Ron Cox

HW9
605.202.81
Data Structures

1. Comparing the Efficiency of Sequential Search on Ordered and Unordered Tables

1(a). No Record with the Key Target is Present

- **Unordered Table**:
  - **Worst-case Time Complexity**: O(n)
  - You have to search through all n elements to determine that the key is not present.
- **Ordered Table**:
  - **Worst-case Time Complexity**: O(n)
  - Even though the table is ordered, a sequential search will still require scanning all elements to determine the key is not present.
- **Conclusion**: Both ordered and unordered tables have the same worst-case time complexity of O(n) when the target key is not present.

1(b). One Record with the Key Target is Present and Only One is Sought

- **Unordered Table**:
  - **Average-case Time Complexity**: O(n/2)=O(n)
  - On average, you will find the key after searching through half of the table.
- **Ordered Table**:
  - **Average-case Time Complexity**: O(n/2)=O(n)
  - Even though the table is ordered, sequential search does not benefit from this ordering, so it also takes on average n/2 comparisons.
- **Conclusion**: Both ordered and unordered tables have the same average-case time complexity of O(n)) when searching for a single key.

2. Sequential Search with Multiple Records with the Same Key

2(a). Finding Only the First Record with the Key Target

- **Unordered Table**:
  - **Average-case Time Complexity**: O(n/2)=O(n)
  - You search through the table until you find the first occurrence of the key.
- **Ordered Table**:
  - **Average-case Time Complexity**: O(n/2)=O(n)

    - Even though the table is ordered, finding the first occurrence of the key still requires scanning sequentially through the elements.
- **Conclusion**: Both ordered and unordered tables have the same average-case time complexity of O(n) for finding the first occurrence of the key.

2(b). Finding All Records with the Key Target

- **Unordered Table**:
    - **Worst-case Time Complexity**: O(n)
    - You need to search through all elements to find all occurrences of the key.
- **Ordered Table**:
    - **Worst-case Time Complexity**: O(n)
    - Even though the table is ordered, sequential search will still require checking all elements to find all occurrences of the key.
- **Conclusion**: Both ordered and unordered tables have the same worst-case time complexity of O(n)O(n)O(n) for finding all occurrences of the key.

3. Delete Method for a Binary Search Tree (BST)

Pseudo-code for `delete(key1, key2)`:

```
function delete(key1, key2)
      function deleteNode(root, key)
      if root is null
            return null
      if key < root.key
            root.left = deleteNode(root.left, key)
      else if key > root.key
            root.right = deleteNode(root.right, key)
      else
            if root.left is null
                  return root.right
            else if root.right is null
                  return root.left
            minLargerNode = findMin(root.right)
            root.key = minLargerNode.key
            root.right = deleteNode(root.right, minLargerNode.key)
      return root

      function findMin(node)
      while node.left is not null
            node = node.left
      return node
```

```
function rangeDelete(root, key1, key2)
if root is null
      return null
if root.key > key2
      root.left = rangeDelete(root.left, key1, key2)
else if root.key < key1
      root.right = rangeDelete(root.right, key1, key2)
else
      root.left = rangeDelete(root.left, key1, key2)
      root.right = rangeDelete(root.right, key1, key2)
      root = deleteNode(root, root.key)
return root

root = rangeDelete(root, key1, key2)
return root
```

This method recursively traverses the tree, deletes nodes whose keys fall within the range `[key1, key2]`, and restructures the tree accordingly.

4. Delete a Record from a B-Tree of Order n

Pseudo-code for `delete(key)` in a B-Tree:

```
function delete(key)
      node = findNode(key)
      if node is a leaf
      delete key from node
      if node underflows
            fixUnderflow(node)
      else
      predecessor = findPredecessor(key)
      swap key with predecessor
      delete(predecessor)

function fixUnderflow(node)
      if node has a sibling with more than the minimum number of keys
      borrow a key from sibling
      else
      merge node with a sibling
      if parent underflows
```

```
            fixUnderflow(parent)

function findNode(key)
      // Traverse the B-Tree to find the node containing the key

function findPredecessor(key)
      // Find the predecessor key in the subtree
```

This method first locates the node containing the key, then handles the deletion depending on whether the node is a leaf or an internal node. It also ensures that the B-Tree properties are maintained after deletion, by fixing underflows through borrowing keys or merging nodes as necessary.