CSC263 Assignment 2

Yongzhen Huang, Xinyi Liu, Kewei Qiu

January 31, 2019

Question 1

Written by: Yongzhen Huang, Xinyi Liu Checked by: Kewei Qiu

a) `INCREASE_KEY(H, x, k)`
- If $k \leq x.key$ then do nothing. Otherwise set $x.key = k$
- while x is not *root* and $x.key$ is greater than $x.parent$ key, swap x and $x.parent$. Note that x is updated to a new position each time.

This algorithm is of time $\mathcal{O}(\log n)$ because if H has n nodes in total, the worst case is that H has only one tree and x is at the very bottom of the tree and increasing $x.key$ to k make it the biggest key in the tree so that we need to compare all the way from the bottom to the top of the tree, the root. Since this case has only one tree and the height is $\log_2 n$ (since if height k, total nodes is 2^k). Therefore, there is a total of $\mathcal{O}(\log n)$ comparisons and thus total time of $\mathcal{O}(\log n)$.

b) `DELETE(H, x)` Suppose x in tree B_k
- Increase $x.key$ to positive infinity i.e. `INCREASE_KEY(H, x, ∞)`. That is, increase $x.key$ to some value larger than any element in the heap. Now x is the root of a tree – say it B_k.
- Remove x from tree B_k. Since x is the root, by Lemma 2 on the slides, the rest of the B_k is a separate *binomial max heap* and call this collection T. The other trees belong to collection S.
- Do `UNION(T, S)` on the trees and a new binomial max heap is created.

By **a)**, we know that `INCREASE_KEY(H, x, ∞)` takes $\mathcal{O}(\log n)$ time. After removing x, the rest of B_k is a separate *binomial max heap*. Then `UNION(T, S)` takes $\mathcal{O}(\log n)$ time. Therefore the total is $\mathcal{O}(\log n)$ time.

Question 2

Written by: Yongzhen Huang, Kewei Qiu Checked by: Xinyi Liu

1. We make use of the data structure binomial heap. In particular, we use a structure which contains both binomial max heap and binomial min heap. The node stores the key and several kinds of pointers stored in each node: pointers to its parent, child, right sibling like a normal binomial heap (except with two copies).

What is different in this structure is that each node stores two pointers to its parents, two pointers to its children and two pointers to its right siblings. One is for max heap and one is for min heap. As a result, this data structure can have properties for both binomial max heap and binomial min heap.

2. The operations are performed in the following ways,

- `Insert(k)` When inserting, we insert it to both H_{max} and H_{min}. Since inserting into one heap takes $\mathcal{O}(\log n)$ time, inserting two still takes $\mathcal{O}(\log n)$ time as required.
- `ExtractMax()` This operation will be first performed on the H_{max} of the data structure. From class, such operation takes $\mathcal{O}(\log n)$ time. Then for its value in H_{min}, we can easily obtain the value with its

pointer to that heap. Then we perform `Delete(`H_{min}`), k` as in question 1 which takes $\mathcal{O}(\log n)$ time. So the total time is still $\mathcal{O}(\log n)$.

- `ExtractMin()` This follows directly from `ExtractMax()` except with the role of H_{max} and H_{min} reversed. Total time is still $\mathcal{O}(\log n)$.

- `Merge(D, D')` Since D, D' are both SuperHeaps, they both contain their own H_{max} and H_{min}. So we simply perform `Union(`H_{max}, H'_{max}`)` on the binomial max heaps together and separately `Union(`H_{min}, H'_{min}`)` on the binomial min heaps together. From class, the `Union()` operation takes $\mathcal{O}(\log n)$ time each, so the total run-time is still $\mathcal{O}(\log n)$.

Question 3

Written by: Yongzhen Huang Checked by: Kewei Qiu, Xinyi Liu

a. This is essential `Search(`$root, k$`)` along with counting the number of steps spent searching, namely,

- Initialize variable *path_len* to 0. If k equal to $key(root)$, return *path_len*. If k greater than $key(root)$, we look at $rchild(root)$ and perform PathLengthFromRoot($rchild(root)$, k) and return its return value. Otherwise if k is less than $key(root)$, we look at $lchild(root)$ and perform PathLengthFromRoot($lchild(root)$, k) and return its return value.

path_len $\leftarrow 0$;
if $key(root) == k$ **then**
  return 0;
else if $key(root) < k$ **then**
  path_len += PathLengthFromRoot($rchild(root), k$) + 1;
else
  path_len += PathLengthFromRoot($lchild(root), k$) + 1;
end
return *path_len*;

Algorithm 1: PathLengthFromRoot

This algorithm is in $\mathcal{O}(h)$ because we are only going downwards from the root and never going back up. At each node, the operations performed are only comparison and addition and these are constant time. So the maximum number of steps is the height and therefore $\mathcal{O}(h)$ as desired.

b. Consider the following,
- If one of k, m is $key(root)$ itself, then *root* has to be the common parent. So return $key(root)$. Then, if k, m are **both** greater than or less than the value of current input *root* (i.e. $key(root)$), we call `FCP(`$rchild(root), k, m$`)` or `FCP(`$lchild(root), k, m$`)`, respectively, and return the returned node. If one of k, m is greater than $key(root)$ and the other one is less than $key(root)$, then the current *root* is the only furthest common parent they have; in this case, simply return *root*.

if $key(root) == k$ *OR* $key(root) == m$ **then**
  return *root*;
else if $key(root) < k$ *AND* $key(root) < m$ **then**
  return FCP($rchild(root), k, m$);
else if $key(root) > k$ *AND* $key(root) > m$ **then**
  return FCP($lchild(root), k, m$);
else
  return *root*;
end

Algorithm 2: FCP

First, this algorithm makes 2 comparisons at each call. Then, it only traverses down the tree and never go up or start over. Therefore, the maximum number of times it can go is the height or the tree, namely, h. Therefore, the time complexity is $\mathcal{O}(h)$ as desired.

c. Consider the following,
First, we need to find the most common parent between k, m as in part b), $parent\_val$ = `FCP`$(root, k, m)$. Then we find the path length from the node with $parent\_val$ to each of k, m and sum them up. Finally, we compare this summed path length to t to determine true or false.

1. $parent\_node \leftarrow \text{FCP}(root, k, m)$
2.
3. $k\_path\_len \leftarrow \textsc{PathLengthFromRoot}(parent\_node, k)$
4. $m\_path\_len \leftarrow \textsc{PathLengthFromRoot}(parent\_node, m)$
5.
6. $total\_len \leftarrow k\_path\_len + m\_path\_len$
7.
8. return if $total\_len \leq t$

Algorithm 3: IsTAway

This algorithm takes $O(h)$ because line 1 `FCP` takes $\mathcal{O}(h)$, each of line 4, 5 takes $\mathcal{O}(h)$ time, and line 7, 9 take constant time. Since there are constant number of $\mathcal{O}(h)$, the total time is still in $\mathcal{O}(h)$ as desired.