CSCI 2270 - Data Structures and Algorithms
Instructor: Hoenigman
Recitation 10[1]

In this recitation, you will be learning how to insert into Red Black Trees.

Red Black Trees

One common approach to tree balancing is to build the BST as a red-black tree. The red black tree algorithm is used to balance trees. In the red-black tree algorithm, each node in the BST is assigned a color, either red or black, and the nodes in the tree are ordered such that no path from the root to a leaf can be more than twice as long as any other path. This coloring results in red-black trees having a height of O(log n), which **guarantees a worst-case runtime of O(log n) on search, insert, and delete operations**.

Red-black node properties

Each node in a red-black tree has at least the following properties:
• **color**
• **key**
• **left child**
• **right child**
• **parent**

The only red-black property not found in a regular BST is the color, which is added to the nodes to create the structure in the tree. The properties that the tree must exhibit in order to be a valid red-black tree are:

- Property 1: A node is either red or black.
- Property 2: The root node is black.
- Property 3: Every leaf (NULL) node is black.
- Property 4: If a node is red, then both of its children must be black.
- Property 5: For each node in the tree, all paths from that node to the leaf nodes contain the same number of black nodes.

Another difference between a regular BST and a red-black tree is how the leaf nodes are represented. In a red-black tree, the leaf nodes are external sentinel nodes (NIL) with all of the same properties as a regular node, but they are effectively empty nodes. The leaf nodes are black to satisfy Property 3 of a red-black tree. Sentinel nodes are just used as an alternative to NULL.

[1] Excerpt From: Rhonda Hoenigman. “Visualizing Data Structures.” iBooks.

The red-black tree ADT contains similar functionality to the BST ADT to insert and delete nodes in the tree. Searching a red-black tree uses the same algorithm as searching a regular BST. The red-black tree ADT, contains public methods for the insert, delete, and search operations, as well as private methods to support these operations.

```
RedBlackTree:
private:
    root
    leftRotate(node)
    rightRotate(node)
    insertRB(value)
    rbBalance(node)
public:
    Init()
    redBlackInsert(value)
    redBlackDelete(value)
    search(value)
    deleteTree()
```

Inserting a node into a Red Black Tree

Nodes are added to red-black trees in the same way they are added to a regular BST. However, when a node is added, the operation can destroy the red-black tree properties, which requires that there are additional steps in the algorithm to restore these properties.

There are **three changes** to the BST insert operation needed to support a red-black tree.
In a red-black tree:

1. Replace all instances of NULL in the BST insert() algorithm with the sentinel node - nullNode. This change sets the parent of the root to nullNode and the left and right children of a new node to nullNode.
2. Set the color of the new node to red when you are insearting.
3. Resolve any violation of the red-black properties using **tree balancing**.

If x is a node added to a red-black tree, then the initial conditions on x are:
• x->color = red
• x->leftChild = nullNode
• x->rightChild = nullNode

When a node is added to the tree, the two properties that can be violated are.
1. The root must be black.
2. The children of a red node must be black.

Both violations are possible because a new node is initially colored red. There are six possible configurations that a red-black tree can take on when a new node is inserted. Three configurations are symmetric to the other three depending on whether the parent of the new node is the left or right child of its parent. Figure 1 shows an example where the parent is the left child. In this figure, the new node is labeled x, and its parent is the 15. The steps needed to rebalance the tree depend on the color of the new node's "uncle" node. Figure 1 shows an example of an "uncle" node.

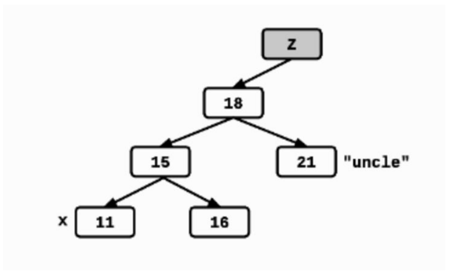


Figure 1. The "uncle" of x is x->parent->parent->rightChild. The color of x's "uncle" node determines the steps needed to rebalance the tree after inserting a node.

Case 1: The "uncle" node is red.
If the parent->parent->rightChild of the new node is red (shown as "uncle" in Figure 2), then parent of the new node is also red, and the parent.parent of the new node is black. In Figure 2, the new node is labeled x. It is initially colored red, and its parent and "uncle" are also red.

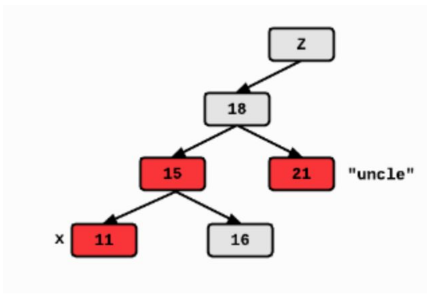


Figure 2. Case 1 example, where the "uncle" node is red. The x points to the new node. The parent of x is also red, which violates the constraint that a red node can't have a red child.

Steps to resolve a Case 1 violation in the tree:
1. Recolor both the parent and the "uncle" of the new node to be black, and recolor the parent->parent of the new node to be red. This recoloring resolves the violation up to the parent->parent level in the tree.
2. **Move up two levels** in the tree by setting x = x->parent->parent. Figure 3 shows the red-black tree after the violations have been resolved. The x in Figure 3 points to the node that would be recolored next, if additional iterations of recoloring were necessary.
3. Repeat Steps 1 and 2 **until x is the root of the tree or x's parent is black**.

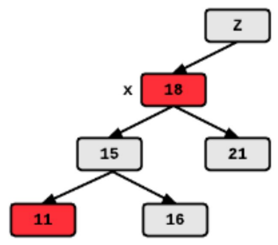


Figure 3. The red-black tree after recoloring to fix the violation of a red node having a red child.

Case 2: The new node is a right child and its uncle is black.
Case 3: The new node is a left child and its uncle is black.

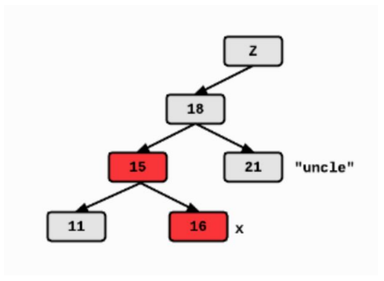


Figure 4. Case 2 where the "uncle" node is black and the new node x is a right child. The violation is that a red node has a red child.

In both Case 2 and Case 3, the “uncle” node is black. The difference in the cases is whether the new node is a left or right child of its parent.

Steps to resolve a Case 2 violation in the tree:
1. Set x = x->parent.
2. **Apply leftRotate(x) to convert a Case 2 configuration to a Case 3 configuration**. Additional rebalancing can then be applied to resolve the Case 3 violation. The result of the left rotation on the tree in Figure 4 is shown in Figure 5. The new node is now a left child of its parent.

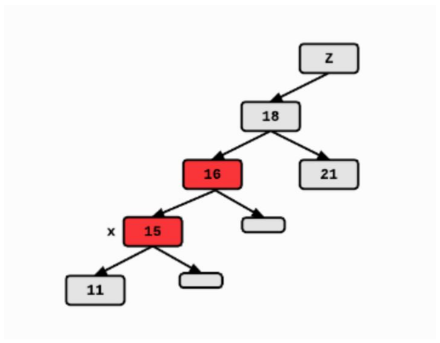


Figure 5. A Case 3 configuration is generated from a left rotation on x's parent on the tree in Figure 4.

Steps to resolve a Case 3 violation in the tree:
1. Recolor x->parent and x->parent->parent.

2. Apply a right rotation to x->parent->parent on the tree in Figure 5 to get the tree in Figure 6.

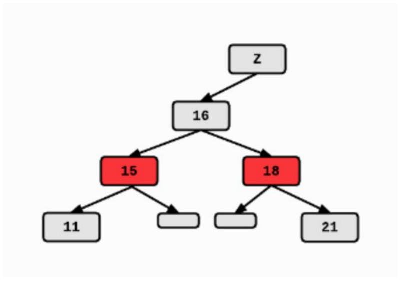


Figure 6. Red-black sub-tree after a right rotation on the tree shown in Figure 6. The tree is now balanced; all red-black violations have been resolved.

The algorithm to insert a node into a red-black tree is shown below. The redBlackInsert() algorithm takes the value of the node to insert as an argument, and calls insertRB(), to add the node to the tree. The functioninsertRB() exists to create and return a pointer to the new node.

<u>redBlackInsert(value)</u>

Insert a node into a red-black tree and apply the appropriate tree-balancing algorithm to restore the red-black properties.

<u>Algorithm redBlackInsert(value)</u>

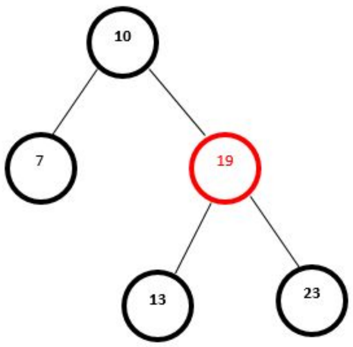
```
1.    x = insertRB(value)
2.    while ((x != root) and (x->parent->color == red))
3.       if (x->parent == x->parent->parent->left)       // left child
4.            uncle = x->parent->parent->right
5.            if (uncle->color == red)
6.                  //RBCase1Left(x, uncle)
7.                  x->parent->color = black
8.                  uncle->color = black
9.                  x->parent->parent->color = red
10.                 x = x->parent->parent
11.          else     //uncle color is black
12.                 if (x == x->parent->right) //Case2Left
13.                           x = x->parent
14.                           leftRotate(x)
15.                  //Case 3 - x is now left child
16.                 //RBCase3Left(x)
17.                 x->parent->color = black
18.                 x->parent->parent->color = red
19.                 rightRotate(x->parent->parent)
17.      else       //x's parent is a right child so x->parent == x->parent->parent->right. Right child
```

```
18.          //Exchange right and left
4.       uncle = x->parent->parent->left
5.       if (uncle->color == red)
6.           //RBCase1Left(x, uncle)
7.           x->parent->color = black
8.           uncle->color = black
9.           x->parent->parent->color = red
10.           x = x->parent->parent
11.      else    //uncle color is black
12.          if (x == x->parent->left)
13.              x = x->parent
14.               rightRotate(x)
15.          //Case 3 - x is now left child
16.          // RBCase3Left(x)
17.          x->parent->color = black
18.          x->parent->parent->color = red
19.          leftRotate(x->parent->parent)
20.  root->color = black
```

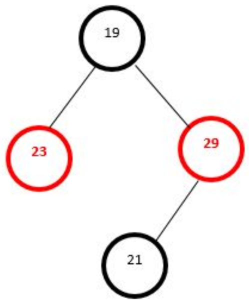
In class work

1. Which of the trees shown is **not** a valid RB tree?

A)

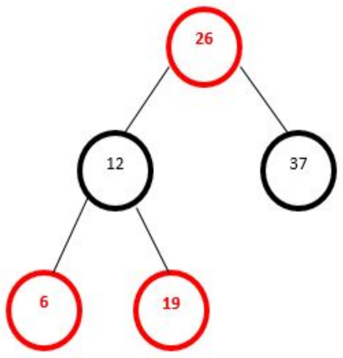


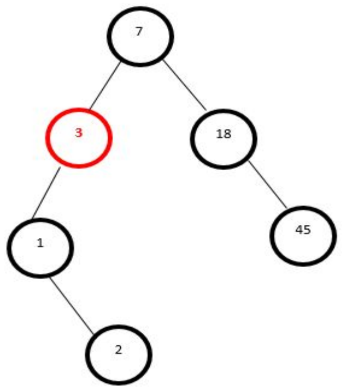
B)



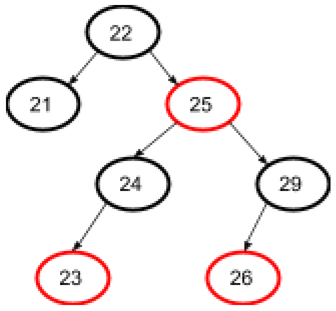
C)

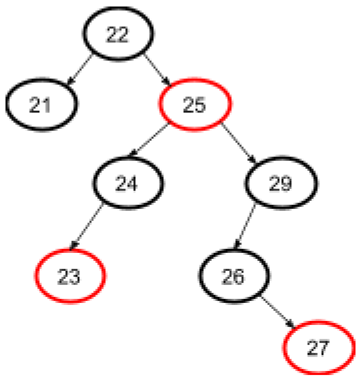
D)



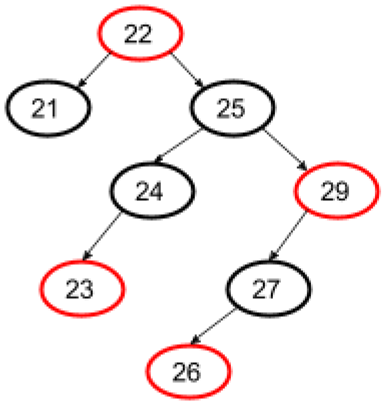


2. Show the tree after 27 is inserted and rebalancing is applied?

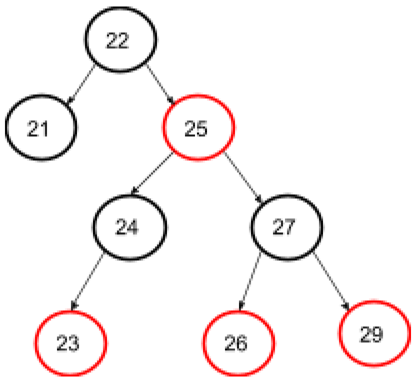




a.



b.

c. 

Recitation 10 Programming Assignment

There is a link on Moodle to a Recitation 10 exercise. You will have until Sunday at 5pm to complete.