COP 5536 Spring 2016 Project Report

The red-black tree based event counter has been implemented in Java. The project has been tested using Oracle JDK 1.8 (javac 1.8.0_73). The structure of the project and the method prototypes are described below.

The project is divided into 3 Java files for purposes of modularity.

1. **bbst.java** – This class holds the `main()` method. This reads the input event details from the text file passed in the command line arguments and builds the Red-Black tree. Initially, the program reads the event IDs and the counts to create `RedBlackTreeNode` objects which are added to an `ArrayList` instance. The class then reads the commands from the redirected standard input, calls the specific methods in the `RedBlackTreeEventCounter` class and writes the results to standard output.

```
public class bbst {
    public static void main(String[] args) {
         …
         fileReader = new BufferedReader(new InputStreamReader(new FileInputStream(args[0])));
         while((scannedLine = fileReader.readLine()) != null) {
                  //Read event ID and count here
         }
         //Build tree here
         while (stringScanner.hasNext()) {
                  String command = stringScanner.next();
                  //Read command here and call appropriate method in RedBlackTreeEventCounter
         }
         …
    }
}
```

2. **RedBlackTreeEventCounter.java** – This class implements and exposes the operations of the event counter at a high level. The low level Red-Black tree operations are delegated to the `RedBlackTree` class. The class structure is shown below.

```java
public class RedBlackTreeEventCounter {
	RedBlackTree redBlackTree = new RedBlackTree();

	public void buildEventCounter(ArrayList<RedBlackTree.RedBlackTreeNode>
eventArrayList) {
		…
	}

	public int increase(int ID, int amount) {
		…
	}

	public int reduce(int ID, int amount) {
		…
	}

	public int count(int ID) {
		…
	}

	public int inRange(int ID1, int ID2) {
		…
	}

	public RedBlackTree.RedBlackTreeNode next(int ID) {
		…
	}

	public RedBlackTree.RedBlackTreeNode previous(int ID) {
		…
	}
}
```

The below methods are exposed to bbst class.

i. **`buildEventCounter:`** Builds the event counter from the list of RedBlackTreeNode objects. This runs in O(n) time. This method calls the `buildTreeFromSortedList` method in RedBlackTree class.

```java
public void buildEventCounter(ArrayList<RedBlackTree.RedBlackTreeNode> eventArrayList) {
	//Call buildTreeFromSortedList in RedBlackTree
}
```

ii. **`increase:`** Increases the count of the event ID by amount. If ID is not present, inserts it. Print the final count. This runs in O(lg n) time. This method calls the `treeSearch` and `redBlackInsert` methods from `RedBlackTree` class.

```java
public int increase(int ID, int amount) {
	//Search the tree for the ID.
	//Increase the count if found else insert a new node.
}
```

iii. **`reduce:`** Decreases the count of the event ID by amount. If the ID's count becomes less than or equal to 0, removes the ID from the counter. Prints the count of the ID after deletion or 0 if the ID is removed or is not present. This runs in O(lg n) time. This method calls the `treeSearch` and `redBlackDelete` methods from the `RedBlackTree` class.

```java
public int reduce(int ID, int amount) {
        //Search tree for the ID
        //Subtract count if found and delete node if count <= 0.
}
```

iv. `count:` Searches the counter for the event ID and returns the count. Returns 0, if not present. This method calls the `treeSearch` method from `RedBlackTree` class.

```java
public int count(int ID) {
        //Search tree for the ID
}
```

v. `inRange:` Returns the total count for IDs between ID1 and ID2. This method calls the `rangeSearch` method from the `RedBlackTree` class.

```java
public int inRange(int ID1, int ID2) {
        //Find all the nodes with IDs between ID1 and ID2
        //Iterate through the returned nodes and get sum of their counts
}
```

vi. `next:` Returns the event with the lowest ID that is greater than ID. This method calls the `treeSuccessor` method from `RedBlackTree` class.

```java
public RedBlackTree.RedBlackTreeNode next(int ID) {
        //Find the successor/ceiling of ID
}
```

vii. `previous:` Returns the event with the greatest ID that is less than ID. This method calls the `treePredecessor` method from `RedBlackTree` class.

```java
public RedBlackTree.RedBlackTreeNode previous(int ID) {
        // Find the predecessor/floor of the ID
}
```

1. **RedBlackTree.java**: This class implements the low level operations of the Red-Black tree and exposes these methods for use by the `RedBlackTreeEventCounter` class. As shown below, this class comprises a static inner class called `RedBlackTreeNode`, an enum called `NodeColor` and methods for red-black operations.

```
public class RedBlackTree {
        static class RedBlackTreeNode {      … }

        private enum NodeColor { … }

        public void redBlackInsert(int ID, int count) { … }

        private void redBlackInsertFixUp(RedBlackTreeNode redBlackTreeNode) { … }

        private void leftRotate(RedBlackTreeNode redBlackTreeNode) { … }

        private  void rightRotate(RedBlackTreeNode redBlackTreeNode) { … }

        public void redBlackDelete(RedBlackTreeNode nodeToDelete) { … }

        private void redBlackTransplant(RedBlackTreeNode nodeToReplace, RedBlackTreeNode
nodeToReplaceWith) { … }

        public RedBlackTreeNode treeMinimum(RedBlackTreeNode redBlackTreeNode) { … }

        public RedBlackTreeNode treeMaximum(RedBlackTreeNode redBlackTreeNode) { … }

        private void redBlackDeleteFixUp(RedBlackTreeNode redBlackTreeNode) { … }

        public RedBlackTreeNode treeSearch(int theID) { … }

        public RedBlackTreeNode treeSuccessor(RedBlackTreeNode redBlackTreeNode, int ID) { …
}

        public RedBlackTreeNode treePredecessor(RedBlackTreeNode redBlackTreeNode, int ID) {
… }

        public void buildTreeFromSortedList(int size, Iterator<RedBlackTreeNode> iterator) {
… }

        private RedBlackTreeNode buildTree(int currentLevel, int begin, int end, int
redLevel,
                                   Iterator<RedBlackTreeNode> iterator) { … }

        public ArrayList<RedBlackTreeNode> rangeSearch(RedBlackTreeNode rootNode, int ID1,
int ID2,
                                             ArrayList<RedBlackTreeNode> nodesInRange) { … }

        private static NodeColor colorOf(RedBlackTreeNode redBlackTreeNode) { … }

        private static RedBlackTreeNode parentOf(RedBlackTreeNode redBlackTreeNode) { … }

        private static void setColor(RedBlackTreeNode redBlackTreeNode, NodeColor color) { …
}

        private static RedBlackTreeNode leftChildOf(RedBlackTreeNode redBlackTreeNode) { … }

        private static RedBlackTreeNode rightChildOf(RedBlackTreeNode redBlackTreeNode) { … }
}
```

i. **`RedBlackTreeNode:`** This inner class is used to represent the nodes in the Red-Black tree. It contains fields for ID, count, left child, right child, parent and color. The constructor colors every new node object RED.

ii. **`NodeColor:`** This enum holds two values – RED and BLACK to represent the color notations of the tree nodes.

iii. **redBlackInsert**: Builds a new node from ID and count and inserts into the tree. This method inserts a node into the tree as if it were an ordinary binary search tree. As the node inserted is RED by default, to rebalance the tree after insertion the method redBlackInsertFixUp is called. If there is no node in the tree, the new node is made the root and colored BLACK.

```
public void redBlackInsert(int ID, int count) {
        //If no node, insert new node as root and color BLACK
        //Else insert node as if unbalanced BST and call Fix Up method
}
```

iv. **redBlackInsertFixUp:** Restores the red black properties which might have been violated after insertion of a new node. This method uses a sequence of color flips and rotations to rebalance the tree. It handles the cases where the inserted node's uncle is RED, inserted node's uncle is BLACK and the node is a right child, the inserted node's uncle is BLACK and the node is a left child.

```
private void redBlackInsertFixUp(RedBlackTreeNode redBlackTreeNode) {
        //Use color flips and left and right rotations as necessary based on the case
}
```

v. **leftRotate:** Performs left rotation on the subtree to restore balance. The left rotation pivots around the link from the node to its right child.

```
private void leftRotate(RedBlackTreeNode redBlackTreeNode) {
        //Pivot around the link from redBlackTreeNode to its right child
}
```

vi. **rightRotate:** Performs right rotation on the subtree to restore balance. The right rotation pivots around the link from the node to its left child.

```
private  void rightRotate(RedBlackTreeNode redBlackTreeNode) {
        //Pivot around the link from redBlackTreeNode to its right child
}
```

vii. **redBlackDelete:** Deletes a node from the tree. This method keeps track of a node called replacementSuccessor which might cause violations of the Red-Black properties. After deletion the method redBlackDeleteFixUp is called, if the original node color was BLACK.

```
public void redBlackDelete(RedBlackTreeNode nodeToDelete) {
        RedBlackTreeNode replacementNode = nodeToDelete;
        RedBlackTreeNode replacementSuccessor;
        NodeColor replacementNodeOriginalColor = replacementNode.nodeColor;
        …
        //Delete the node keeping its children in consideration
        …
        if(replacementNodeOriginalColor == NodeColor.BLACK) {
                redBlackDeleteFixUp(replacementSuccessor);
        }
}
```

viii. **redBlackTransplant:** Replaces one subtree as a child of its parent with another subtree. Used by the redBlackDelete method.

```
private void redBlackTransplant(RedBlackTreeNode nodeToReplace, RedBlackTreeNode
nodeToReplaceWith) {
        //Replace nodeToReplace with nodeToReplaceWith
}
```

ix. **`treeMinimum:`** Finds element in a tree/subtree whose key is a minimum. Used by the `redBlackDelete` and `treeSuccessor` methods.

```
public RedBlackTreeNode treeMinimum(RedBlackTreeNode redBlackTreeNode) {
        //Recursively find the left child of redBlackTreeNode
}
```

x. **`treeMaximum:`** Finds element in a tree/subtree whose key is a maximum. Used by `treePredecessor` method.

```
public RedBlackTreeNode treeMaximum(RedBlackTreeNode redBlackTreeNode) {
        //Recursively traverse the right subtree of redBlackTreeNode
}
```

xi. **`redBlackDeleteFixUp:`** Restores the red black properties that might have been violated after deletion of a node. This method uses a sequence of color flips and rotations to rebalance the tree. It handles the cases where the successor node's sibling is `RED`, successor's sibling is `BLACK` and both of the sibling's children are `BLACK`, the successor's sibling is `BLACK` whose left child is `RED` and right child is `BLACK`, successor's sibling is `BLACK` whose right child is `RED`.

```
private void redBlackDeleteFixUp(RedBlackTreeNode redBlackTreeNode) {
        //Use color flips and left and right rotations as necessary based on the case
}
```

xii. **`treeSearch:`** Searches for a node with the input ID in the tree.

```
public RedBlackTreeNode treeSearch(int theID) {
        //Traverse the left or right subtrees based on whether the input ID is lesser or
greater than root's ID
}
```

xiii. **`treeSuccessor:`** Finds the successor of the input ID in the sorted order determined by an inorder traversal, if the ID exists in the tree. Otherwise, returns the node with the lowest ID which is greater than the input ID.

```
public RedBlackTreeNode treeSuccessor(RedBlackTreeNode redBlackTreeNode, int ID) {
        //If right subtree is empty, go up the tree until the current node is a left child
and return parent
        //Otherwise return the minimum in the right subtree of redBlackTreeNode
        //If ID doesn't exist in the tree, recursively search the left and right subtrees for
the successor based on whether ID is lesser or greater than redBlackTreeNode's ID
}
```

xiv. **`treePredecessor:`** Finds the predecessor of the input ID in the sorted order determined by an inorder traversal, if the ID exists in the tree. Otherwise, returns the node with the greatest ID which is less than the input ID.

```
public RedBlackTreeNode treePredecessor(RedBlackTreeNode redBlackTreeNode, int ID) {
         //If left subtree is empty, go up the tree until the current node is a right child
and return parent
         //Otherwise return the maximum in the left subtree of redBlackTreeNode
         //If ID doesn't exist in the tree, recursively search the left and right subtrees for
the predecessor based on whether ID is lesser or greater than redBlackTreeNode's ID
}
```

xv. `buildTreeFromSortedList:` Builds tree from sorted ArrayList of RedBlackTreeNode objects. This method finds the depth upto which all nodes can be assigned `BLACK`. The rest of the nodes are colored `RED`. Then calls the `buildTree` method with the `currentLevel` value 0, the `redLevel` calculated, the iterator of the `ArrayList` and the start and end indexes of the `ArrayList`.

```
private RedBlackTreeNode buildTree(int currentLevel, int begin, int end, int redLevel,
                                   Iterator<RedBlackTreeNode> iterator) {
         //The middle element is made the root. The left subtree is recursively is constructed
from elements from the ArrayList with 0 to mid – 1. The right subtree is recursively is built
from the elements from the ArrayList with mid + 1 to size(ArrayList).
         //If the currentLevel matched redLevel, color the node RED
}
```

xvi. `rangeSearch:` Stores all nodes in the tree whose IDs are in the range between ID1 and ID2.

```
public ArrayList<RedBlackTreeNode> rangeSearch(RedBlackTreeNode rootNode, int ID1, int ID2,
                                               ArrayList<RedBlackTreeNode> nodesInRange) {
         //If rootNode's ID lies between ID1 and ID2, store it
         //If it is greater than ID1 recursively traverse left
         //If it is less than ID2 recursively traverse right
}
```

<u>Running Times:</u> I tested this program on my personal laptop with 8GB RAM and an i7 Ultrabook CPU. The running times for the program under different input sizes is given below. The JVM heap size was default for the first 3 input conditions and was increased to 6GB for the last condition with 100 million input size.

1. 100 input events: 0.279s
2. 10^6 input events: 1.265s
3. 10^7 input events: 7.529s
4. 10^8 input events: 2m 4.706s