By: Jacob Chmura, Eric Koehli, Conor Vedova

Question 1. [1 MARK]

For the following two parts, let H be a max binomial heap, let n be the number of elements in H, let x be a node in H, and let k be the value of a key.

Part (1) [0 MARK]

```
1       def increase_key(x: Node, k: Key) -> Node:
2           """
3           Increase the key of a given item <x> in a max binomial heap <H> to
4           become <k>. Do not change anything if k <= x.key.
5           """
6           if k <= x.key:
7               return x                                 # do nothing
8
9           x.key = k
10          while x.parent is not None and x.key > x.parent.key:
11              # bubble up the tree swapping nodes with parent
12              x, x.parent = x.parent, x
13          return x
```

We first check to see if anything needs to be done. If $k \leqslant x.key$, then we return nothing on line 7 and the function is finished. Otherwise, we set x's key to the new key value k and start bubbling up the heap to restore the possibly violated max-heap property. If x is not the root, then we check if x's key is greater than it's parents key. If this condition is true, then we swap the nodes x with it's parent node. We know this algorithm runs in $\mathcal{O}(\lg(n))$ time since the height of H is at most $\lg(n)$, where n is the number of nodes in H. Thus the while loop iterates at most $\mathcal{O}(\lg(n))$ times.

Part (2) [0 MARK]

```
def delete(x: Node) -> None:
    """
    Delete the given node <x> from the max binomial heap <H>.
    """
    # boost x to the top of the max binomial heap
    while x is not root and x.key < x.parent.key:
        x = increase_key(x, x.parent.key + 1)

    # we're at a root of the forest now
    k = max(keys of all roots)
    x = increase_key(x, k)
    x.extract_max()
```

The high level idea for algorithm `delete` is to keep increasing the key of the given node x until it is at the top of the max binomail heap. Once node x has become one of the roots of the max binomial heap, then we can simply call `extractMax(H)` on the max binomial heap. This algorithm runs in the required $\mathcal{O}(\lg(n))$ time because the while loop iterates at most the height of the binomial heap ($\lg(n)$). Also, we know from class that `extractMax` runs in $\lg(n)$ time. Hence, delete takes $2 \cdot \lg(n)$, which is $\mathcal{O}(\lg(n))$ time.

Algorithm by: Jacob Chmura, Eric Koehli, Conor Vedova
Proof by: Jacob Chmura
Edited by: Eric Koehli

Question 2. [1 MARK]

Superheap

Superheap is a binomial max heap with one additional field. For every node A, node A also contains a pointer to the node with the minimum key in the subtree rooted at the node A. In this way, we are able to find the minimum value of the whole heap. The algorithms will be explained below

Algorithm Explanation

- **Extract max** Find the max of all of the roots in the heap. There are at most $\lg(n)$ roots. So, this step cannot take more than $\lg(n)$. Next, extract the max of this tree, and update the min pointer of the new root by choosing the min of the old pointer and the pointer of the other child of the original root. In this way, the new heap retains the Superheap structure. Together, there are no more than $2\lg(n)$ steps. So, this algorithm takes $\mathcal{O}(\lg(n))$.

- **Extract Min** This algorithm is structured in a similar way to Extract Max. However, we find the minimum value by finding the min of the keys of the nodes pointed to by all of the roots. Since there are at most $\lg(n)$ roots, this step takes at most $\lg(n)$ time. Next, apply a variation of the delete(x) function defined in 1b, where x is the minimum node of the heap. The only difference is that for every parent that the node x passes, we must update the min pointer to be the minimum of the min pointers of its new children[1]. Then, apply extract max to the tree containing the min node. Make sure to update the minimum pointers like we did in Extract Max defined above. Notice that delete is $\mathcal{O}(\lg(n))$. So, Extract Min is $\mathcal{O}(\lg(n))$.

- **Merge** Merge is performed in almost the same way as Merge of a binomial max heap. The only difference is that when combining two trees, not only must you find the new root with the max key of the two trees, but you must also take the min of the min pointers of the two roots of the two trees and assign this min pointer to the new root. Since we have only added a constant operation, Merge will be the same runtime as merge of the binomial max heap which is $\mathcal{O}(\lg(n))$.

- **Insert(x)** Create a Superheap with just x as the root of the only tree. Assign the min pointer of x to point to itself. Then apply merge(H, x), where x is the Superheap defined above. This algorithm is $\mathcal{O}(\lg(n))$ because creating the Superheap is constant time and merge is $\lg(n)$

Algorithm by: Jacob Chmura, Eric Koehli, Conor Vedova
Proof by: Conor Vedova
Edited by: Eric Koehli, Jacob Chmura

[1] the children of the parent are found after x and x.parent have been swapped

Question 3. [1 mark]

Part (1) [0 mark]

```
1   def path_length_from_root(root: Node, k: Key) -> int:
2       """
3       Given the <root> of a BST and a key <k>, return the length of the path
4       between root and node(k). Assume that the key k is in the BST.
5       """
6       if node(k) == root:
7           return 0
8
9       elif k > key(root):
10          return 1 + sum([path_length_from_root(rchild(root), k)], [])
11
12      else:
13          return 1 + sum([path_length_from_root(lchild(root), k)], [])
```

There are three paths to consider. The first path occurs on line 6 when the node with key k is equal to the root of the BST rooted at *root*. When this is the case, there is no edge between the node with key k and the node *root*, so the length is zero. The second path occurs if the passed in key k is greater than the key of the node rooted at *root*. When this occurs, we return 1 plus the sum of the algorithm `pathLenthFromRoot` with *root*'s right child passed and the same key k. This is correct because `pathLengthFromRoot` returns the path length of the *root*'s right child to the node with key k, but we need to add one to this value to include the edge connected to *root*. Lastly, the same line of reasoning applies when k is less than the key rooted at *root* and line 13 executes. The runtime of this algorithm is at most the height of the tree h rooted at *root* since we are only traversing the tree downward. Thus the time complexity is $\mathcal{O}(\lg(n))$.

Part (2) [0 mark]

```
1   def FCP(root: Node, k: Key, m: Key) -> Node:
2       """
3       Given the <root> of a BST and two distinct keys <k> and <m>, return the
4       FCP of k and m in the BST rooted at root. Assume that both k and m are
5       present in the BST.
6       """
7       x = root                                # the node to compare
8
9       # while our current key belongs to a common parent of k and m
10      while ((k < key(x) and m < key(x)) or (k > key(x) and m > key(x))):
11
12          # if k and m are to the left of node x, traverse left subtree
13          if k < key(x) and m < key(x):
14              x = lchild(x)
15
16          # if k and m are to the right of node x, traverse right subtree
17          else:
18              x = rchild(x)
19
20      return x
```

The algorithm works by starting at the root and traversing down the tree until we are in a position in which our comparison node x is sandwiched in between k and m. If we are at any point before reaching this point, this will be the furthest common parent of k and m. Until that point is reached, we traverse down the tree, traversing the side of the tree that contains both k and m. This algorithm achieves runtime linear to the height of the tree since the while loop iterates at most h times, where h is the height of the tree and every other operation takes constant time.

Part (3) [0 MARK]

```
def is_t_away(root: Node, k: Key, m: Key, t: int) -> bool:
    """
    Given the <root> of a BST, two distinct keys <k> and <m>, and a non-negative
    integer <t>, return true if the length of the path between node(k) and
    node(m) is at most t, and false otherwise. Assume that k and m are present
    in the BST.
    """
    fcp = FCP(root, k, m)        # locate the furthest common parent

    # compute the distance to root from each node and compare to t
    path_len = path_length_from_root(fcp, k) + path_length_from_root(fcp, m)

    return t == path_len
```

The algorithm first locates the furthest common parent between m and k. This will be the node that the longest path between m and k must go through. We can compute the longest path distance by taking the distance from m to the furthest common parent, and adding the distance from the furthest common parent to k. Each of these are handled by the algorithm from Part(1). Lastly, we compare this distance to t. Since each call to previous algorithms is $O(h)$, then the entire algorithm is also $O(h)$ by big-oh properties.

Algorithm by: Jacob Chmura, Eric Koehli, Conor Vedova
Proof by: Eric Koehli, Jacob Chmura, Conor Vedova
Edited by: Eric Koehli, Conor Vedova