Alexandria University,

Faculty of Engineering,

Computer and Systems Engineering Dept.

DataStructue2

*Assignment2*

Implementing Red Black Tree

&

Tree Map Interface

Names:

1. Ziad Hisham Ali Abdel-Hameed. (19)
2. Youssef Ahmed Sayed Gabriel. (61)
3. Problem Statement:

Implementing red black tree & implement tree map using red black tree

1. Main Modules ”Classes” :

* **Node**

Implement the operations that done on the node in the redBlackTree

* setParent
* getParent
* setLeftChild
* getLeftChild
* SetRightChild
* getRightChild
* setValue
* getValue
* setColor
* getColor
* **RedBlackTree**

Implement the operation that can be done on the redBlackTree

* getRoot
* isEmpty
* clear
* search
* contains
* insert
* delete
* Left\_Rotate
* Right\_Rotate
* RB\_Insert\_Fixup
* Delete\_Fixup
* Minimum
* Maximum
* Transplant
* Find
* findCeil
* findFloor
* **TreeMap**

Implement the treemap using redBlackTree

* ceilingEntry
* ceilingKey
* clear
* containsKey
* containsValue
* entrySet
* firstEntry
* firstKey
* floorEntry
* floorKey
* get
* headMap
* keyset
* lastEntry
* lastKey
* pollFirstEntry
* pollLastEntry
* put
* putAll
* remove
* size
* values
* printInorder

1. Assumptions :

* Node class was has no constructor & has 6 private variables

‘INode parent’ , ‘INode leftChild’ , ‘INode rightChild, ‘T key’,

‘V value, ‘Boolean color

* RedBlackTree class has constructor that has empty parameter that make two Nodes dummy & root , then set their parent, leftChild, rightChild, Color & Key

dummy node : parent, leftChild, rightChild & Key are null

parent node parent, leftChild, rightChild & Key are new Node

Has three private variables ‘INode root’ , ‘INode dummy’, ‘int size’

* TreeMap class has no constructor

Has five private variables ‘IRedBlackTree rbt’

, ‘Arraylist headMapArr’ , ‘Set<T> keys’ , ‘Set<T> values’

, ‘Set<T,V> entries

1. Algorithms :

* **Search(Comparable key)**

Take the passed key, then a make object of type node and find that node in the redBlackTree if it found return it

* **Contains(Comparable key)**

Take the passed key, then find the Node of this key in the redBlackTree if it found return true

* **Insert(Comparable key, Object value)**

Make a Node pointer that point to the root ,if the passed key < temp.key, then go in left path & if the passed key > temp.key , then go to right path but if passed key = temp.key, just update the key of the node , in each case RB\_Insert\_FixUp is called to maintain the proprieties of the redBlackTree

* **RB\_Insert\_FixUp** **(INode z)**

While the parent has red color, there are two cases

1. If z.parent = z.parent.parent.leftChild , if uncle != null , then z = z.parent.parent , if z = z.parent.rightChild ,then leftRotate at z else rightRotate at z.parent.parent
2. If z.parent = z.parent.parent.rightChild , if uncle != null , then z = z.parent.parent , if z = z.parent.leftChild ,then rightRotate at z else leftRotate at z.parent.parent

* **delete(Comparable key)**

find the node that has the passed key if node.leftChild = null , then

transplant(node, node.rightChild) .. else if node.rightChild = null , then

transplant(node,node.leftChild) ..else get the minimum at node.rightChild

if minimum.parent = node, then set mimimium.rightChild the parent of

of the minimum else transplant (minimum, minimum.rightChild)

Eventually, transplant (node ,minimum) and if the minimum.color = black

Call DeleteFixUP(minimum.rightChild)

* **DeleteFixUp (INode node)**

While the passed node isn’t the root and color=black there are two cases

1. node = node.parent.leftChild , if node.parent.rightChild.color = red

, then set color black and leftRotate(node.parent) .. if node.parent.rightChild and left child are black , then set color red .. else if rightChild.color = black , then rightRotate(node.parent.rightChild) .. if node.parent.rightChild.color = red , then node.parent.rightChild.rightChild.color = black and leftRotate(node.parent)

1. node = node.parent.rightChild , if node.parent.rightChild.color = red, then set color black and rightRotate(node.parent) .. if node.parent.leftChild and right child are black , then set color red .. else if leftChild.color = black , then leftRotate(node.parent.leftChild) .. if node.parent.leftChild.color = red , then node.parent.leftChild.leftChild.color = black and rightRotate(node.parent)

* **Transplant(INode x, INode y)**

If x.parent = null , then root = y .. elseif x =x.parent.leftChild , then x.parent.setLeftChild(y) .. else x.parent.setRightChild(y) , Eventually y.setParent(x.parent)

* **FindCeil(Comparable key)**

Pointer Node x point to the root While x != null , if x.key = key , then return x .. elseif x.key < key , then x =x .rightChild .. else x.key > key , then x =x .leftChild , Eventually find the successor of x

* **FindFloor(Comparable key)**

Pointer Node x point to the root While x != null , if x.key = key , then return x .. elseif x.key < key , then x =x .rightChild .. else x.key > key , then x =x .leftChild , Eventually find the predecessor of x

* **PrintInorder(INode node, Boolean less, Boolean inclusive, Comparable toKey, Collection arr)**

It’s a recursive function , base case: if node = null

First recursive on left subtree by calling printInorder(node.leftChild, less, inclusive, toKey, arr)

If less & node.key < tokey OR inclusive & node.key = tokey ,then push in arr the node and if not less , then push in values node.value

Eventually recursive on right subtree by calling printInorder(node.rightChild, less, inclusive, toKey, arr)

1. Sample Runs:

