CSCI 2270 – Data Structures and Algorithms

Instructor: Godley

Assignment 5

Due: Sunday, July 15th at 11:59pm

Red-Black Trees

In this assignment, you need to answer the following two questions. Your answers must be typed to receive credit. Submit a pdf of your written answers to Moodle to the Assignment 5 link.

Question 1: Does inserting a node into a red-black tree, re-balancing, and then deleting it result in the original tree?

Question 2: Does deleting a node with no children from a red-black tree, re-balancing, and then reinserting it with the same key always result in the original tree?

Your answers to these questions need to include a specific example of a tree showing the original tree, the results of the re-balancing, and the final tree after deleting. Include a graphic of your tree generated in a graphics program, or the Red-black tree visualization website: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html.

Your answer also needs to include an explanation of how the algorithm proceeds to insert and delete nodes in the tree. Include information such as which node is the parent, grandparent, and uncle at each step, and which node is the argument for the left and right rotate steps. Refer to your specific trees in your explanation.

Red-black algorithms for insert and delete

For your reference, the insert and delete algorithms are provided here.

Insert algorithm

```
redBlackInsert(value)
	{ x = insert(value) //add a node to the tree as a red node
	while(x != root and x.parent.color == red){
		If(parent == x.parent.parent.left){
			uncle = x.parent.parent.right
			if(uncle.color == red){
				x.parent.color = black
				uncle.color = black
				x.parent.parent.color = red
				x = x.parent.parent
			}else{
				if(x == x.parent.right){
					x = x.parent
					leftRotate(x)
				}
				x.parent.color = black
				x.parent.parent.color = red
				rightRotate(x.parent.parent)
			}
		}else{
			//x.parent is a right child. Swap left and right for algorithm }
	}
	root.color = black
}
```

Delete algorithm

```
redBlackDelete(value){
	node = search(value)
	nodeColor = node.color
	if(node != root){
		if (node.leftChild == nullNode and node.rightChild == nullNode){
		//no children
			node.parent.leftChild = nullNode x = node.leftChild
		} else if (node.leftChild != nullNode and node.rightChild != 
	nullNode){ //two children
			min = treeMinimum(node.rightChild)
			nodeColor = min.color //color of replacement
			x = min.rightChild
			if (min == node.rightChild){
				node.parent.leftChild = min
				min.parent = node.parent
				min.leftChild = node.leftChild
				min.leftChild.parent = min
			} else {
				min.parent.leftChild = min.rightChild
				min.rightChild.parent = min.parent
				min.parent = node.parent
				node.parent.leftChild = min
				min.leftChild = node.leftChild
				min.rightChild = node.rightChild
				node.rightChild.parent = min
				node.leftChild.parent = min
			}
			min.color = node.color //replacement gets nodes color
		}else{ //one child
```

```
                x = node.leftChild
                node.parent.leftChild = x
                x.parent = node.parent
        }else{
                //repeat cases of 0, 1, or 2 children
                //replacement node is the new root
                //parent of replacement is nullNode
        }
        if (nodeColor == BLACK){
                RBBalance(x)
        }
        delete node
}
```

Red-black rebalancing after delete

```
RBBalance(x){
        while (x != root and x.color == BLACK){
                if (x == x.parent.leftChild){
                        s = x.parent.rightChild
                        if (s.color == RED){ //Case 1
                                s.color = BLACK
                                x.parent.color = RED
                                leftRotate(x.parent)
                                s = x.parent.rightChild
                        }
                        if (s.leftChild.color == BLACK and s.rightChild.color ==
                        BLACK){ //Case 2
                                s.color = RED
                                x = x.parent
```

```
                } else if (s.leftChild.color == RED and s.rightChild.color
                == BLACK){ //Case 3
                      s.leftChild.color = BLACK
                      s.color = RED
                      rightRotate(s)
                      s = x.parent.rightChild
                }else{
                      s.color = x.parent.color //Case 4
                      x.parent.color = BLACK
                      s.rightChild.color = BLACK
                      leftRotate(x.parent)
                      x = root
                }
          }else{
                //x is a right child //exchange left and right
          }
    }
    x.color = BLACK
}
```