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*, 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: Always keep the minimum node of each binomial tree in the bottom, leftmost position, and make every node in the tree keep track of it. Then it takes less time to find the min node.
   How: We store an extra information (attribute) called **min** in each node,"a pointer to the leftmost(bottom) node in the binomial tree".
   Let's say we perform all operations on a *Binomial Min Heap* T, whose number of nodes= n.

2. Implementation of methods

   (a) *Merge(D, D'):* Similar to the algorithm of *Union* we talked about in class, which takes $\mathcal{O}(log_2 n)$ of time, except we perform one extra step, comparing and switching the leftmost node when merging two trees.
   Merging two heaps is in fact performing "merging two trees" a bunch of times.
   We first describe how to merge two trees. Call the two trees $t1$ and $t2$.

   i. Compare the root of t_1 and t_2, the greater root becomes the root of the new tree, and let's suppose t_1 has the greater root.

   ii. Then compare the leftmost node of both trees. If the leftmost node of the t_1 is smaller, then switch the node with the leftmost node of the t_2, to make sure the new tree's min node is in the leftmost position.

   iii. Note: When switching the two leftmost node, only switch the value, then we don't have to modify other node's **min** attriubute to point to the leftmost node.

   The rest is the same as binary addtion. Comparing and switching the value of two nodes takes constant time.
   Overall, *Merge(D, D')* still takes $\mathcal{O}(log_2 n)$ of time.

   (b) *Insert(k):*

   i. Make a new binomial tree T_2 with a single node with key k.

   ii. *Merge(T, T_2)* (takes $\mathcal{O}(log_2 n)$)

   Making a single node takes constant time, and *Merge(T, T_2)* takes $\mathcal{O}(log_2 n)$ of time.

   (c) *ExtractMax():*
   It's exactly symmetric to *ExtractMin()* on *Binomial Min Heap*

   i. Search through the roots of every binomial tree in the binomial heap, call the node "max" and the tree it belongs to "t". There are at most $\lfloor log_2 n \rfloor$ trees in the heap, so it takes $\mathcal{O}(log_2 n)$ of time to find max.

   ii. Let $U = T - t$ (U is the rest of the trees)

   iii. Let $S = t - max$ (S is the remainders of t after max is removed from it)

   iv. Let $T = U \cup S$ (This takes $\mathcal{O}(\lfloor log_2 n \rfloor)$ of time)

   ExtractMax(): takes $\mathcal{O}(\lfloor log_2 n \rfloor)$ of time.

   (d) *ExtractMin():*

   i. Loop through every tree of T, and Compare the minimum(using the **min** attribute of the root of each tree) node of every tree, and find the min of the heap.
   Let's call the minimum node x. (there are at most $\lfloor log_2 n \rfloor$ trees, thus takes $\lfloor log_2 n \rfloor$ of time)

   ii. Call *Remove(T, x)* from **Question 1** (takes $\lfloor log_2 n \rfloor$ of time)

   Overall, it takes $\lfloor log_2 n \rfloor$ of time to extract the minimum node from a our **SuperHeap**.

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 $\mathcal{O}(h)$.