LAB 1 - Data Structures and Algorithms

PREPARED BY

Ahmed Akram Ahmed Shawky -18010056

Rana Ayman Hussien -18010662

Mostafa Ahmed Abd El-hamed - 18011774

Mohamed Mohamed Mostafa Radwan -18011596

Amr Ayman Ibrahim Momtaz Ibrahim Momtaz - 18011178

Time Analysis

Part 1- Red Black Tree

getRoot():

It will take O(1) because it will return the abstract variable root

isEmpty():

It will take O(1) because it will check if the root was nill node then it is an empty tree else it is not

clear():

“clear” function run in O(1), setting the root pointer to Nil node

search(**Comparable key**):

“search” function run in O(lg n) :

Search technique:

1-Comparing the given key with a root pointer’s key.
2 If greater then compare with right pointer’s and consider it the new root and repeat.
3-If smaller Then compare with the left pointer and do the same in the right pointer until finding the given key or not.

Analysis:

1-According to that technique in search , the algorithm checks only one path of “RBTree” data structure.
2-From “RBTree” definition, it’s a **balanced BST,** So that it has height at most 2lg(n+1).

3-From (1) and(2) searching function runs in O(lg n) in the worst case.

contains(**Comparable key**):

“contains” function run in O(lg n): it follows the same technique of search function.

insert(comparable key,object value):

*Insertion function run in O(lg n):

Insertion technique:

1-It applies a search to find the appropriate position for this node as if it was a binary search tree which will take O(lg n)

2-after that applies some operations that help to make the tree balanced again after the the insertion which are :

Left rotation and right rotation which it takes O(1)

Analysis:

1. To find the place to insert takes the height of the tree, or O(lg n).
2. To add the node is O(1).
3. To fix possible double reds, a rotation is O(1).
4. Worst case, the double red can cascade all the way to the root = lg n

   Therefore, insertion = o(lg n)+o(1)+O(lg n) = O(lg n)

delete(T key):

*delete function run in O(lg n):

Delete technique:

1-It applies a search function to find the key which wants to be deleted : takes O(lg n) .

2-After the deletion it applies some operations to maintain the tree balanced which are:

-**“left rotation” , “right rotation”** and **“get sibling of node”** take O(1)

- **“get the successor for a node”** takes O(lg n)

Analysis:

1. Finding the node to delete plus finding the successor is proportional to the height of the tree, so it is O(lg n).
2. The swapping and deletion is O(1).
3. Each individual fix (rotation left and right , find sibling and successor) is O(1).
4. In the worst case, a double-black may get passed up to the root thus it is O(lg n).

Therefore, the worst case cost of deletion is O(lg n).

Part 2- Treemap

ceilingEntry(T key):

This function calls the function searchCeilingKeyEntry(INode,T,Map.Entry<T,V>). It sends to it the parameters such that the root is the tree node and the key is the key sent to it and the successor as null. The second function works as follows: if the root element key is smaller than the key sent. Then we go right. Else if the root key element is bigger then we go left AFTER UPDATING the current successor to the current value (because maybe there could be another entry with a key which is smaller than the successor found but bigger than the key we are searching the ceiling for). Else if the key is the same as the root element key then we return this entry immediately without the need to carry on searching.

Time complexity : O(lg(n)). We go right and left till we reach the leaves. The best case is O(1) if the root of the tree is the key which we are searching the ceiling for.

ceilingKey(T key):

It is exactly the same as the previous function except that it returns the Key not an entry. Same algorithm is applied with everything. And hence the same complexity applies. (The helper function in this case is searchCeilingKey(INode,T,T).

clear():

This function just calls the function clear() which is in the RedBlackTree itself which just sets the root to nil node. So the old root will have no references for it which will be collected by the garbage collector and succevily till the whole tree is removed (freed)

containsKey(T key):

This function searches for a specific key if it is in the tree it returns true. It uses the helper function searchKey(INode,T) which searches for the key in the map by sending the tree root to it and the key we are searching for. It compares the key sent to the root we are at. If the key is bigger then we go right else we go left. If found we return true. If we reach the nill node then the key is not found so we return false (Binary Search).

Time complexity : O(lg(n)) because we just go left and right we don't need to traverse the whole tree to search for the key. And we have the best case O(1) if the root key equals the key sent(best-case scenario).

containsValue(V value):

It searches for a specific value if it exists in the tree then we return true. Otherwise we return false. We use a helper function searchValue(INode,V) where we send the tree root and the value we are searching for. The function searches in all the tree nodes for the specific value because there is NO RELATION between the key and the value. It searches left and right. If it is found in the left OR the right then it returns true. And **SHORT CIRCUIT EVALUATION** occurs so the program won't continue searching since one returned true.

Time Complexity: O(n) because we are searching in the whole tree nodes for the value. And we still have the same best-case scenario where the tree root contains the value which is O(1).

<u>entrySet():</u>

This function returns a set of entries in ascending order. It uses the helper function getEntrySet(INode,HashSet). The function just performs the in-order traversing for the tree :

- Go Left
- Visit the node (Add it to the HashSet)
- Go Right

Time Complexity : O(n) since we are visiting the whole tree nodes. We don’t have either average or best case. They are all equal to O(n)

<u>firstEntry():</u>

This function returns the first entry associated with the least key in the map. It uses the helper function searchFirstEntry(INode) which searches for the first entry by just traversing left all the way down to get the least key associated entry.

Time Complexity : O(lg(n)) Because we are going all the way down. The best case is still O(lg(n)) = [½ lg(n)] if the tree is right biased (The right subtree contains maximum numbers of red nodes. And the left subtree contains no red nodes).

<u>firstKey():</u>

This function is exactly the same as the previous one except that it returns the key itself instead of the whole tree entry. Hence it uses a different helper function which is searchFirstKey(INode).

The complexity of this function is the same as this previous one (exactly).

<u>floorEntry(T key):</u>

This method traverse the whole tree looking for the greatest key less than or equal to the given key and return the Entry associated with that key.

Time complexity is : O(logn)

<u>floorKey(T key):</u>

This method traverse the whole tree looking for the greatest key less than or equal to the given key and return that Key.

Time complexity is : O(logn)

<u>get():</u>

This method traverse the whole tree looking for specified key and return its value

Time complexity is : O(logn)

headMap(T toKey):

It iterates over all entry sets and looks for keys less than the given key then puts its entry in a new ArrayList.

Time complexity is : O(n)

headMap(T toKey, boolean inclusive):

It iterates over all entry sets and looks for keys less than or equals to the given key then puts its entry in a new ArrayList.

Time complexity is : O(n)

keySet():

This method traverse every node in the Tree to collecting all keys

Time complexity is : O(n)

lastEntry():

This method traverse all right nodes looking for the last node to return its value

Time complexity is : O(logn)

lastKey():

This method traverse all right nodes looking for the last node to return its key

Time complexity is : O(logn)

<u>pollFirstEntry():</u>

This method traverse all left nodes looking for the least node to delete it and return its Entry

Time complexity is : O(logn)

<u>pollLastEntry():</u>

This method traverse all right nodes looking for the last node to delete it and return its Entry

Time complexity is : O(logn)

<u>put():</u>

This method Insert key and its value in its correct place in the tree

Time complexity is : O(logn)

<u>putAll():</u>

This method copies all of the mappings from the specified map to the tree

Time complexity is : O(n)

<u>remove():</u>

This method traverse the tree looking for specified key to delete it

Time complexity is : O(logn)

size():

This function returns the size of the tree by visiting all its nodes using the helper function getSize(INode). If the visited node is neither nil nor null then it increments the counters and it visits both left and right children.

Time Complexity : The function is O(n) because we need to visit all nodes to get the tree size. There is no average nor best case because they are all O(n).

values():

It iterates over all entry sets and stores their values in a new List<V> then returns this list.

Time complexity is : O(n)