)$ of time, except we perform the steps twice. We have to union both max heap and min heap in our data structure. We discussed the steps to perform union in Binomial Min Heap, and steps to perform union in max heap is similar except we keep max value on the top. The worst case run time of merge is twice as large as union, thus is $\mathcal{O}(log_2 n)$.

   (b) *Insert(k):* When we perform insert operation, we have to insert the node k to both min heap and max heap, which means we need two node with the same value and they point to each other with their **twinValue** attribute. The process to insert one to min heap and the other to max heap is exactly the same as what we discussed in lecture. The worst case run time is twice as large as Insert which talked in class, thus is $\mathcal{O}(log_2 n)$.

   (c) *ExtractMax():*
   It's exactly symmetric to *ExtractMin()* in *Binomial Min Heap* that we discussed during lecture, except we need to remove the max value in both min heap and max heap in our data structure.

      i. Extract the max value in $MaxH$ by comparing the root of every tree in $MaxH$, which costs $\mathcal{O}(log_2 n)$. Let's call the node extracted $Node$.
      ii. Then, find corresponding max node in $MinH$ using the attribute **twinValue** of $Node$, which costs constant time.
      iii. Finally, we perform *Remove(MinH, Node.twinValue)* to *MinH* to delete the corresponding node. The remove function comes from part b of Question 1, which has a runtime of $\mathcal{O}(log_2 n)$. Note that, in Question 1, the *Remove* function is for Max Heap, here we use the Min-Heap version of Remove, which is exactly the opposite: decrease the key of a node to $-\infty$ and perform $ExtractMin()$ to $MinH$.

   Thus, *ExtractMax()* takes $\mathcal{O}(log_2 n)$ of time.

   (d) *ExtractMin():* ExtractMin function is almost the same as Extract Max, the only difference is that, this time, we extract the min value from both heaps.

      i. Extract the min value from $MinH$ by comparing the root of every tree in $MinH$, which costs $\mathcal{O}(log_2 n)$. Let's call the node extracted $Node$.
      ii. then, we find the corresponding min node in the $MaxH$ using the twinValue of $Node$, which costs constant time.
      iii. Finally, perform $Remove(MaxH, Node.twinValue)$ function to $MaxH$ to delete the corresponding node. The Remove function comes from part b of Question 1, which has runtime of $\mathcal{O}(log_2 n)$.

   Thus, *ExtractMin()* takes $\mathcal{O}(log_2 n)$ of time.

Question 3

a)

```
PathLengthFromRoot(root, k){
    if(key(root) == k){
        return 1;
    }
    if(k > key(root)){
                return 1 + PathLengthFromRoot(rchild(root), k);
    }else{
                return 1 + PathLengthFromRoot(lchild(root), k);
    }

}
```

Worst-Case Time Complexity: the height of the BST is h. Each step of the algorithm will increase depth by 1 and loop at most h times which is the height of the BST and thus is $\mathcal{O}(h)$.

b)

```
FCP(root, k, m){
    if(k <= key(root) <= m || m <= key(root) <= k){
                return root;
    }else if(k < key(root) && m < key(root)){
                return FCP(lchild(root), k, m);
    }else{
                return FPC(rchild(root), k, m);
    }
}
```

Worst-Case Time Complexity: Each step of the algorithm will increase depth by 1 and it will loop at most h times which is the height of the BST and thus is $\mathcal{O}(h)$.

c)

```
IsTAway(root, k, m, t){
    ParentNode = FCP(root, k, m);
    Path1 = PathLengthFromRoot(root, k);
    Path2 = PathLengthFromRoot(root, m);
    return (Path1 + Path2 <= t);
}
```

Worst-Case Time Complexity: the worst-case run time of FCP and PathLengthFromRoot is $\mathcal{O}(h)$ and thus the total runn time of IsTAway will also be c_1*h, where c_1 is a constant, thus it is $\mathcal{O}(h)$.