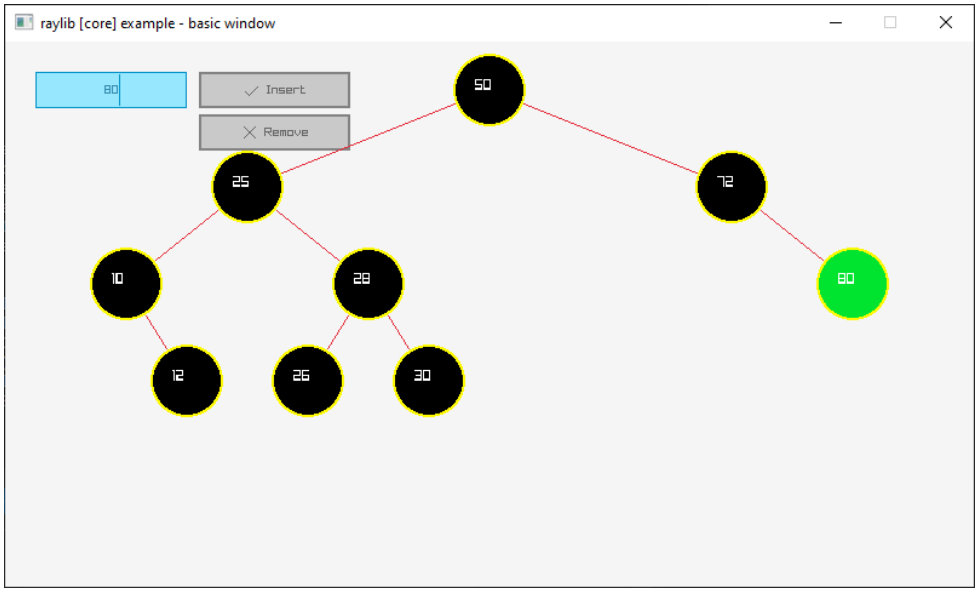
Tutorial – Binary Trees

Introduction:

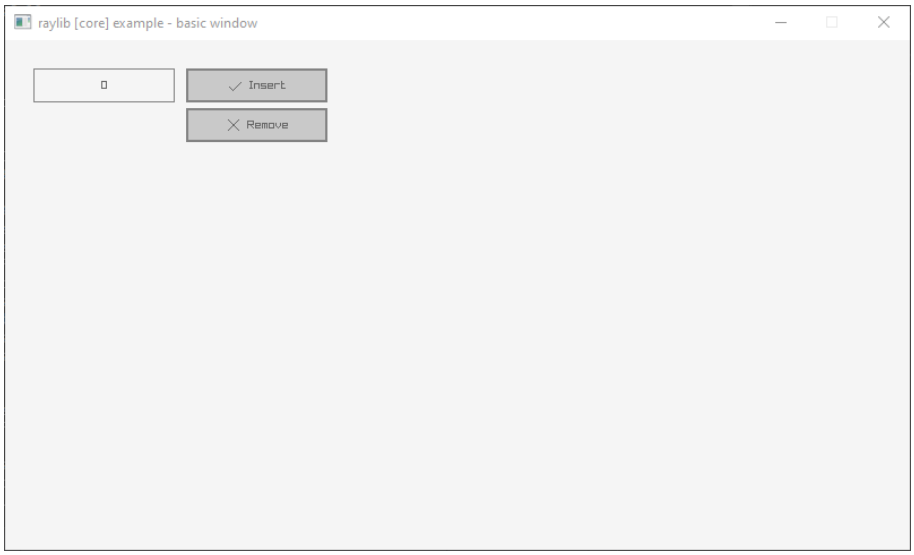
In this tutorial, we will be implementing a binary tree data structure and binary tree search algorithm.

While we could complete this exercise by making a plain old console application, but to visualize our tree structure and data we will be using the RayLib framework with the RayGUI library.



To begin, download the starter project from the *AIE Year 1 Samples* git repository: https://github.com/AcademyOfInteractiveEntertainment/AIEYear1Samples

This will give you a basic application that contains some GUI components you can use to add and remove nodes from your binary tree. (You can add another button for searching as an extension)



The Tree Node Class:

There are two major ways to implement a binary tree. The first is using a class or structure for the node and storing a pointer to both the left and the right branch. This is the approach we will be using in this tutorial. This approach is sometimes referred to as a *Linked Binary Tree*.

The advantage of a linked binary tree is that the maximum tree size is limited only by the total memory available. The disadvantage being that we need 2 pointers (one for each branch), plus a third for the data itself (or, if we're storing simple data types, the actual data itself). In this tutorial we'll be storing integers in our tree, which means the tree structure takes up 3 times the amount of space of the data we are storing.

The other way we can implement a binary tree is using an array. This approach is typically called an *Arrayed Binary Tree*. In this approach, the root is stored at index 0. The *left* child of any index *i* is stored at index *i*2*, while the right child is stored at index *(i*2)+1*. Thus, each node in our tree has a set, pre-defined and calculatable index in our array.

The advantage of this technique is that it is relatively fast to traverse and saves much more disk space. But adding nodes that go beyond the array capacity will require the array to be resized.

To find out more about *Arrayed Binary Trees*, you can refer to the following web article: https://www.gamedev.net/resources/_/technical/general-programming/trees-part-2-binary-trees-r1433

To implement our *Linked Binary Tree*, create a new class called *TreeNode* and implement the following interface (this matches the TreeNode class presented in the lecture slides):

```
class TreeNode
{
public:
	TreeNode(int value);
	~TreeNode();

	bool HasLeft() { return (m_left != nullptr); }
	bool HasRight() { return (m_right != nullptr); }

	int GetData() { return m_value; }
	TreeNode* GetLeft() { return m_left; }
	TreeNode* GetRight() { return m_right; }

	void SetData(int value) {  }
	void SetLeft(TreeNode* node) {  }
	void SetRight(TreeNode* node) {  }

	void Draw(int x, int y, bool selected=false);

private:
	// this could also be a pointer to another object if you like
	int m_value;

	// node's children
	TreeNode* m_left;
	TreeNode* m_right;
};
```

As you can see, most of this class is implemented in the header file and consists of getters and setters.

The draw function is simple and uses the RayLib framework to draw a graphical representation of the node. (Note, the node holds no information regarding where it is in the tree or where it should be drawn. This must be determined by the Binary Tree itself).

```
void TreeNode::Draw(int x, int y, bool selected)
{
	static char buffer[10];

	sprintf(buffer, "%d", m_value);

	DrawCircle(x, y, 30, YELLOW);

	if (selected == true)
		DrawCircle(x, y, 28, GREEN);
	else
		DrawCircle(x, y, 28, BLACK);


	DrawText(buffer, x - 12, y - 10, 12, WHITE);
}
```

The Binary Tree Class:

Here is the interface for the binary tree class:

```
class BinaryTree
{
public:
	BinaryTree();
	~BinaryTree();

	bool IsEmpty() const;
	void Insert(int a_nValue);
	void Remove(int a_nValue);
	TreeNode* Find(int a_nValue);

	void PrintOrdered();
	void PrintUnordered();

	void Draw(TreeNode* selected = nullptr);

private:
	//Find the node with the specified value.
	bool FindNode(int a_nSearchValue, TreeNode*& ppOutNode, TreeNode*&
ppOutParent);

	//Used to recurse through the nodes in value order and print their values.
	void PrintOrderedRecurse(TreeNode*);
	void PrintUnorderedRecurse(TreeNode*);


	void Draw(TreeNode*, int x, int y, int horizontalSpacing, TreeNode* selected
= nullptr);

	//The root node of the tree
	TreeNode* m_pRoot;
};
```

The only function we'll give you is the draw function:

```
void BinaryTree::Draw(TreeNode* selected)
{
    Draw(m_pRoot, 400, 40, 400, selected);
}

void BinaryTree::Draw(TreeNode* pNode, int x, int y, int horizontalSpacing,
TreeNode* selected)
{
    horizontalSpacing /= 2;
    if (pNode)
    {
        if (pNode->HasLeft())
        {
            DrawLine(x, y, x - horizontalSpacing, y + 80, RED);
            Draw(pNode->GetLeft(), x - horizontalSpacing, y + 80,
horizontalSpacing, selected);
        }

        if (pNode->HasRight())
        {
            DrawLine(x, y, x + horizontalSpacing, y + 80, RED);
            Draw(pNode->GetRight(), x + horizontalSpacing, y + 80,
horizontalSpacing, selected);
        }
        pNode->Draw(x, y, (selected == pNode));
    }
}
```

The *draw()* function is a recursive function that will step through the left, then the right branch of the tree. After drawing the branches, the function then draw the current node.

We've also used some magic numbers to ensure the nodes are evenly spaced according to our set screen dimensions.

The constructor will simply set the root to null. The *isEmpty()* function is similarly simple, and returns true if the root is null.

Below is pseudo code for each binary tree method you'll need to implement.

insert(int value)

If the tree is empty, the value is inserted at the root
Set the current node to the root
While the current node is not null
    If the value to be inserted is less than the value in the current node
        Set the current node to the left child and continue
    If the value to be inserted is greater than the current node
        Set the current node to the right child and continue
    If the value to be inserted is the same as the value in the current node
        The value is already in the tree, so exit
end While

```
Get the parent of the current node (before it was set to null)
If value to be inserted is less than parent
        insert value as left child node
otherwise insert value as right child node
```

findNode(int a_nSearchValue, TreeNode*& ppOutNode, TreeNode*& ppOutParent)

The *find()* function could be implemented as a recursive function or using a while loop. If you find the former easier, you may wish to modify your class accordingly.

```
Set the current node to the root
While the current node is not null
        if the search value equals the current node value,
                return the current node and its parent
        otherwise
                If the search value is less than the current node
                        set the current node to the left child
                otherwise set the current node to the right child
end While
If the loop exits, then a match was not found, so return false
```

The pointer references in the function declaration may seem confusing. Remember, you are passing in a reference to a pointer. We do this so that we can change the value of the pointer, and this change persists after the function exits (we're changing the value of the pointer).

If we passed in a plain pointer, the value of the pointer is copied to the heap. This means as we update the pointer during the processing of the function, we're changing the value of this copy. When the function exits, our changes would be lost.

If you have trouble understanding this concept, be sure to do an internet search or ask your teacher to go over it.

Why do we need to return both the current node (ppOutNode) and its parent (ppOutParent)? The parent isn't necessary when we're simply searching for a value in our tree. But we do want this information when we *delete* a node from our tree.

When removing a node, we must find the node, delete it, then clean up the pointers of the parent. So to save us writing the search algorithm twice, we include the parent information as output for use during the *remove()* function.

remove(int value)

find the value in the tree, obtaining a pointer to the node and its parent
If the current node has a right branch, then
find the minimum value in the right branch by iterating down the left branch of the current node's right child until there are no more left branch nodes
copy the value from this minimum node to the current node
find the minimum node's parent node (the parent of the node you are deleting)
if you are deleting the parent's left node
set this left child of the parent to the right child of the minimum node
if you are deleting the parent's right node
set the right child of the parent to the minimum node's right child

If the current node has no right branch
if we are deleting the parent's left child, set the left child of the parent to the left child of the current node
If we are deleting the parent's right child, set the right child of the parent to the left child of the current node
If we are deleting the root, the root becomes the left child of the current node

With these functions completed, you should have a working binary tree.

The final step is to draw the graphical representation of our binary tree and test the insert, remove, and find functions.

Testing the Binary Tree:

You should be able to compile and run your application. Use the GUI to add, remove and search for nodes in your binary tree to insure your binary tree class is correct.