|  |  |  |
| --- | --- | --- |
| Tree |  |  |
| Root | Node |  |
| numOfLeaf | Int |  |
|  |  |  |
| **Node** |  |  |
| parent | Node |  |
| rChild | Node |  |
| mChild | Node |  |
| lChild | Node |  |
| key | <K> | Internal node->key = max key of descendants  Leaf node->key |
| value | <L> | Internal node->value = sum of descendants  Leaf node -> value |
| numOfDesc | Int | Number of descendants |
| numOfChild | Int | Number of children |

Extreme Cases:

sumValuesInterval

* If key 1 > key 2 SOLUTION: if statement
* If there are no keys found in the given range. Needs to be outputted null. Solution: Save last highest key that was checked. If they are the same, return null.

/\*  
\* The {@code BalancedTree} represents an 2-3 Tree with some additional feature  
\* to it to be able of handling the Runtime demands.  
\* \*/  
  
public class BalancedTree<K extends Key,V extends Value> {  
  
 /\*Root of the tree is a Node\*/  
 private Node<K, V> root;  
 /\*Number OF Leaf in the tree, helps in insertaion method\*/  
 private int numOfLeaf;  
  
 /\*Constructor for tree, default key and value is null \*/  
 public BalancedTree() {  
 this.root = new Node<K, V>(null, null);  
 this.numOfLeaf = 0;  
 }  
  
 /\*Helper Node dataType for {@code BalanceTRee}, represents Node for the tree\* \*/  
 private class Node<K extends Key, V extends Value> {  
 private K key;  
 private V value;  
 private Node<K, V> parent;  
 private Node<K, V> rChild;  
 private Node<K, V> mChild;  
 private Node<K, V> lChild;  
 private int numOfLeaf;  
 private int numOfChild;  
  
 private Node(K key, V value) {  
 this.key = key;  
 this.value = value;  
 this.numOfLeaf = 0;  
 this.numOfChild = 0;  
 this.parent = null;  
 this.rChild = null;  
 this.mChild = null;  
 this.lChild = null;  
 }  
 }  
  
 */\*\*  
 \** ***@param*** *newKey : new Key for insertion, must have same data type as K(tree data type).  
 \** ***@param*** *newValue : new Value for insertion, must have same data type as V(tree data type).  
 \* Method insert, copy the {****@code*** *newKey} and it's {****@code*** *newValue} to the tree in log(n)  
 \* time complexity (n is the number of elements store in the current tree).  
 \* Note : insert method assume that all key are unique.  
 \*/* public void insert(K newKey, V newValue) {  
 Node<K, V> z = new Node<K, V>((K) newKey.createCopy(), (V) newValue.createCopy());  
 z.numOfLeaf = 1; //Leaf has 1 leaf "under" him.  
 /\*Extreme case, tree has 1 leaf or none (this case occur when the tree just been Initialized). \*/  
 if (this.numOfLeaf < 2) {  
 initializedInsert(z);  
 return;  
 }  
 /\*If the method got to here, the tree is already Initialized (has at least 2 leaf\*/  
 Node y = this.root;  
 while (y.numOfChild != 0) {  
 if (z.key.compareTo(y.lChild.key) < 0)  
 y = y.lChild;  
 else if (z.key.compareTo(y.mChild.key) < 0 || y.rChild == null)  
 y = y.mChild;  
 else  
 y = y.rChild;  
 }  
 Node x = y.parent;  
 Node w = insertAndSplit(x, z);  
 while (x != this.root) {  
 x = x.parent;  
 if (w != null)  
 w = insertAndSplit(x, w);  
 else  
 updateNode(x);  
 }  
 if (w != null) {  
 Node<K, V> s = new Node<K, V>(null, null);  
 setChildren(s, x, w, null);  
 this.root = s;  
 }  
 numOfLeaf++;  
 }  
  
 */\*\*  
 \** ***@param*** *x : Node  
 \* {****@code*** *updateNode} Method update the key of the maximum key,number of descended, number of  
 \* children and  
 \* sum of values in x subtree.  
 \* Note : (only) {****@code*** *nodeX.rChild} can be null.  
 \*/* private void updateNode(Node<K, V> x) {  
 x.numOfLeaf = x.lChild.numOfLeaf + x.mChild.numOfLeaf;  
 x.numOfChild = 2;  
 x.value = (V) x.lChild.value.createCopy();  
 x.value.addValue(x.mChild.value);  
 x.key = x.mChild.key;  
 if (x.rChild != null) {  
 x.numOfLeaf = +x.rChild.numOfLeaf;  
 x.value.addValue(x.rChild.value);  
 x.key = x.rChild.key;  
 x.numOfChild++;  
 }  
 }  
  
 */\*\*  
 \** ***@param*** *newLeaf : new Leaf to insert  
 \* {****@code:initializedInsert****} method is responsible for insert a new leaf  
 \* in the extreme case (number of leafs is less then two).  
 \*/* private void initializedInsert(Node<K, V> newLeaf) {  
 newLeaf.parent = this.root;  
 newLeaf.numOfLeaf = 1;  
 this.root.value = (V) newLeaf.value.createCopy();  
 this.root.key = newLeaf.key;  
 if (this.root.lChild == null) {  
 this.root.lChild = newLeaf;  
 this.numOfLeaf = 1;  
 } else {  
 this.root.value.addValue(this.root.lChild.value);  
 if (newLeaf.key.compareTo(this.root.lChild.key) > 0) {  
 this.root.mChild = newLeaf;  
 } else {  
 this.root.mChild = this.root.lChild;  
 this.root.lChild = newLeaf;  
 this.root.key = this.root.mChild.key;  
 }  
 this.numOfLeaf = 2;  
 }  
 this.root.numOfLeaf++;  
 this.root.numOfChild++;  
 }  
  
 */\*\*  
 \** ***@param*** *x : root Node in subtree.  
 \** ***@param*** *l : left Node in subtree.  
 \** ***@param*** *m : middle Node in subtree.  
 \** ***@param*** *r : right Node in subtree.  
 \* {****@code:setChildren****} method set l, m and r to be the left, middle and right children respectively, of x.  
 \* Note : r is the only note that can be null.  
 \*/* private void setChildren(Node<K, V> x, Node<K, V> l, Node<K, V> m, Node<K, V> r) {  
 x.lChild = l;  
 x.mChild = m;  
 x.rChild = r;  
 x.lChild.parent = x;  
 x.mChild.parent = x;  
 if (x.rChild != null)  
 x.rChild.parent = x;  
 updateNode(x);  
 }  
  
 */\*\*  
 \** ***@param*** *x : the root Node in subtree.  
 \** ***@param*** *z : Node to be added.  
 \* Insert node z as a child of node x, split x if necessary and return the new node (null  
 \* if the method didn't split).  
 \*/* private Node<K, V> insertAndSplit(Node<K, V> x, Node<K, V> z) {  
 Node<K, V> l = x.lChild;  
 Node<K, V> m = x.mChild;  
 Node<K, V> r = x.rChild;  
 /\*Dont need to split Node x\*/  
 if (x.numOfChild == 2) {  
 if (z.key.compareTo(l.key) < 0)  
 setChildren(x, z, l, m);  
 else if (z.key.compareTo(m.key) < 0)  
 setChildren(x, l, z, m);  
 else setChildren(x, l, m, z);  
 return null;  
 }  
 /\*If the method got to here, split is needed\*/  
 Node<K, V> y = new Node<K, V>(null, null);  
 if (z.key.compareTo(l.key) < 0) {  
 setChildren(x, z, l, null);  
 setChildren(y, m, r, null);  
 } else if (z.key.compareTo(m.key) < 0) {  
 setChildren(x, l, z, null);  
 setChildren(y, m, r, null);  
 } else if (z.key.compareTo(r.key) < 0) {  
 setChildren(x, l, m, null);  
 setChildren(y, z, r, null);  
 } else {  
 setChildren(x, l, m, null);  
 setChildren(y, r, z, null);  
 }  
 return y;  
 }  
  
 */\*\*  
 \** ***@param*** *key : the key to delete  
 \* {****@code:delete****} method will delete the Node with the desire key if  
 \* the tree has a Node with the same key, if not, nothing will happens.  
 \*/* public void delete(K key) {  
 Node<K, V> x = auxSearch(this.root, key);  
 /\*If the current key is not in the tree\*/  
 if (x == null)  
 return;  
 Node<K, V> y = x.parent;  
 if (x == y.lChild)  
 setChildren(y, y.mChild, y.rChild, null);  
 else if (x == y.mChild)  
 setChildren(y, y.lChild, y.rChild, null);  
 else  
 setChildren(y, y.lChild, y.mChild, null);  
 while (y != null) {  
 if (y.mChild == null) {  
 if (y != this.root)  
 y = borrowOrMerge(y);  
 else {  
 this.root = y.lChild;  
 y.lChild.parent = null;  
 }  
 } else  
 y = y.parent;  
 }  
 }  
  
 */\*\*  
 \** ***@param*** *y : Node  
 \* {****@code:borrow****} method borrow a Node child from a sibling x of y or merge x and y.  
 \* return a reference to the parent of y (and x).  
 \*/* private Node<K, V> borrowOrMerge(Node y) {  
 Node z = y.parent;  
 if (y == z.lChild) {  
 Node x = z.mChild;  
 if (x.rChild != null) {  
 setChildren(y, y.lChild, x.lChild, null);  
 setChildren(x, x.mChild, x.rChild, null);  
 } else {  
 setChildren(x, y.lChild, x.lChild, x.mChild);  
 setChildren(z, x, z.rChild, null);  
 }  
 return z;  
 }  
 if (y == z.mChild) {  
 Node x = z.lChild;  
 if (x.rChild != null) {  
 setChildren(y, x.rChild, y.lChild, null);  
 setChildren(x, x.lChild, x.mChild, null);  
 } else {  
 setChildren(x, x.lChild, x.mChild, null);  
 setChildren(z, x, z.rChild, null);  
 }  
 return z;  
 }  
 Node x = z.mChild;  
 if (x.rChild != null) {  
 setChildren(y, x.rChild, y.lChild, null);  
 setChildren(x, x.lChild, x.mChild, null);  
 } else {  
 setChildren(x, x.lChild, x.mChild, y.lChild);  
 setChildren(z, z.lChild, x, null);  
 }  
 return z;  
 }  
  
 public Value search(Key key) {  
 //key is larger than largest key in data structure  
 if (key.compareTo(this.root.key) > 0) {  
 return null;  
 }  
 Node<K, V> x = auxSearch(this.root, key);  
 if (x == null)  
 return null;  
 return x.value.createCopy();  
 }  
  
 private Node<K, V> auxSearch(Node currNode, Key key) {  
 if (currNode.lChild == null) {  
 if (currNode.key.compareTo(key) == 0)  
 return currNode;  
 else  
 return null;  
 }  
  
 //navigate to appropriate internal node  
 if (key.compareTo(currNode.lChild.key) <= 0)  
 return auxSearch(currNode.lChild, key);  
 else if (key.compareTo(currNode.mChild.key) <= 0)  
 return auxSearch(currNode.mChild, key);  
 else //because we checked at root if key is inside data table there will always be right child  
 return auxSearch(currNode.rChild, key);  
 }  
  
  
 public int Rank(Key key) {  
 int rank = 0;  
  
 //check if the key is larger than all the keys in the data structure  
 if (key.compareTo(this.root.key) > 0)  
 return 0;  
  
 //search data structure for key and linear ordering  
 //the initial rank is 0, no sentinels  
 return auxRank(this.root, key, 0);  
 }  
  
 private int auxRank(Node currNode, Key key, int currRank) {  
 //if at leaf  
 if (currNode.lChild == null) {  
 if (currNode.key.compareTo(key) == 0) {  
 if (currNode.equals(currNode.parent.lChild))  
 return currRank + 1;  
 else if (currNode.equals(currNode.parent.mChild))  
 return currRank + 2;  
 else  
 return currRank + 3;  
 }  
 //if the key is not in the data structure  
 else  
 return 0;  
 }  
 //navigate to appropriate node and count the number of leaves to the left of next node  
 if (key.compareTo(currNode.lChild.key) <= 0) {  
 return auxRank(currNode.lChild, key, currRank);  
 } else if (key.compareTo(currNode.mChild.key) <= 0) {  
 currRank =+ currNode.lChild.numOfLeaf;  
 return auxRank(currNode.mChild, key, currRank);  
 } else {//because we checked at root if key is inside data table there will always be right child  
 currRank =+ currNode.lChild.numOfLeaf + currNode.mChild.numOfLeaf;  
 return auxRank(currNode.rChild, key, currRank);  
 }  
 }  
  
 public Key select(int index) {  
 //check if the index is larger than the total amount of leaves or smaller then 1  
 if ((index > this.root.numOfLeaf) || index < 1)  
 return null;  
  
 return auxSelect(this.root, index);  
 }  
  
 private Key auxSelect(Node currNode, int index) {  
 //if arrived at leaf return key  
 if (currNode.lChild == null)  
 return currNode.key.createCopy();  
  
 //if at internal node, check to which child to continue  
 if (index <= currNode.lChild.numOfLeaf)  
 return auxSelect(currNode.lChild, index);  
 else {  
 index =- currNode.lChild.numOfLeaf; //subtract from index the num of descendents in the left child sub-tree  
 if (index <= currNode.mChild.numOfLeaf)  
 return auxSelect(currNode.mChild, index);  
 else {  
 index =- currNode.mChild.numOfLeaf; //subtract from index the num of descendents in the mid child sub-tree  
 return auxSelect(currNode.rChild, index);  
 }  
 }  
 }  
  
 public Value sumValuesInInterval(Key key1, Key key2) {  
 //check if key1 is smaller/equal to key2  
 if (key1.compareTo(key2) > 0)  
 return null;  
 }  
  
 private Key findNextHighestKey(int number) {  
 return null;  
 }  
}