Question 1.

Zijin Zhang wrote the solution to this question and Quan Xu read this solution to verify its clarity, correctness and correct some part of solution.

(a)
Consider the worse case scenario of a binomial max heap with n items, the maximum height of each individual binomial tree is at most $\mathcal{O}(\log n)$
The algorithm to increase the given key k has following steps.
1. Locate the item x in the binomial max heap and this will take constant time since the given x is a pointer.
2. Compare the k and $x.key$, there will be two cases:
(i) $k <= x.key \implies$ no operation needed
(ii) $k > x.key \implies$ we increase $x.key$ to k
For the worse case scenario, assume that $k > x.key$ and let $x.key = k$. The comparison and assignment will both take constant times.
3. The last step is to make sure that this binomial tree still follows max heap properties. Since $x.key$ is increased, we only need to check x and its ancestors. We will check x and its parent and do a swap if $x.key > parent.key$. Each comparison or swap operation will take constant times respectively. Since we need to do the check propagate up about its height(in worst case), we will do the comparison and swap at most $\mathcal{O}(\log n)$ times.
Therefore, the total worse case running time in $\mathcal{O}(\log n)$.

(b)
The algorithm to delete the given node x has following steps.
1. Locate the item x in the binomial max heap and this will take constant time since the given x is a pointer.
2. Make multiple swap to bubble up the node x until it become the root of that binomial tree. For now, this particular binomial tree does not follow the max heap properties since the root is smaller than its children, but this will be fixed after deleting the node. In worse case scenario, the maximum height of each individual binomial tree in an binomial max heap with n item is at most $\mathcal{O}(\log n)$ and each swap will take constant time. So the total running time for this step in worse case is $\mathcal{O}(\log n)$.
3. Delete the new root(node x) and split that binomial tree into a binomial max heap, denotes as H'. This step will take constant time.
4. Merge the new binomial max heap H' with the rest of the binomial trees left in the original binomial max heap H(if there is any). As we have proved in class, the $Union$ operation has the worse case running time $\mathcal{O}(\log n)$.

Each step above either take constant time or $\mathcal{O}(\log n)$; therefore, the total running time in worse case scenario should be $\mathcal{O}(\log n)$.

Question 2.

Quan Xu wrote the solution to this question and Zijin Zhang read this solution to verify its clarity, correctness and correct some part of solution.

Assume we want to store n keys to the SuperHeap.

Dsign of date structure $SuperHeap$:
Say that a $SuperHeap$ is the combination of one Min Binomial Heap and one Max Binomial Heap. The Min Binomial Heap store the same node as what store in the Max Binomial Heap(i.e. both Min Binomial Heap and Max Binomial Heap store the same node, but have different shape and order).

For example, let SH denote a $SuperHeap$, assume we want to store n element to SH. Let Max_Fn denote the Max Binomial Heap in SH and let Min_Fn denote the Min Binomial Heap in SH. It will store all the n element in both Min_Fn and Max_Fn. Min_Fn and Max_Fn share the same node(i.e. if a node n_0 in Min_Fn and there must be a same node n_0 in Max_Fn and both n_0 have the same pointer since they are actually the same node).

Addition Information:
For each node in $SuperHeap$, it stores a tuple. The first item of that tuple is a key and the second item is the pointer of this node(we can locate this node by the pointer).

High-Level Description of $merge(D,\ D')$:
(a) Union the Min Binomial Heap, D_{min} in D, with the Min Binomial Heap D'_{min} in D' by using the $Union(D_{min},\ D'_{min})$ operation of Binomial Heap.
(b) Union the Max Binomial Heap, D_{max} in D, with the Max Binomial Heap D'_{max} in D' by using the $Union(D_{max},\ D'_{max})$ operation of Binomial Heap.

Note: For each step, we use the $Union()$ operation of Binomial Heap to unite two Binomial Heap. Say we want to store n keys, then there will be n nodes in each Binomial Heap, and it will take at most $\mathcal{O}(logn)$ key-comparison for step (a) and step (b) respectively in worse-case running time. In total, $merge(D,\ D')$ takes $a\log n + b\log n \in \mathcal{O}(\log n)$ in worse-case scenario, where $a,\ b \in \mathbb{R}^{>=0}$.

High-Level Description of $insert(k)$:
(a) Make a new Binomial Heap K' that only contain node k. It can be considered as both Min Binomial Heap and Max Binomial Heap since it only have one node.
(b) Unite the Min Binomial Heap, Min_BH, in the $SuperHeap$ with K' by using $Union(Min\_BH, K')$ operation of Binomial Heap.
(c) Unite the Max Binomial Heap, Max_BH, in the $SuperHeap$ with K' by using $Union(Max\_BH, K')$ operation of Binomial Heap.

Note: Making a one-node Binomial Heap in step (a) takes constant time C. In step (b) and (c), assume there are n nodes stored in the Min Binomial Heap and Max Binomial Heap respectively. So using $Union()$ operation of Binomial Heap to unite the original Max/Min Binomial Heap with one-node Binomial Heap, K', has the worse-case running time $\mathcal{O}(\log n)$. One key-comparison takes constant time. So it would take at most $logn$ comparisons of Max/Min Binomial Heap respectively. In total, $insert(k)$ takes $C + a\log n + b\log n \in \mathcal{O}(\log n)$ in worse-case scenario, where $a,\ b \in \mathbb{R}^{>=0}$.

High-Level Description of $ExtractMax()$:
(a) Extract the maximum key of Max Binomial Heap, Max_BH, in $SuperHeap$ by using the $Extract\_Max()$ of Max Binomial Heap and get the pointer, max_ptr of this node n_{max}.
(b) Since we have the max_ptr, we can locate this node n_{max}. Then we will delete this node n_{max} from the Min Binomial Heap, Min_BH of the $SuperHeap$ by using $Delete(Min\_BH,\ n_{max})$ operation of Min Binomial Heap.

Note: In step(a), since $Extract\_Max()$ is a operation of Binomial Heap, its worse case running time is $\mathcal{O}(\log n)$ as we discussed in class and get the pointer of maximum node will take constant time C_1. In step(b), locate the maximum node in the Min Binomial Heap of the $SuperHeap$ takes constant time C_2 since we have its pointer. According to the textbook "CLRS" and lecture slides, deleting the node from Min Binomial Heap by using $Delete()$ operation of Binomial Heap has the worse-case running time $\mathcal{O}(\log n)$. Since Min Binomial Heap follows min-heap property, so the node have max priority locates at leaf. After deleting the max priority node, the Min Binomial Heap maintains min-heap priority. So in total, $ExtractMax()$ takes $(C_1 + C_2) + a\log n + b\log n \in \mathcal{O}(\log n)$ in worse-case scenario, where $a,\ b \in \mathbb{R}^{>=0}$.

High-Level Description of $ExtractMin()$:
(a) Extract the minimum key of Min Binomial Heap, Min_BH, in $SuperHeap$ by using the $Extract\_Min()$ of Min Binomial Heap and get the pointer, min_ptr of this node n_{min}.
(b) Since we have the min_ptr, we can locate this node n_{min}. Then we will delete this node n_{min} from the Max Binomial Heap, Max_BH of the $SuperHeap$ by using $Delete(Max\_BH,\ n_{min})$ operation of Max Binomial Heap.

Note: In step(a), since $Extract\_Min()$ is a operation of Binomial Heap, its worse case running time is $\mathcal{O}(\log n)$ as we discussed in class and get the pointer of minimum node will take constant time C_1. In step(b), locate the minimum node in the Max Binomial Heap of the $SuperHeap$ takes constant time C_2 since we have its pointer. According to the textbook "CLRS" and lecture slides, deleting the node from Max Binomial Heap by using $Delete()$ operation of Binomial Heap has the worse-case running time $\mathcal{O}(\log n)$. Since Max Binomial Heap follows max-heap property, so the node have min priority locatesat leaf. After deleting the min priority node, the Max Binomial Heap maintains max-heap priority. So in total, $ExtractMin()$ takes $(C_1 + C_2) + a\log n + b\log n \in \mathcal{O}(\log n)$ in worse-case scenario, where $a,\ b \in \mathbb{R}^{>=0}$.

Question 3.

Quan Xu and Zijin Zhang wrote the solution to this question. Zijin Zhang and Quan Xu read this solution to verify its clarity, correctness and correct some part of solution.
a)

```
1  PATHLENGTHFROMROOT(root, k):
2      if root has no child or key(root) == k:
3          return 0
4      else:
5          if k > key(root):
6              return  1 + PATHLENGTHFROMROOT(rchild(root), k)
7          else:
8              return 1 + PATHLENGTHFROMROOT(lchild(root), k)
```

Description:
- Line 2-3: When the root has no child and the key of roof is k, it should return 0, since there is no path between the root and the node with value k.
- Line 4-8: It not satisfy line 2-3, so we call PATHLENGTHFROMROOT() on the right child or left child, since it is a BST. And the height of BST rooted by rchild(root) [on line 6] and the height of BST rooted by the lchild(root) [on line 8] are one less than the height of the BST rooted by root. And they will eventually reach the Base Case(line 2-3). The "+1" is because we deep to the child of root from root itself, So there is a path of length 1 between root and its child.

Runtime:
- Basecase:(line 2-3), the two base cases take constant time, denote as C_1 times for each iterator of program, (where $C \in \mathbb{N}$).
- Recursion step(line 4-8): since it is a if *statement* and *else statement*, but no matter which statement it run, it will call PATHLENGTHFROMROOT() on the child of the root. And the thing that need to mention is that, it will go into either the if *statement* or *else statement*, since it is a BST. It will only call recursion on one of the child based on the if condition. Since the height of BST rooted by root is one larger than the height of BST rooted by child of root. So the height is always decrease by 1 on each call of PATHLENGTHFROMROOT(). And it eventually reaches the base case. For line 6-10, besides the recursion call, all other lines take constant time, denote as C_2.
Assume that the BST rooted by root has height h.
Therefore the recursive step take at most iterate h times.
Total runtime would be $(C_1 + C_2)h \in \mathcal{O}(h)$ in worst-case.

b)

```
1  FCP(root, k, m):
2      if k == key(root):
3          return root
4      if m == key(root):
5          return root
6      if (k < key(root) and m > key(root) or (m < key(root) and k > key(root))):
7          return root
8      else:
9          if k < key(root) and m < key(root):
10             return FCP(lchild(root), k, m)
11         else:
12             return FCP(rchlid(root), k, m)
```

Description:
Basecase:
- Line 2-3: If root contains value k, then root is the node parent that satisfy the rules. Since if we deep to the child of k's node, then it would not have k in the BST rooted by descendant of k but would have m in it.. So the node k

is the node parent that follows the rules. Then return root which contains k.
- Line 4-5: If root contains value m, then root is the node parent that satisfy the rules. Since if we deep to the child of m's node, then it would not have m in the BST rooted by descendant of m but would have k in it.. So the node m is the node parent that follows the rules.Then return root which contains m.
- Line 6-7: If the node contains k and the node contains m are located at the different side of root (one is on the left subtree, another one is in the right subtree.), then the root is the node parent of k and m.
Recursive stip:
- Line 8-12: It consider m's node and k's node are at same side of root (both at the right side or left side). Then we deep one level down to the child of root then call FCP() on lchild() if m's node and k's node in left subtree of root or call FCP() on rchild(root) if both m, k nodes in right subtree of root. It eventually will deep to the base case.

Runtime:
- Basecase(line 2-7): Basecase takes constant time, denote as C_1, in each iteration of program.
- Recursive step(line 8-12): it will run either the if *statement*(line 9) or *else statement*(line 11), and it will call FCP() in the children of root. And the thing that need to mention is that, it will go into either the if *statement* or *else statement*, since it is a BST. It will only call recursion on one of the child based on the if condition. The height of BST rooted by root is one larger than the height of the BST rooted by rchild(root) and the height of the BST rooted by lchild(root). So then height is always decrease by 1 on each call of FCP(). For line 8-12, besides the recursion call, all other lines take constant times, denote as C_2. Assume the BST rooted by root has height h. So it will eventually reaches the base case after h iteration.
Total runtime would be $(C_1 + C_2)h \in \mathcal{O}(h)$ in worst-case.

c)

```
1  IsTAway(root, k, m ,t)
2      length_k_to_root  = PATHLENGTHFROMROOT(root, k)
3      length_m_to_root = PATHLENGTHFROMROOT(root, m)
4      nodeparent_of_mk = FCP(root, k, m)
5      length_nodeparent_to_root = PATHLENGTHFROMROOT(root, nodeparent_of_mk)
6      result  = length_k_to_root + length_m_to_root − 2 ∗ length_nodeparent_to_root
7
8      return t == result
```

Description and runtime:
let $a, b, c, d \in \mathbb{R}^{>=0}$.
Assume the BST rooted by root has height h.

- line 2 & line 3 take ah times and bh times respectively, since we have proved in part (a) that the worse-case running time of $PATHLENGTHFROMROOT()$ is $\mathcal{O}(h)$ where h is the height of Binary Search Tree rooted by root.
- line 4 takes ch times, since worse-case running time of $FCP(root,\ k,\ m)$ is $\mathcal{O}(h)$ where h is the height of Binary Search Tree rooted by root.
- line 5 takes dh times, since the worse-case running time of $PATHLENGTHFROMROOT()$ is $\mathcal{O}(h)$ where h is the height of the Binary Search Tree rooted by root.
- line 6 & 8 takes constant tiome, denotes as C
Therefore, the total running time of $IsTAway(root, k, m, t)$ is $ah + bh + ch + dh + C = (a + b + c + d)h + C \in \mathcal{O}(h)$ in worse-case scenario.