Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

0/10

1.
Question a
Algorithm:

```
function searchValue(root, key) {
  if (root.key < key)
    return searchValue(root.right, key)
  else if (root.key > key)
    return search Value(root.left, key)
  else if (root.key == key)
    return root.value
}
```

Idea:
Not considering the priority part of PK-tree, searching a value corresponding to a specific key is in the same way as in a binary search tree. Since it's the same as binary search tree search, the time complexity is O(logn) and space complexity is also O(logn).

Question b
Algorithm:

```
// Suppose parent relationship is maintained in both
// rightRotate and leftRotate
function rightRotate(root) {
    pivot = root.left;
    root.left = pivot.right;
    pivot.right = root;
    root = pivot;
}

function leftRotate(root) {
    pivot = root.right;
    root.right = pivot.left;
    pivot.left = root;
    root = pivot;
}

function keyInsertion(root, item) {
    if (item.key < root.key) {
        if (root.left == null) {
            root.appendToLeft(item);
            return;
        } else {
            return keyInsertion(root.left, item);
        }
    } else if (item.key > root.key) {
```

Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

1/10

```
        if (root.right == null) {
            root.appendToRight(item);
            return;
        } else {
            return keyInsertion(root.right, item);
        }
    }
}

function balanceNode(root) {
    if (root.parent == null) return;
    if (root.parent.priority < root.priority) {
        // rotate root and its parent
        if (root.isLeftChild) rightRotate(root.parent);
        else leftRotate(root.parent);
        balanceNode(root);
    }
}

function insertItem(root, item) {
    // insert key
    keyInsertion(root, item);
    // rebalance tree
    balanceNode(item);
}
```

Idea:
First insert the new item to a place where the relationship of key is maintained. The position is unique in a binary search tree. Secondly rebalance related nodes so that priority relationship is also maintained.

Proof of correctness:
The key insertion is the same as key insertion in normal binary search tree and thus is trivial. Based on definition of rightRotate and leftRotate, the key relationship is maintained after the operation.
So the only thing to prove is that after balanceNode, priority relationship is maintained.
Precondition:

1. priority and key relationship is maintained in all parent-child pairs except of the newly inserted node and its parent

Postcondition:

1. priority and key relationship is maintained for every parent-child pairs

Proof:
Base case: n = 1, the contains only one element which is the root, the new element to insert will be append as left or right child of root. if the newly inserted element has higher priority

Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

than root, through right/left rotation, it would become root and original root would become its child. The priority relationship is maintained. Q.E.D
Hypothesis: After rotation, priority relationship on the path of target node n to the original inserted position is maintained, where 1(inserted position) < n < N(root)
Step: target n + 1 node, if (node[n+1].parent.priority <= node[n+1].priority), terminates, and based on hypothesis, the n + 1 node's children all has higher priority, while based on precondition, n + 1 node's parent all has lower priority.
if not terminates, right/left rotation will change the parent-child relationship of n + 1 node and its parent, making parent.priority < child.priority, and continue to next recursion until reaching the root. Q.E.D

Time complexity:
The binary search(keyInsert) has a time complexisty O(logn)
The balance tree has a time complexity O(logn) because on average a node's level is O(logn) and on the worst case we need to call balanceNode itself on all those node on the path from newly inserted node all the way to root where the each recursion of balanceNode takes constant time O(1).
The overall time complexity is O(logn)

Space complexity:
For each recursion, we allocate a fixed amount of memory, and thus the space complexity is the same as time complexity which is O(logn).

Question c
Algorithm:

```
function keyRemoval(root, key) {
    if (root == null) return;
    if (key < root.key) {
        return keyRemoval(root.left);
    } else if (key > root.key) {
        return keyRemoval(root.right);
    } else if (key == root.key) {
        // Do the removal
        if (root.left == null && root.right == null) {
            removeFromParent(root);
        } else if (root.left == null || root.right == null) {
            appendToParent(root); // append root's only child to root's
parent
        } else {
            min = root.right.findMin();
            replaceNode(root, min); // replace current node with min
node in right subtree
        }
        return root;
```

3/10

```
    }
}

function balanceChildren(root) {
    if (root.left.priority > root.priority && root.left.priority >=
root.right.priority) {
        rightRotate(root);
        balanceChildren(root);
    } else if (root.right.priority > root.priority && root.left.priority
< root.right.priority) {
        leftRotate(root);
        balanceChildren(root);
    }
}

function removeItem(root, key) {
    position = keyRemoval(root,key);
    balanceChildren(position);
}
```

Idea:
First remove the item as normal binary search tree which is trivial. In all 3 cases of removal of item in BST, only the third case would generate possible priority conflicts with removal position node's children. Balance those children nodes to rebalance the whole tree.

Proof of correctness:
for the third case in keyRemoval, the min node from right subtree, must has a lower priority than the item just removed. Suppose the new tree at the min node has height N.
Hypothesis: On the path from base node to node n, priority relationship is maintained.
Base case: N = 1, the tree has height 1, and thus has only 1 root node, satisfying the priority relationship.
Step: from base node to node n + 1, priority relationship is maintained.
If the n + 1 node has no children or both children's priority is bigger than n + 1 node's priority, based on definition, priority relationship is satisfied.
If the n + 1 node has children's priority bigger than n + 1 node's priority, make left/right rotation so that the n + 1 node position has a lower priority than both its children, and continue the process on original n + 1 node which has been rotated to a lower level. Since the n + 1 node position has the lower priority than both children, based on definition, the priority relationship is maintained. Q.E.D.

Similar to insertItem
Time Complexity:
O(logn)
Space Complexity:

Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

4/10

O(logn)

2.
Algorithm:

```
function MATCHPREFIX(str, prefix)
  for i=1 to prefix.length
    if prefix[i] < str[i]
      return 1
    else if prefix[i] > str[i]
      return -1
  return 0

function SEARCH(root, prefix)
  if root == NIL
    return
  if MATCHPREFIX(root.str, prefix) == 1
    SEARCH(root.left, prefix)
  else if MATCHPREFIX(root.str, prefix) == -1
    SEARCH(root.right, prefix)
  else
    SEARCH(root.left, prefix)
    PRINT(root.str)
    SEARCH(root.right, prefix)
  return
```

Idea: The strings are stored in a BST in lexicographical order, whereas a prefix specifies a lexicographical range in which all strings should be printed.
The algorithm is a combination of BST search and traversal. First, we find the first string in this lexicographical range, then do inorder traversal until the prefix doesn't match anymore.

Proof of Correctness:
This recursive algorithm's correctness can be proved with induction.

- **Base**: For a empty tree, the algorithm is correct as it doesn't print anything.
- **Hypothesis**: Suppose the algorithm is correct for all subtrees of tree T.
- **Step**: We need to prove that the algorithm is correct for T.
  Case 1: The string of T.root is lexicographically greater than the prefix. The algorithm then searches the left subtree of T, where all strings that matches the prefix lies.

Case 2: The string of T.root matches the prefix. The algorithm prints the T.root string, then searches both left and right subtree of T, because strings that matches are possible to be in either subtree.
Case 3: Opposite of Case 1. The string of T.root is lexicographically smaller than the prefix. The algorithm then searches the right subtree of T, where all strings that matches the prefix lies.
Thus, the algorithm is correct.

Proof of Termination:
This recursive algorithm's termination can be proved with induction.

- **Base**: For a empty tree, the algorithm is returns immediately.
- **Hypothesis**: Suppose the algorithm terminates on all subtrees of T.
- **Step**: We need to prove that the algorithm terminates for T.
  Case 1: The string of T.root is lexicographically greater than the prefix. The algorithm then searches the left subtree of T, which terminates.
  Case 2: The string of T.root matches the prefix. The algorithms prints the string at T.root, then searches both left and right subtree, which both terminate.
  Case 3: Opposite of Case 1. The string of T.root is lexicographically greater than the prefix. The algorithm then searches the right subtree of T, which terminates.

Thus, the algorithm terminates.

Time complexity:

1) Searching for the first node with prefix x takes O(lg(n)).
2) Do inorder traversal of BST where strings have prefix x takes O(k), because every node with prefix x is visited once, and the algorithm terminates when we reach a node whose prefix doesn't match.
3) Checking if a string matches prefix x takes at most O(x.length).

O(x.length*(k+lg(n))) = O(k+lg(n))

Space complexity:
The algorithm have O(1) space complexity, as no additional space is needed to search or traverse the BST.

3. Hashing
Double hashing equation:
h(i, k) = (h_1(k) + i * h_2(k)) mod |T|
where |T| is size of hash table, in our case 11

Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

6/10

Number	h_p	h_s
7	7	1
9	9	3
88	0	0
11	0	1
25	3	3
23	1	1
22	0	2
28	6	0
14	3	2
21	10	3

- slot #7 empty, insert 7 to slot #7
- slot #9 empty, insert 9 to slot #9
- slot #0 empty, insert 88 to slot #0
- slot #0 taken, try h(1, 11) = 1, insert 11 to slot #1
- slot #3 empty, insert 25 to slot #3
- slot #1 taken, try h(1, 23) = 2, insert 23 to slot #2
- slot #0 taken, try h(1, 22) = 2, taken, try h(2, 22) = 4, insert 22 to slot #4
- slot #6 empty, insert 28 to slot #6
- slot #3 taken, try h(1, 14) = 5, insert 14 to slot #5
- slot #10 empty, insert 21 to slot #10

Final table state:

Slot Number	Stored Value
0	88
1	11
2	23
3	25
4	22
5	14
6	28

Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

7	7
8	EMPTY
9	9
10	21

4. Greedy Algorithms

Algorithm:

```
function SCHEDULING()
  for $i$ from 1 to n
    $J_i$.value = $J_i.w_i / J_i.d_i$
  let $l[1...n]$ be a new array
  $l$ = Quicksort($J_i.value$)  decreasingly
  $w(S)$ is initialed with 0
  for $i$ from 1 to $n$
    $l[i].f_i = l[i-1].f_{i-1} + l[i].d_i$
    $w(S) = w(S) + l[i].f_i * l[i].w$
return $w(S)$
```

idea:
(1) . calculate the value (weight per time) of each job
(2) . using quicksort to sort jobs by decreasing value
(3) . calculate the finish time of the jobs in the sequence to get the *w*(S)

Proof of correctness:
(1). let *L* = {J1'... Jn'} be the solution generated by my algorithm, and let O = {J1''...Jn''} be an optimal feasible solution.
(2). Assume that O is not the same as L and there is a 2 consecutive elements in O in a different order than that in L. e.g. Ji' and Jj' in L, but Jj' and Ji' in O
(3).L.w(S) - O.w(S) = Jj'.w * Ji.d - Ji'.w * Jj'.d
Because of the greedy properties, Ji'.w / Ji'.d >= Jj'.w / Ji'd and L.w(S) - O.w(S) <= 0
The L.w(S) is no larger than O.w(S) and hence is an optimal solution.

Running time analysis:
(1). calculating the value cost O(n) time
(2). quicksort costs O(nlogn) time
(3). calculating finish time and adding total weighted time up cost O(n) time
Therefore the total running time is O(nlogn) time.

Team Info:
Chenlei Hu 1002030651
Zhaoning Kong 1004654288
Jiyang Zhang 1004654304

8/10

5. Dynamic Programming
Pick k to be a station planned to be build between i and j
Cost(i to j) = min(Cost(i to k) + Cost(k to j) + b(k), c(i, j))
if j = i + 1, by definition, Cost(i, j) = c(i, j)

Algorithm:

```
function MINCOSTROUTE($n, b, c$)
  let $cost$[1...n] be a new array
  let $prevStation$[1...n] be a new array
  let $route$ be an empty array
  $cost[1] = b[1]$
  $prevStation[1] = 0$
  for $i = 2$ to $n$
    $cost[i] = +\infty$
    for $j = 1$ to $i$-1
      if $cost[j] + b[i] + c[j, i] < cost[i]$
        $cost[i] = cost[j] + b[i] + c[j, i]$
        $prevStation[i] = j$
  $i = n$
  while $prevStation[i]$ != 0
    $route$.insert($prevStation[i]$)
    $i = prevStation[i]$
  route.reverse()
  return route
```

Idea:
The algorithm uses dynamic programming. Assume we have the minimal cost of constructing a pipeline from station 1 to i-1, we can get the minimal cost from 1 to i by searching for cost from any intermediate station j to i, in addition to the minimal cost from 1 to j, and choose the route with minimal cost.
Apart from the minimal cost, we have to get the route with minimal cost. For each station, we keep its previous station on its minimal cost route. Then, from the last station, we can trace back to the first station and get the route.

Proof of Correctness
The correctness of the algorithm can be proven with strong induction.

- **Base**: cost[1] = b[1], which is the associate cost to build the station at 1. Because this is the only possible way, the cost is minimal. prevStation[1] is 0, because there is no previous stations.
- **Hypothesis**: Suppose cost[1….i-1] is the minimal cost to construct a pipeline from 1 to 1….i-1, and prevStation[i] indicates the previous station on the minimal cost route to station i.
- **Step**: To construct the pipeline from station 1 to i, we can construct the pipeline from 1 to any intermediate station j, and then from j to i, which costs cost[j] + b[j] + c[j, i]. If

9/10

we traverse all possible j and select the one with minimal cost, such cost is minimal, since cost[j] is the minimal cost from 1 to j, as we supposed. Then, we store the index of the previous station on the minimal cost route to i in prevStation[i].

Thus, the algorithm is correct.

Proof of Termination

The algorithm terminates as long as we prove both the two loops terminates.

1. Proof that the first loop terminates:

The termination of the first loop can be proved with induction.

- **Base**: When i = 1, we assign b[1] to cost[1].
- **Hypothesis**: Suppose loop i-1 terminates.
- **Step**: We need to show that loop i terminates. Within the loop, the algorithm traverses the cost array to find minimum, which takes O(i) time, and then terminates.

Thus, the first loop terminates.

2. Proof that the second loop terminates:

The termination of the second loop can be proved with contradiction.

Assume towards contradiction that this loop does not terminate by keep assigning prevStation[i] to i. Because prevStation[i] != i, both prevStation[i] > i and prevStation[i] < i have to exist to create such a cycle.

However, it is impossible that prevStation[i] > i, because the previous station's index is always less than that of the current station, which is a contradiction.

Thus, the second loop terminates.

Time complexity

The first loop takes: O(1)+O(2)+....+O(n) = O(n^2)

The second loop takes: O(k), where k is the number of stations passed, and k < n.

Thus, the time complexity of the algorithm is O(n^2) +O(k) = O(n^2).

Space complexity

The algorithm uses three arrays:

cost: O(n)

prevStation: O(n)

route: O(k), where O(k) is the number of stations passed, and k < n.

Thus, the time complexity of the algorithm is O(n) + O(n) + O(k) = O(n)