Computer Science CSC263H — January 31, 2019
St. George Campus — University of Toronto

Solutions for Homework Assignment #2

**Answer to Question 1.**

**a.** Algorithm to *increase* the key of a given item $x$ in a binomial max heap $H$ to become $k$: If $k \leq x.key$ do nothing. If $k > x.key$ then set the key of $x$ to $k$ and then bubble up the item $x$ till it reaches its correct place in its binomial tree (i.e., while $k$ is greater than the key of the parent of $x$ in the tree, swap $x$ with its parent). Since the depth of the trees in $H$ is $O(\log n)$, the worst-case running-time of this algorithm is $O(\log n)$.

**b.** Algorithm to *delete* a given item $x$ from a binomial max heap $H$: First increase the key of $x$ to $+\infty$, this will cause $x$ to bubble up to the root of its binomial tree, then remove $x$ using the ExtractMax operation on $H$. By part (a) increasing the key of $x$ to $+\infty$ takes at most $O(\log n)$ time. We know (from class) that ExtractMax takes at most $O(\log n)$ time. So the worst-case running-time of this algorithm is $O(\log n)$.

**Answer to Question 2.** (Solution Sketch)

**a.** A SuperHeap $D$ is made of a binomial min heap $Q_{min}$ and a binomial max heap $Q_{max}$: each item $x$ has two copies one in $Q_{min}$ and one in $Q_{max}$, and the two copies are linked together with a double pointer. When we remove an item $x$ from one heap we also remove it from the other. When we insert an item $x$ into one heap we also insert it in the other. Merging two superheats consists of merging their binomial min heaps, and merging their binomial max heaps: these two merges are done independently, "in parallel".

**b.** Suppose $D$ is implemented with binomial min and max heaps pairs $(Q_{min}, Q_{max})$.

- An ExtractMin on $D$ is first performed on $Q_{min}$ with the usual binomial queue ExtractMin algorithm. Note that this will effectively remove the root $x$ of some tree of $Q_{min}$. After removing $x$ from $Q_{min}$, we use the link to locate the copy of $x$ in $Q_{max}$, and we then we delete it from $Q_{max}$ (note that this copy of $x$ is *not* necessarily a root of $Q_{max}$, so to delete it we use the Delete operation that we saw in the previous question).
- An ExtractMax on $D$ is performed in a symmetric way.
- To insert $x$, insert a copy of $x$ in each heap, and link both copies of $x$ with a double pointer.
- Finally, suppose SuperHeap $D'$ is implemented with binomial min and max heaps pairs $(Q'_{min}, Q'_{max})$. To merge $D$ with $D'$, just merge $Q_{min}$ with $Q'_{min}$ and $Q_{max}$ with $Q'_{max}$.

**Answer to Question 3.**

In each of the following algorithms, we assume that the BST is not empty (*root* is not NIL), does not contain duplicates, and every input key to the algorithm is present in the BST.

**a.** We do a simple search in the BST, while keeping track of the number of edges in the path from the *root* to $k$.

```
PathLengthFromRoot(root, k)
if k = key(root) then
    return 0
else if k < key(root) then
    return 1 + PathLengthFromRoot(lchild(root), k)
else
    return 1 + PathLengthFromRoot(rchild(root), k)
end if
```

The analysis of worst-case time complexity is identical to that of a simple search in a BST. In a nutshell, the recursive procedure only visits the nodes in the path from *root* to $node(k)$. Hence the worst-case time complexity is $O(h)$, where $h$ is the height of the BST.

**b.** The high level idea is as follows. If both $k$ and $m$ are smaller than *root*'s key, then the FCP of $k$ and $m$ should be in the left subtree of *root*, hence we recurse on the left subtree of *root*. Similarly, if both $k$ and $m$ are greater than root's key, we recurse on the right subtree of *root*. However, if $k < key(root) < m$ or $m < key(root) < k$, then *root* is the FCP of $k$ and $m$, hence we return *root*. The following recursive function computes the FCP of $k$ and $m$ in the BST rooted at *root*.

Without loss of generality assume $k < m$.

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

Again, the analysis of worst-case time complexity is identical to that of a simple search in a BST and hence the worst-case time complexity is $O(h)$, where $h$ is the height of the BST.

**c.** We first find the FCP of $k$ and $m$ in the BST rooted at *root*, using the FCP procedure. Let *parent* be the node returned by this procedure. We then compute the length of the path between $node(k)$ and $node(m)$, as follows: $PathLength = \textsc{PathLengthFromRoot}(parent, k) + \textsc{PathLengthFromRoot}(parent, m)$. Finally, we check whether *PathLength* is at most $t$.

IsTAway$(root, k, m, t)$
$parent \leftarrow$ FCP$(root, k, m)$
$PathLength \leftarrow \textsc{PathLengthFromRoot}(parent, k) + \textsc{PathLengthFromRoot}(parent, m)$
**if** $PathLength \leq t$ **then**
  **return** ***True***
**else**
  **return** ***False***
**end if**

We make one call to FCP and two calls to PathLengthFromRoot, and each call takes $O(h)$ time in the worst-case, where $h$ is the height of the BST. Hence, the worst-case time complexity is $O(h)$.