**Programming Fundamentals III**

**COSC 2436**

**Lab 4: Binary Search Trees**

**Lab 19.1 Binary Tree Basics**

A **binary tree**, T, is either empty or such that:

1. T has a special node called the **root** node
2. T has two sets of nodes, LT and RT, called the **left subtree** and **right subtree** of T, respectively
3. LT and RT are binary trees

Suppose that T is a binary tree with the root node A. Let LA denote the left subtree of A and RA denote the right subtree of A. Now LA and RA are binary trees. Suppose that B is the root node of LA and C is the root node of RA. B is called the **left child** of A; C is called the **right child** of A. Moreover, A is called the **parent** of B and C. A binary tree can be shown pictorially. In the diagram of a binary tree, each node of the binary tree is represented as a circle, and the circle is labeled by the node. The root node of the binary tree is drawn at the top. The left child of the root node (if any) is drawn below and to the left of the root node. Similarly, the right child of the root node (if any) is drawn below and to the right of the root node. Children are connected to the parent by an arrow from the parent to the child. An arrow is usually called a **directed edge** or a **directed branch** (or simply a **branch**). Because the root node, B, of LA is already drawn, we apply the same (recursive) procedure to draw the remaining parts of LA. RA is drawn similarly. If a node has no left child, for example, we draw an arrow from the node to the left, ending with three stacked lines. That is, three lines at the end of an arrow indicate that the subtree is empty.

A node in a binary tree is called a **leaf** if it has no left and right children.

The **length** of a path in a binary tree is the number of branches on that path.

The **level** of a node in a binary tree is the number of branches on the path from the root to the node.

The **height** of a binary tree is the number of nodes on the longest path from the root to a leaf.

# Objectives

In this lab, you review the terminology and theory underlying binary trees.

# After completing this lab, you will be able to:

* Identify the terms used to describe binary trees, and understand how they are useful when the elements used in the trees can be sorted.

# Estimated completion time: 15–20 minutes

|  |  |  |
| --- | --- | --- |
|  | **Circle the correct answer** | |
| 1. The only node in a binary tree without a parent is the root. | **True** | **False** |
| 3. All nodes in a binary tree have a left, middle, and right child. | **True** | **False** |
| 4. Postorder traversal visits the right subtree of the given node first. | **True** | **False** |
| 5. Nodes without children are called leaf nodes. | **True** | **False** |
| 6. The height of a tree is the longest path from the root to a leaf node. | **True** | **False** |
| 7. A binary search tree is a tree that requires only two steps to find any node. | **True** | **False** |
| 8. Inorder traversal never visits the current node. | **True** | **False** |
| 9. The minimum height a binary tree with eight nodes can have is three. | **True** | **False** |
| 10. A binary search over a sorted array and a binary search tree have the same runtime properties. | **True** | **False** |

Provide short answers for the following questions.

1. Although binary search trees can reduce the expected runtime of insertion, deletion, and search, they are not guaranteed to do so. Give a sequence of numbers that, when inserted into a binary search tree, cause future operations to take *O*(*n*) time.

1,5,9,12,15,19,99

1. Generalize the sequence from Question 11 such that the first number is *n*. This should show that any sequence of numbers of this type should cause this problem.
2. *Critical Thinking Exercise:* Suppose you wanted to convert a binary tree into a binary search tree in place (i.e., not allocate a new tree). Devise an algorithm to perform this action in place.

# Lab 19.2 Using the BST Class

1. **Load the BinarySearchTree Class header files from Module 5** (Canvas EOL) and upload them into a new project in Visual Studio.
2. Create a source file with code to test various operation of binary search tree:
   1. Create two BSTs of integers named tree1 and tree2
   2. Insert in tree1 the numbers in the following order: 30, 35, 13, 22, 42, 28, 23, 16, 25
   3. Display the values in tree1 using preorder traversal.
   4. Modify function preorderTraversal so if the left child of a node is null, it displays LN and if the right child is null is displays RN.
   5. Insert in tree2 the numbers in the following order: 13, 16, 22, 23, 25, 28, 30, 35, 42
   6. Display the values in **tree2** using preorder traversal.
   7. Remove node 30 from tree1
   8. Display the values in tree1 using preorder traversal.

# Lab 19.3 Adding Rotations to Binary Search Trees

Binary Search Trees (BSTs) are an improvement over regular binary trees in that the elements in BSTs are ordered a certain way, making operations faster to execute. However, as you saw in Lab 19.1, it is possible to receive a sequence of input that causes a BST to perform no better than an unsorted binary tree.

The next three labs will focus on a specific kind of BST, called the Red-Black Tree (RBT), which is a BST that uses a small amount of minor extra information at each node in order to keep the tree height near-optimal as the tree grows. In addition to requirements for BSTs, an RBT requires that:

1. A node is colored either red or black.
2. The root is always colored black.
3. All leaves are colored black.
4. A red node has only black children.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

Using these requirements, RBTs operate at near optimal speeds for insertion, deletion, and searching. We will add the functionality to the rbTreeType to make it enforce the above requirements.

## Objectives

In this lab, you will add a parent pointer to the node type, and add rotateLeft and

rotateRight methods to the original BST code.

## After completing this lab, you will be able to:

* Rotate a subtree of a binary search tree clockwise.
* Rotate a subtree of a binary search tree counterclockwise. Estimated completion time: **45–60 minutes**

# Steps: Adding Rotations to Binary Search Trees

1. Rotations are useful to help rebalance a subtree, which will be an essential operation for the Red-Black Tree to maintain optimal behavior. The first step is to modify the structure that contains the node information, nodeType, to contain a pointer to its parent. Open **redBlackTree.h** (attached). Add a field called plink, which will point to the parent of the structure. Now modify the insert method to make sure the plink field is properly set. *Also modify method* ***copyTree*** *to copy the pLink field of each node copied.*

Now that each node also points to its parent, you can implement rotations. You will now examine the preconditions and post-conditions involved in the rotations. Assume the subtree we would like to rotate is structured as follows:

* + The root of the subtree is node D.
  + The parent of node D is node P; D is the left sub-child of P.
  + LD is rooted at node B, and RD is rooted at node F.
  + LB is rooted at node A, which is a leaf node.
  + RB is rooted at node C, which is a leaf node.
  + LF is rooted at node E, which is a leaf node.
  + RF is rooted at node G, which is a leaf node. Call this subtree T1. Draw T1 as described above.

1. The rotateRight(D) method rotates the tree in a clockwise fashion about node D. After rotation, the subtree is structured as follows:
   * The root of the subtree is node B.
   * The parent of node B is node P; node B is the left subchild of P.
   * LB is rooted at node A, which is a leaf node.
   * RB is rooted at node D.
   * LD is rooted at node C, which is a leaf node.
   * RD is rooted at node F.
   * LF is rooted at node E, which is a leaf node.
   * RF is rooted at node G, which is a leaf node.

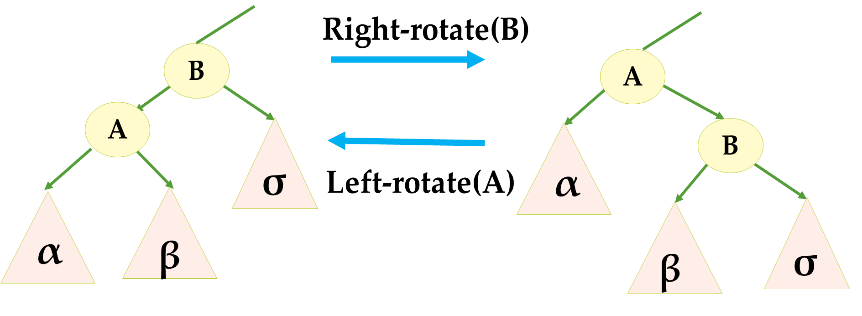
Let this subtree be called T2. Draw T2 as described above.

1. rotateLeft(B) will rotate the subtree about node B in a counterclockwise fashion. Calling rotateLeft(B) with the nodes structured as in T2 results in T1, which demonstrates the point that rotateLeft and rotateRight are inverses of each other. For illustrative purposes, suppose we call rotateLeft(D)on T1; this results in a subtree organized as follows:
   * The root of the subtree is node F.
   * The parent of node F is node P; node F is the left subchild of P.
   * RF is rooted at node G, which is a leaf node.
   * LF is rooted at node D.
   * RD is rooted at node E, which is a leaf node.
   * LD is rooted at node B.
   * RF is rooted at node C, which is a leaf node.
   * LF is rooted at node A, which is a leaf node. Let this subtree be T3. Draw T3 as described.
2. Now open the **redBlackTree.h** file (attached). Implement the rotateLeft and rotateRight methods. Add statements to the **testRBTree.cpp** file to test the added methods to verify that they operate correctly.

Note: the rotate functions should take a pointer to the node over which the rotation is to be done (call it the pivot) e.g.:

void rotateLeft( nodeType<elemType> \* pivot);

so in the diagram below if n is pointer to A and need to do a left-rotation over A, call the function: rotateLeft(n);



Both left and right rotate function should be private. For the purpose of testing and debugging these function, however, you need to define a public public function that calls the private functions with root as the pointer to the pivot i.e.

void rotateLeftTree()

{

rotateLeft(root);

}

1. Compile, link, and execute the file to generate results that convince you the rotation methods were implemented correctly (Note: you may need to make the rotation methods public to test them; however, typically they would be private).

# Lab 19.4 Coloring Nodes and Using the Sentinel

This lab continues modification of Binary Search Trees (BSTs) to adhere to the requirements of Red-Black Trees (RBTs):

1. A node is colored either red or black.
2. The root is always colored black.
3. All leaves are colored black.
4. A red node has only black children.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

The previous lab added rotation methods in order to rebalance the tree. This lab will add a color property to each of the nodes, which will allow you to enforce the requirements stated above.

This lab will also introduce a sentinel object that is used internally by the RBT class. The sentinel is used to provide a boundary for all of the links in the valid nodes of the tree, while still maintaining a color property (the sentinel is colored black). An RBT implementation may use null instead of a sentinel, but the logic is more complicated due to extra checking for the null value.

## Objectives

In this lab, you add a color property to the nodeType structure, and create a sentinel object that acts as the boundary for all links in the Red-Black Tree.

## After completing this lab, you will be able to:

* Modify existing binary search tree code to have extra bookkeeping properties.
* Use a sentinel object to form a logical boundary for the tree nodes, while allowing that boundary to share properties common to the nodes of the tree.

Estimated completion time: **30–40 minutes**

# Steps: Coloring Nodes and Using the Sentinel

1. The simpler task is to add coloring to all nodes; therefore, the first exercise is to add a color property to all nodes in the tree. Open **redBlackTree.cpp** (attached).

1a. The color property can take on the value RED or BLACK; therefore, creating an enum called COLOR that can take on these values is appropriate. Add the enum definition to the rbTreeType class.

1b. Now add a field of type COLOR to the nodeType structure.

1c. Modify the insert method to assign the color RED to all newly added nodes.

1d. Modify method **copyTree** to copy the color field of each node.

~~During deletion if a node’s info is copied, its color field shouldT be copied as well.~~

1e. Make sure that when a node is printed, it also now displays its color. Remember that you can’t display an enum value using the << operator. So you need to use an if (or switch) statement:

if (color == RED)

cout << “RED”;

etc.

1. The second task is to use a sentinel to act as a logical bound for all nodes inserted into the tree. The sentinel is a node; however, it is created when the tree is created, and only deleted when the tree is deleted.

2a. Create an internal variable called sentinel that is a pointer to a nodeType. It would be easier to do this in class binaryTreeType. You may also add a Boolean field on nodeType that would be set to true of a node is a sentinel and false otherwise.

struct nodeType

{

elemType info;

nodeType<elemType> \*lLink;

nodeType<elemType> \*rLink;

nodeType<elemType> \*pLink;

bool sentinelFlag;

ColorType color;

};

Note: You don’t necessarily need both the sentinel pointer in class binaryTreeType and the sentinel flag in nodeType. However, if it makes your algorithms less complex, then you can use both.

2b. The sentinel pointer member of class binaryTreeType should be allocated a node in the constructor of the binaryTreeType class. Its color field should be BLACK, and all links should point to itself.

sentinel = new nodeType;

sentinel->color = BLACK;

sentinel->lLink = sentinel->rLink = sentinel->pLink = sentinel;

sentinel->sentinelFlag = true;

2c. Modify the destructor of binaryTreeType class to delete the sentinel node after all other nodes are deleted.

2d. Modify the other methods of the rbTree class to use the sentinel to indicate “no node” instead of null. This includes:

* 1. Inserting a new node: left and right links should point to sentinel and sentinelFlag set to false.
  2. Searching the tree: stop search when you hit a sentinel (instead of nullptr)
  3. Deleting a node from the tree: same as above
  4. Left and right rotations

You must also guarantee that the plink field is properly set during the modifying operations (this was added in Lab 19.2). Upon insertion, when a child link is set, the parent link should also be set. Upon deletion, if a node is swapped with another (again, a child link is set), then the parent link must be updated as well.

Modify the traversal function to exit the loop when a sentinel node is encountered. For testing purpose, have the traversal functions display the color of each in addition to the info field. When a node is a sentinel, just output “leaf”

1. If the traversals are modified correctly, the sentinel will not be printed. Modify the **testRBTree.cpp** file to print out the sentinel variable separately, to check for correctness. After you confirm the correct state of the sentinel, you should make it private to the rbTree. Be careful not to let the print of the sentinel recursively check its links, as it links to itself.
2. Use the **testRBTree.cpp** file to test the modified rBlackTreeType class. Compile, link, and execute the file. All nodes added to the tree externally should be colored RED. The sentinel should be colored BLACK.

Demo: For the purpose of demoing use the function main() below. Note that up to this task, the insertFixup() and deleteFixup() are not implemented yet. So the tree will not be red-black tree but it is a binary search tree.

int main()

{

// Implement code to create, populate, and manipulate

// a red-black tree here.

rBlackTreeType<int> tree1;

tree1.insert(30);

tree1.insert(35);

tree1.insert(13);

tree1.insert(22);

tree1.insert(42);

tree1.insert(28);

tree1.insert(23);

tree1.insert(16);

tree1.insert(25);

cout << endl;

tree1.preorderTraversal();

tree1.rotateLeftTree();

tree1.preorderTraversal();

cout << endl;

system("pause");

return 0;

}

# Lab 19.5 RBT Insertions and Deletions

In the previous two labs, we modified a Binary Search Tree (BST) to be able to rotate around nodes, make use of the sentinel object, and use colors as an additional field for all nodes.

This final lab completes the modification of the BST to a fully functional Red-Black Tree (RBT), which maintains near-optimal speeds for insertions, deletions, and searches over the tree.

Recall that the RBT must maintain the following properties:

1. A node is colored either red or black.
2. The root is always colored black.
3. All leaves are colored black.
4. A red node has only black children.
5. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

## Objectives

In this lab, you will re-implement the insert and delete methods to ensure that the properties required of an RBT are maintained.

## After completing this lab, you will be able to:

* Fully understand the functionality of an RBT.
* Know how to insert into an RBT while maintaining the properties of the RBT.
* Know how to delete from an RBT while maintaining the properties of the RBT. Estimated completion time: **75–90 minutes**

# Steps: RBT Insertions and Deletions

There are two major operations that need to be properly implemented to maintain the RBT properties: insert and delete. We begin with insert.

1. The insert method itself is already nearly complete. At the bottom of the insert method, add a call to insertFixup(n), assuming *n* is the node just added to the tree. The work to maintain the RBT properties will be done in the insertFixup method.
2. Create the new private method

void insertFixup(nodeType<elemType>\*&n)

This method will ensure that the newly inserted node does not violate the RBT properties. The method must consider five discrete cases when the new node is inserted. To facilitate the descriptions below, let N be the newly inserted node, P be the parent of N, and refer to the left and right children of N as LN and RN, respectively. To simplify the explanation, we also define G as the grandparent (parent of the parent) of N, and U as the “uncle” of N (sibling to the parent of N). We now describe the cases that must be handled, in the order shown, in insertFixup:

* 1. N is the root of the tree. N needs only to be repainted BLACK.
  2. P is the parent of N, and is colored BLACK. No further action needs to be taken.
  3. Both P and U are RED. G is repainted RED and both P and U are painted BLACK. In order to maintain that G does not violate any RBT properties, we now call insertFixup(G).
  4. P is RED, and U is BLACK. There are two subcases to consider here:

Note: P = parent of N, G is grand parent of N, U is uncle of N.

* + 1. N = RP, and P = LG. 🡺 Left rotate P then set N to LN
    2. N = LP, and P = RG 🡺 Right rotate P then set N to RN

After rotation, go to case V.

* 1. P is RED, and U is BLACK.

Note: P = parent of N, G is grand parent of N, U is uncle of N.

Set P color to BLACK and G color to RED

There are two subcases:

* + 1. N = LP, and P = LG. **Right rotate** G.
    2. N = RP, and P = RG. **Left rotate** G. Switch the colors of P and G.

1. Inserts are now complete. We now focus on creating deleteFixup to guarantee that deletion maintains the RBT properties as well. The signature of the new method is void deleteFixup(nodeType<elemType>\*&n).
2. Define a helper method sibling(n), which returns the other node (not N) of N’s parent node. We define N’s sibling as S.
3. The deleteFromTree method requires a call to deleteFixup. This should occur directly before the delete current; statement at the end of deleteFromTree. The call should only take place if the color of current is BLACK, since deleting a RED node does not violate any RBT properties. The replacement node for current (i.e., p) should be the passed argument to the fixup as follows: deleteFixup(p).
4. We now implement deleteFixup(p). The method is best implemented as a cascading call of the different cases that may be handled, as the cases are not mutually exclusive (e.g., case 1 may be followed by case 2). Therefore, define the help functions deleteFixup2 – deleteFixup6. Each function will handle a fall- through case (where N is the node passed to the method):
5. deleteFixup: If N is the root (i.e., P is the sentinel), do nothing as no properties are violated. Otherwise call deleteFixup2 on N.
6. deleteFixup2: If the color of S is RED, set the color of P to RED, and set the color of S to BLACK. If N = LP, rotate left over P. Otherwise, rotate right over P.

N is unconditionally passed to deleteFixup3.

1. deleteFixup3: If the color of P, S, LS, and RS are all BLACK, repaint S to RED, and call deleteFixup on P. Otherwise, pass N to deleteFixup4.
2. deleteFixup4: If the color of P is RED, but the colors of S, LS, and RS are BLACK, repaint S as RED and repaint p as BLACK. Otherwise pass N to deleteFixup5.
3. deleteFixup5: If the color of S is BLACK, take one of the two following actions:
   1. If N = LP, the color of RS is BLACK, and the color of LS is RED, then repaint S to RED and repaint LS to BLACK. Rotate right around S.
   2. If N = RP, the color of LS is BLACK, and the color of RS is RED, then repaint S to RED, repaint RS to BLACK, and rotate left around S.

Node N is unconditionally passed to deleteFixup6.

1. deleteFixup6: S takes on the color of P. P is repainted BLACK. If N = LP, repaint RS to BLACK and rotate left around P. Otherwise, repaint LS to BLACK and rotate right around P.
2. The insert and delete methods have now been modified to maintain the RBT properties. Compile, link, and test the modified code against **testRBTree.cpp** to determine if the data structure still operates correctly.
3. ***Challenge Exercise:***Modify the search method to print out the number of steps taken to find a particular node. Compare the search times between the original Binary Search Tree and the Red-Black Tree.