CSC263 Assignment 2

Angela Zhu, Xinyi Ji, Zhuozi Zou

January 31, 2019

1 Problem 1

(Written by Zhuozi Zou, read by Angela Zhu & Xinyi Ji)

Since x points to an item inside H, we can assume that H is non-empty.

a. The following algorithm *Increase*(H, x, k) increases the key of a given item x in a binomial max heap H to become k only when $k > x.key$:
Step 1: If $k \leq x.key$, the algorithm ends. Otherwise increase $x.key$ to k. Now $x.key = k$.
Step 2: If x points to the root of H, the algorithm ends. Otherwise let m be the pointer to the parent node of the node x currently points to.
Step 3: If $k \leq m.key$, the algorithm ends. Otherwise exchange $x.key$ and $m.key$, and then move the pointer x one level up in the tree.
Repeat steps 2-3 until the algorithm ends.

From lecture 3 notes P.49, we know that the max depth of H is $\lfloor \log n \rfloor$, so steps 2-3 iterates at most $\lfloor \log n \rfloor$ times. Since each iteration of steps 2-3 takes constant time (because each one consists of a constant number of comparisons and exchanges), it is now clear that there is a constant $c > 0$ such that *for all* $n > 0$: for every binomial max heap H with n items, *Increase*(H, x, k) takes at most $c \cdot n$ time. Therefore, the worst-case running-time of *Increase*(H, x, k) is $\mathcal{O}(\log n)$.

b. The following algorithm *Delete*(H, x) deletes a given item x from a binomial max heap H:
Step 1: Do *Max*(H) (similar to *Min*(H) learned in class) to find the largest key in H. Let p be such largest key.
Step 2: Do *Increase*$(H, x, p+1)$ to set $x.key$ to the largest key of H. When *Increase*$(H, x, p+1)$ terminates, x will be the root of H (since $x.key$ is the largest key in H).
Step 3: Do *Extract_Max*(H) (similar to *Extract_Min*(H) learned in class) to remove the root of H. So x is removed from H.

From lecture 3 notes P.18, we know that *Max*(H) is $\mathcal{O}(\log n)$ and *Extract_Max*(H) is $\mathcal{O}(\log n)$. From 1a) we know that *Increase*$(H, x, p+1)$ is $\mathcal{O}(\log n)$. So it is clear that there is a constant $c > 0$ such that *for all* $n > 0$: for every binomial max heap H with n items, *Delete*(H, x) takes at most $c \cdot n$ time. Therefore, the worst-case running-time of *Delete*(H, x) is $\mathcal{O}(\log n)$.

2 Problem 2

(Written by Xinyi Ji, read by Angela Zhu & Zhuozi Zou)

1. I used binomial heaps as the underlying data structure of my solution. The SuperHeap I implemented here is a combination of the max binomial heap and the min binomial heap. The structure of it is quite similar to the min binomial heap we learned in class. The only

difference is that for every node, except the key, it has two degrees (one for min binomial heap, one for max binomial heap), two parent pointers (one for min binomial heap, one for max binomial heap), two child-left pointers (one for min binomial heap, one for max binomial heap), and two sibling-right pointers (one for min binomial heap, one for max binomial heap). Hence it can be treated as a min binomial heap or a max binomial heap.

2. *Insert*(k): First, do the *Insert*(k) of min binomial heap which we learned in the class on the min binomial heap part of the SuperHeap, and change the degree, parent, child-left, and sibling-right attributes in the min binomial heap part of the affected nodes. Second, do the *Insert*(k) of max binomial heap which is symmetrical to the *Insert*(k) of min binomial heap (wherever there is min change to max) on the max binomial heap part, and change the degree, parent, child-left, and sibling-right attributes in the max binomial heap part of the affected nodes. The inserted key, since it has the parent, degree, etc. attributes in both parts, will be a legal element of the super heap.

   ExtractMax(): First, do the *ExtractMax*() (which we learned in class on min binomial heaps, here, it is just a symmetrical operation) on the max binomial heap part, and change the degree, parent, child-left, sibling-right attributes in the max binomial heap part of the affected nodes. Second, perform *Delete*(x) (similar to the delete operation we used in question 1b) on max binomial heaps) on the max key which we deleted in the first step from the min binomial heap part. Then change the degree, parent, child-left, sibling-right attributes in the min binomial heap part of the affected nodes.

   ExtractMin(): First, do the *ExtractMin*() we learned in class on the min binomial heap part, and change the degree, parent, child-left, sibling-right attributes in the min binomial heap part of the affected nodes. Second, perform *Delete*(x) (we implemented in Question 1b)) on the min key which we deleted in the first step from the max binomial heap part. Then change the degree, parent, child left, sibling rights of the max binomial heap part in the changed nodes.

   Merge(D, D'): First, do the *Union*(D, D') of min binomial heaps that we learned in the class on the min binomial heap part, and change the degree, parent, child-left, sibling-right attributes in the min binomial heap part of the affected nodes. Second, do the *Union*(D, D') of max binomial heaps which is just a symmetrical operation of *Union*(D, D') of min binomial heaps (wherever there is min change to max) on the max binomial heap part, and change the degree, parent, child-left, sibling-right attributes in the max binomial heap part of the affected nodes.

3 Problem 3

(Written by Angela Zhu, read by Xinyi Ji & Zhuozi Zou)

a. Pseudocode:

PathLengthFromRoot($root, k$)
```
1    if (key(root) == k)
2        return 0
3    if (key(root) > k)
4        return PathLengthFromRoot(lchild(root), k) + 1
5    if (key(root) < k)
6        return PathLengthFromRoot(rchild(root), k) + 1
```

Description & Time Complexity:

This recursive algorithm first checks the base case: whether or not *node*(k) is *root* itself. If so,

then the length of the path is 0 and the value is returned. If not, the algorithm compares k and the value of *root*. If the key of *root* is larger than k, then $node(k)$ must be contained in the subtree rooted at the left child of *root* (according to the BST properties). Same for if the key of *root* is less than k, then $node(k)$ is contained in the subtree rooted at the right child of *root*. The algorithm then returns the length of the path between the child of *root* and $node(k)$, plus 1 which is the length of the path between *root* and its child. It is done by making a recursive call of the algorithm itself and replacing *root* by one of its children.

The worst-case time complexity of this algorithm is $\mathcal{O}(h)$. The procedure terminates when $key(root)$ is equal to k, and before so, the procedure is recursively being called on k and a child of *root*. Since h is the height of the BST rooted at *root*, it takes at most h recursive calls to go from *root* to its furthest leaf. In other words, after h calls, the *root* in the last call must be a leaf, therefore having no children and no more calls can be made. And, since the steps in the algorithm other than the recursive call (if statements, addition, etc.) take constant time, it is clear that there exists a constant $c_1 > 0$ such that for all BSTs of height h and for every input key k, executing the procedure $PathLengthFromRoot(root, k)$ takes at most $c_1 \cdot h$ time, therefore satisfying $\mathcal{O}(h)$.

b. Pseudocode:

$FCP(root, k, m)$
1 if $(k \leq key(root) \leq m$ or $m \leq key(root) \leq k)$
2 return *root*
3 if $(key(root) > k)$
4 return $FCP(lchild(root), k, m)$
5 if $(key(root) < k)$
6 return $FCP(rchild(root), k, m)$

Description & Time Complexity:
This recursive algorithm first checks the base case: whether or not the key of *root* is greater or equal to one of the input keys, and less or equal to the other input key. If so, then the current *root* is the desired *parent*, and the larger of these two keys is contained in the right subtree of *parent*, while the smaller one is contained in the left subtree of *parent*. This means that there is no further subtree of parent that is able to contain these two keys, thus the length of the path from *parent* to *root* must be the maximum, as it is impossible for the path to extend further.
If this base case is not satisfied, the algorithm compares k and the value of *root*. Comparing one of the two values is sufficient because both of them are either greater than or less than the key of *root*. If the key of *root* is larger than k, then $node(k)$ and $node(m)$ must be contained in the subtree rooted at the left child of *root* (according to the BST properties). Same for if the key of *root* is less than k, then the two nodes must be contained in the subtree rooted at the right child of *root*. The algorithm then returns the root of the subtree furthest away from the child of *root* that contains k and m, which this root is also the furthest away from *root* itself. It is done by making a recursive call of the algorithm itself and replacing *root* by one of its children.

The worst-case time complexity of this algorithm is $\mathcal{O}(h)$. Same logic as a), the procedure terminates when $key(root)$ is greater or equal to one of k and m, and less than or equal to the other. And before so, the procedure is recursively being called on k, m and a child of *root*. Since h is the height of the BST rooted at *root*, it takes at most h recursive calls to go from *root* to its furthest leaf. And, since the steps in the algorithm other than the recursive call (if statements, etc.) take constant time, it is clear that there exists a constant $c_2 > 0$ such that for all BSTs of height h and for every input key k, m, executing the procedure $FCP(root, k, m)$ takes at most $c_2 \cdot h$ time, therefore satisfying $\mathcal{O}(h)$.

c. Pseudocode:

IsTAway$(root, k, m, t)$
1 $parent = FCP(root, k, m)$
2 $kToParent = PathLengthFromRoot(parent, k)$
3 $mToParent = PathLengthFromRoot(parent, m)$
4 $length = kToParent + mToParent$
5 return $length \leq t$

Description & Time Complexity:
This algorithm simply calculates the length of the path between k and m, compares it to t, and returns the result. It first finds *parent*, which is the root of the subtree that is furthest away from *root* containing k and m, using the $FCP(root, k, m)$ procedure from b). Then the algorithm calculates $kToParent$ and $mToParent$, which is the length of the path between k, m to *parent*, using the $PathLengthFromRoot(root, k)$ procedure from a). The length of this path between k and m, *length*, is calculated by adding $kToParent$ and $mToParent$.
As we go up the tree from $node(k)$ and $node(m)$ by finding their parent, and the next parent, etc., we may get two separate paths at first. However, these two separate paths will eventually meet at one parent, which is the *parent* we find using $FCP(root, k, m)$, then we would have a connected path between k and m. Thus, if *length* is less than or equal to t, this implies that the length of the path between k and m is at most t, the algorithm returns ***true***, and vice versa.

The worst-case time complexity of this algorithm is $\mathcal{O}(h)$. The algorithm consists of one call to the procedure $FCP(root, k, m)$, two calls to the procedure $PathLengthFromRoot(root, k)$, and several steps that take a constant time c. According to a) and b), we know that $FCP(root, k, m)$ takes at most $c_2 \cdot h$ time, $PathLengthFromRoot(root, k)$takes at most $c_1 \cdot h$ time. Thus, it is clear that for all BSTs of height h and for every input k, m, t, executing the procedure *IsTAway*$(root, k, m, t)$ takes at most $(c_2 \cdot h + 2 \cdot c_1 \cdot h + c) = (2c_1 + c_2) \cdot h + c$ time, therefore satisfying $\mathcal{O}(h)$.