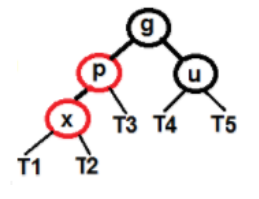
Trevor Stanley

3/18/18

Assignment 7: R-B Trees

1. See example 1. In the first figure, we have our original tree with the newly inserted x. The parent is the node P, the uncle is the node u, and the sibling is T3. We then proceed to the while condition in the algorithm, where the uncle pointer is set to node U. Looking at figure 2, we have inserted the X node and completed a single rotate right (see comments in pseudo code) on the X, P, and G nodes to create the current structure in the second figure. We then change the G node to red and the P node to black. Removing this X node would necessitate the restructuring of the tree. However, this will not always be the case for every tree.

**Example 1**

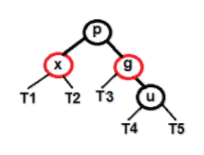


Figure 2

**Insert Algorithm:**

Figure 1

Node\* x = insert(X); //a 4 node is inserted into BST; x points to it

While(x != root && x.parent.color == red) //this condition is true

If(x.parent==x.parent.parent.leftC) //this condition is true

Source:

https://www.geeksforgeeks.org/red-black-tree-set-2-insert/

Node\*uncle=x.parent.parent.rightC //points to a nil node

If(uncle.color==”red”) //This condition fails

x.parent.color = “black”

uncle.color = “black”

x.parent.parent.color = “red”

x = x.parent.parent

else

if(x==x.parent.rightC) //This if condition fails

x=x.parent

leftRotate(x)

x.parent.color = “black” //proceeds to these

x.parent.parent.color = “red”

rightRotate(x.parent.parent)

else

swap left and right in algorithm //x.parent is a rightC

root.color = “black” //makes sure the root is always colored black

1. See example 2. No, deleting a node with no children, re-balancing, and then reinserting does not always result in the original tree (see figure 1-3 in example 2). First, the 14 is deleted via the delete algorithm; it goes into the case where there are no children and the parent’s leftChild (in this case 11) is pointed at a null node. Since nodeColor is black in this case, RBBalance is called for x (which is black and the nil left child of 14). The while and first if condition are true. The S node is then pointed at the black nil right child of the null node where 14 was. Case 4 is then entered into, as the color of s is the same as its parent (black). A single rotate right then occurs on the 7, 8, and 9, resulting in figure 2. Finally, 14 is inserted back into the tree via the insert algorithm, and results in a different tree. 7 is the uncle of 9 and 14 in this case. To summarize, the original tree is not maintained when the node that is deleted is black (14 in this case) and then reinserted (figures 1-3). However, when the node to be deleted is red (such as in figure 4), it is deleted, and then reinserting the same node maintains the original tree.

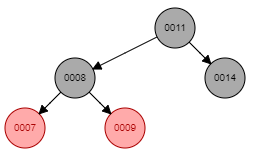
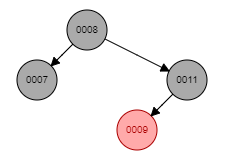
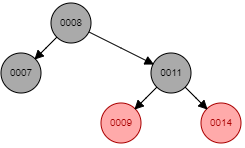
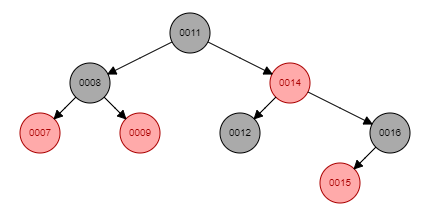
**Example 2**

Figure 1

Figure 2

Figure 3

Figure 4



(see below pseudocode steps)

**Delete Algorithm**

redBlackDelete(value){

node = search(value)

nodeColor = node.color

if(node != root){

if(node.righttChild == nullNode and node.leftChild == nullNode){ //no children

node.parent.rightChild = nullNode //node of value 14 becomes null

x = node.rightChild

}else if(node.rightChild != nullNode and node.lefttChild != nullNode){ //two children

min = treeMinimum(node.leftChild)

nodeColor = min.color //color of replacement

x = min.lefttChild

if (min == node.leftChild){

node.parent.rightChild = min

min.parent = node.parent

min.rightChild = node.rightChild

min.rightChild.parent = min

}else{

min.parent.rightChild = min.leftChild

min.lefttChild.parent = min.parent

min.parent = node.parent

node.parent.rightChild = min

min.rightChild = node.rightChild

min.leftChild = node.lefttChild

node.lefttChild.parent = min

node.rightChild.parent = min

}

min.color = node.color //replacement gets nodes color

}else{ //one child

x = node.righttChild

node.parent.righttChild = 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){ //color is black, so need to call RBBalance (see below)

RBBalance(x)

}

delete node

}

**Red-black rebalancing after delete**

RBBalance(x){

while (x != root and x.color == BLACK){ //true; continue

if (x == x.parent.rightChild){ //true; continue

s = x.parent.lefttChild //S, or sibling, is now the node of value 8

if (s.color == RED){ //Case 1

s.color = BLACK

x.parent.color = RED

rightRotate(x.parent)

s = x.parent.lefttChild

}

if (s.rightChild.color == BLACK and s.leftChild.color == BLACK){ //Case 2

s.color = RED

x = x.parent

}else if(s.rightChild.color==RED and s.leftChild.color == BLACK){ //Case 3

s.rightChild.color = BLACK

s.color = RED

leftRotate(s)

s = x.parent.leftChild

}else{ //Case 4; this is true

s.color = x.parent.color //color is set to black (double black?)

x.parent.color = BLACK // same as above, ensures black parent

s.lefttChild.color = BLACK //colors the node of val 7 to black

rightRotate(x.parent) //repositions the 7, 8 and 11

x = root //x now points to the root

}

}else{

//x is a left child

//exchange left and right

}

}

x.color = BLACK

}