Assignment 3

Jasraj Singh Johal | 400434346 | johalj11

Question 1: Binary Search

SymbolTable is defined with arrays for keys and values, and an integer n representing the length of the arrays.

The binarySearch function is used to find the index of a key in the ordered array.

The delete function utilizes binary search to find the index of the key and then shifts the elements to delete the key from the array.

This implementation assumes that keys and values are stored at corresponding indices in their respective arrays. Additionally, it uses 1-based indexing for the array length (n).

```
Type SymbolTable:
    keys: array of any
    values: array of any
    n: length of array

function binarySearch(st, key):
    low <- 0
    high <- st.n - 1

    while low <= high:
        mid <- (low + high) // 2
        if st.keys[mid] == key:
            return mid  # Key found at index mid
        else if st.keys[mid] < key:
            low <- mid + 1  # Adjust low to search in the right half
        else:
            high <- mid - 1  # Adjust high to search in the left half

    return -1  # Key not found

function delete(st, key):
    index <- binarySearch(st, key)

    if index != -1:
        for i <- index to st.n - 2:
            st.keys[i] <- st.keys[i + 1]  # Shift keys to the left
            st.values[i] <- st.values[i + 1]  # Shift values to the left

        st.n <- st.n - 1  # Reduce the size of the symbol table
```

Question 2: Binary Search Trees

```
# Assumption made - The left subtree has keys less than the root, and the right
subtree has keys greater than or equal to the root.

# Function to check if a given tree is a full or perfect binary search tree
function check_full_perf(t):
    if not isBST(t):
        return 0
    if isfullBST(t):
        if isPerfectBST(t, findDepth(t)):
            return 3
        return 2
    return 1

# Function to check if a tree is a Binary Search Tree (BST)
function isBST(root):
    if root == Null:
        return True
    left_tree <- root.left
    right_tree <- root.right
    if left_tree == Null and right_tree == Null:
        return True
    if left_tree == Null and right_tree != Null:
        return isBST(right_tree) if (right_tree.key >= root.key) else False
    if left_tree != Null and right_tree == Null:
        return isBST(left_tree)
    return isBST(left_tree) and (left_tree.key < root.key < right_tree.key)

# Function to check if a tree is a full binary search tree
function isfullBST(root):
    if root == Null:
        return False
    if root.left == Null and root.right == Null:
        return True
    if root.left != Null and root.right != Null:
        return isfullBST(root.left) and isfullBST(root.right)
    return False

# Function to find the depth of a tree
function findDepth(root):
    if root is null:
        return 0
    leftDepth = findDepth(root.left)
    rightDepth = findDepth(root.right)
```

```
    depth = max(leftDepth, rightDepth) + 1
    return depth

# Function to check if a tree is a perfect binary search tree
function isPerfectBST(root, d, level <- 0):
    if (root is None):
        return True
    if (root.left is None and root.right is None):
        return (d == level + 1)
    if (root.left is None or root.right is None):
        return False
    return (isPerfectBST(root.left, d, level + 1) and
            isPerfectBST(root.right, d, level + 1))
```

Question 3: Heaps

```
# Pseudocode for a MinMaxPriorityQueue datatype using two heaps

# Datatype definition
datatype MinMaxPriorityQueue:
  min_heap: minPQ  # min_heap is a min priority queue
  max_heap: maxPQ  # max_heap is a max priority queue

# Method to insert a value into the priority queue
method insert(pq, value):
  pq.min_heap.insert(value)
  pq.max_heap.insert(value)

# Method to delete the minimum value from the priority queue
method delete_min(pq):
  if pq.min_heap.is_empty():
    return None
  return pq.min_heap.delete_min(pq)

# Method to delete the maximum value from the priority queue
method delete_max(pq):
  if pq.max_heap.is_empty():
    return None
  return pq.max_heap.delete_max()

# Method to find the minimum value in the priority queue
method find_min(pq):
  if pq.min_heap.is_empty():
```

```
    return None
  return pq.min_heap.find_min()

# Method to find the maximum value in the priority queue
method find_max(pq):
  if pq.max_heap.is_empty():
    return None
  return pq.max_heap.find_max()
```

Insert:

- The **insert** operation inserts a value into both the **min_heap** and **max_heap**.
- Since the heaps maintain their properties (min-heap and max-heap), the height of each heap is logarithmic in the number of elements.
- Therefore, inserting an element into a heap takes logarithmic time.
- Since we perform two insert operations, the overall complexity remains logarithmic.

Delete Min:

- The **delete_min** operation removes the minimum value from the **min_heap**.
- Deleting the minimum element from a heap takes logarithmic time as the heap structure needs to be maintained.
- Therefore, the **delete_min** operation has a logarithmic time complexity.

Delete Max:

- The **delete_max** operation removes the maximum value from the **max_heap**.
- Deleting the maximum element from a heap also takes logarithmic time due to the heap structure.
- Thus, the **delete_max** operation has a logarithmic time complexity.

Find Min:

- The **find_min** operation retrieves the minimum value from the **min_heap**.
- Accessing the minimum element in a min-heap takes constant time, as the minimum element is always at the root.
- Therefore, the **find_min** operation has a constant time complexity.

Find Max:

- The **find_max** operation retrieves the maximum value from the **max_heap**.
- Accessing the maximum element in a max-heap also takes constant time, as the maximum element is at the root.
- Thus, the **find_max** operation has a constant time complexity.

Question 4: Hash Tables

```
Node:
    key: any type
    value: any type
    next: Node
```

```
SymbolTable:
    t: array of Nodes, initially null
    n: number of Nodes in t

keys(st):
    key_list <- []  # initialize an empty array to store keys
    for i in st.t:  # loop through the array of nodes in the Symbol table
        if i == Null:  # skip empty indexes
            continue
        current <- i  # loop through the linked list at the current index of the
hash table
        while current != None:
            key_list.append(current.key)  # append the keys to the key list
            current <- current.next  # move to the next node in the linked list
    return key_list
```