How to Code a Binary Search Tree in C++

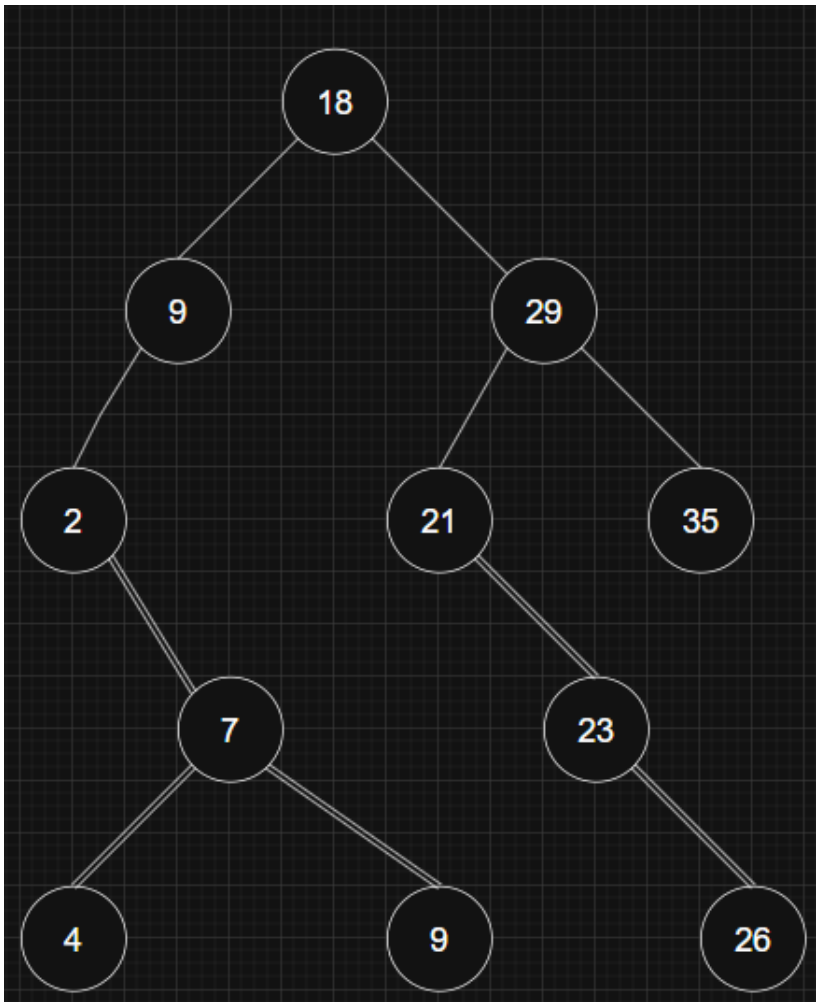

Figure 0: Visual Representation of a Binary Search Tree

Introduction:

The purpose of this document is to inform the audience how to create a binary search tree. This document is intended for computer science students and junior developers. The instructions are written for those new to data structures but have previous experience with basic programming fundamentals. Therefore, it is assumed you are familiar with variable declarations, loops, conditional statements, functions, structures, and pointers.

Description:

The instructions below demonstrate how to code a Binary Search Tree (BST). A BST is a node-based data structure that enables efficient dynamic data storage and retrieval. A node is a very basic unit that contains a key (or value) and points to at most two child nodes.

List of Materials:

- A computer with a working operating system.
- Keyboard, Mouse, and Monitor.
- Visual Studio 2022 Community Edition (https://visualstudio.microsoft.com/vs/).

Safety Information:

Remember when allocating memory with the "new" keyword in C++ to deallocate memory later with the C++ keyword "delete". Failing to remember this best practice leads to memory leaks and could potentially lead to crashes. This code is for educational purposes only, not for implementing into production environments. Handle production code with care.

Directions:

1. Create a new project.

   Note: Microsoft's Visual Studio 2022 Community Edition was used for these instructions. This software is a powerful, integrated development environment (IDE) designed for programmers working with C++ and many other programming languages. Visual Studio is an ideal platform for this project because it offers an environment to build, debug, and test code.

   a. Launch Visual Studio 2022.

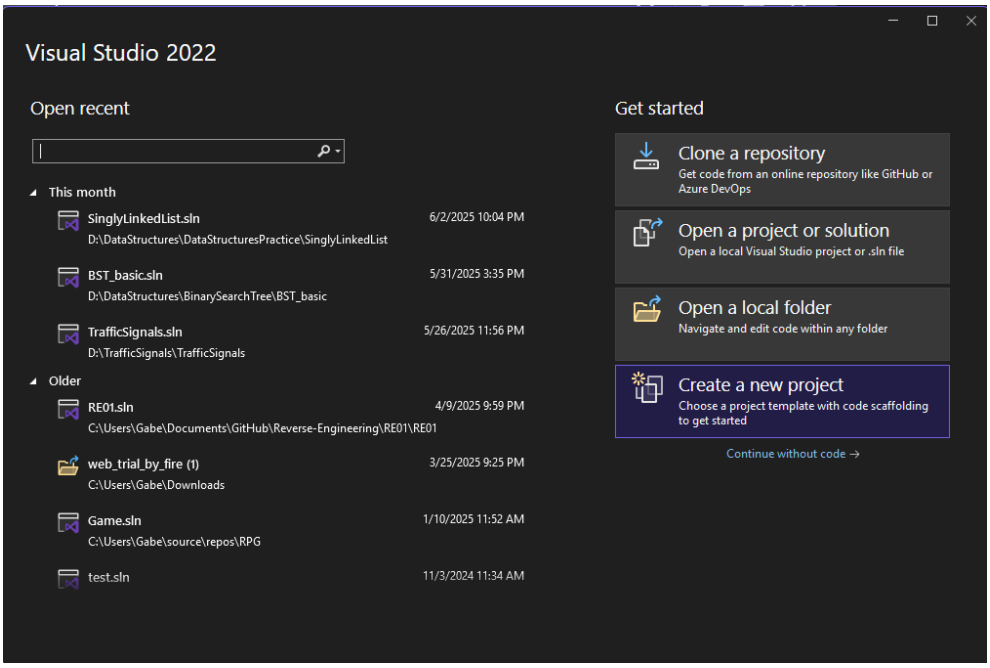

Figure 1: Opening Visual Studio and Creating a New Project

b. **Select** “Create a new project”.
    i. Highlighted in Figure 1 for reference.
c. **Double** click “Empty Project”.
d. Give the project file a proper name, for example BinaryTree.
e. **Select** a file location.

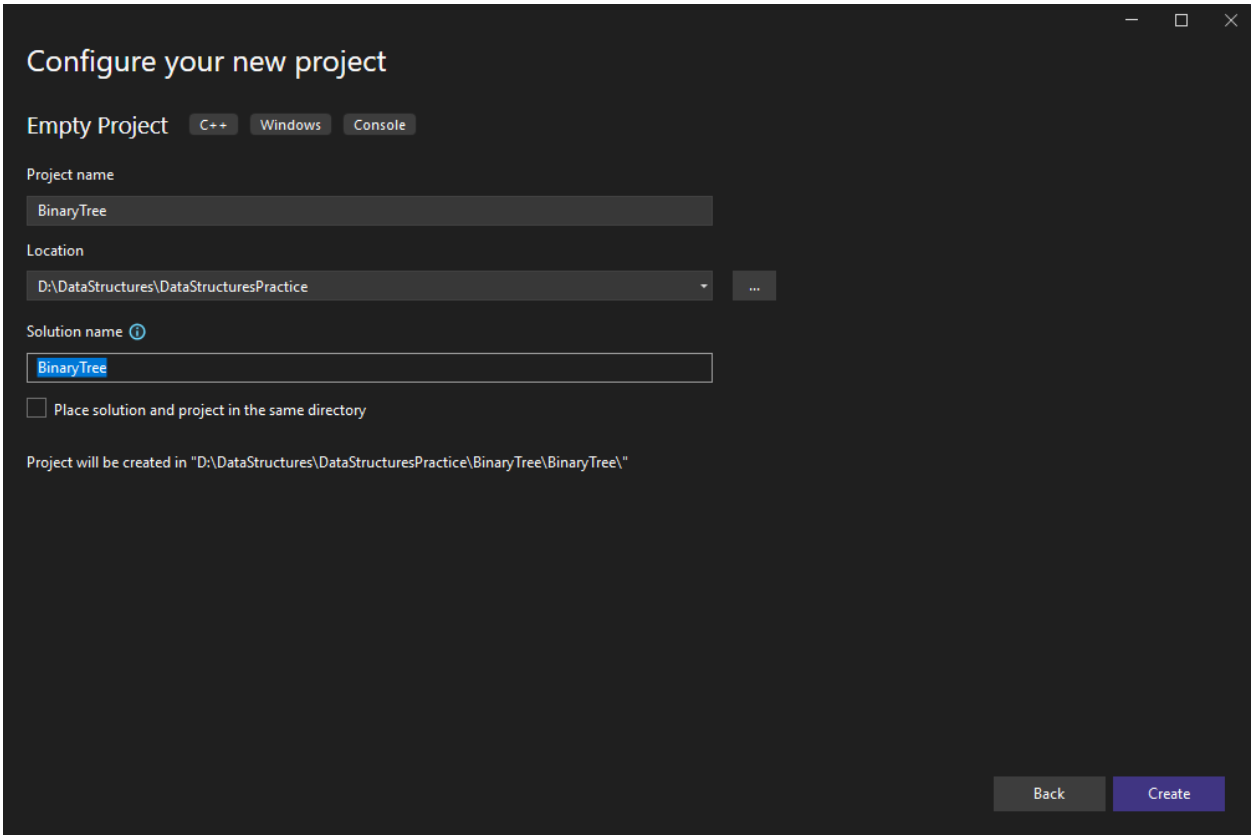

Figure 2: Visual Studio Project Name and File Location

2. Create a main file.

> **Note:** These instructions will include all structure definitions and function implementations inside the source file instead of being split into a header file. Due to the smaller size of the project, all the code can be written inside one file.

a. In the Solutions Explorer **right click** **“**Source Files” and **select** **“**add”, and in the pop-up menu **select** “New Item”.
    i. In Figure 3, the Solutions Explorer is shown on the left, and selections mentioned above are highlighted.

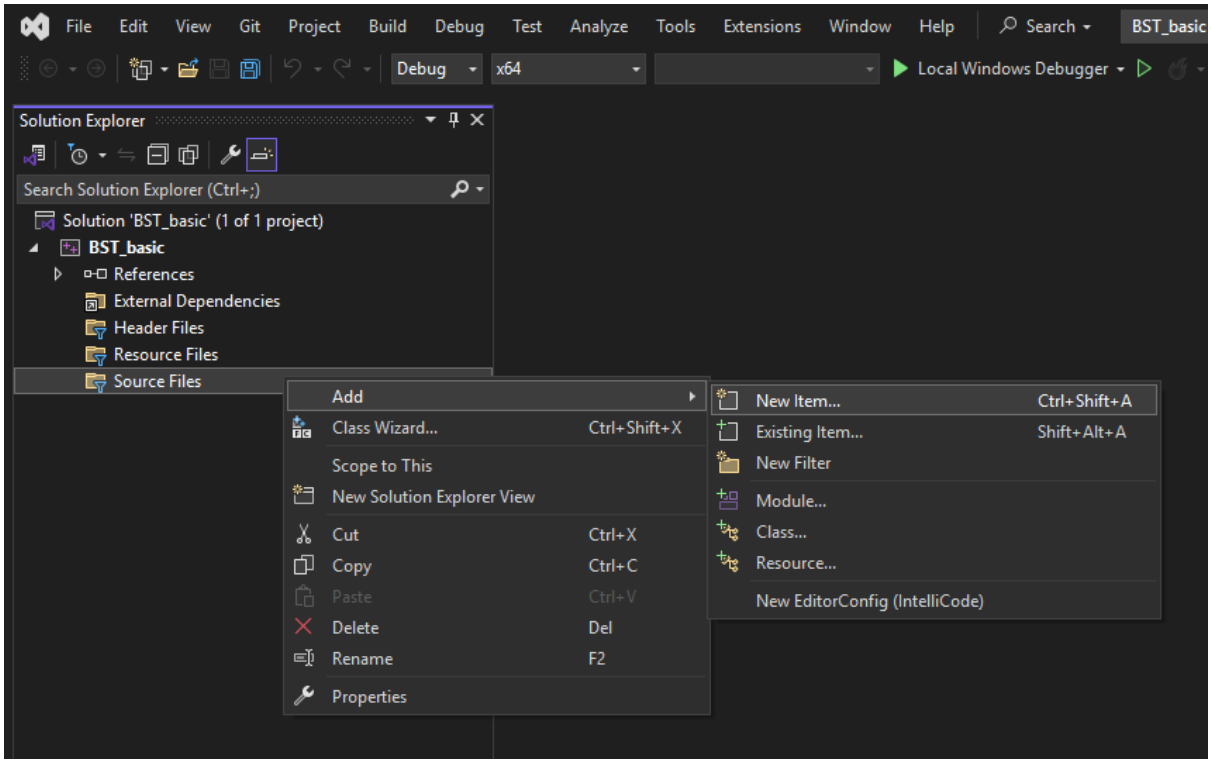

Figure 3: Adding a Source File Inside the Solutions Manager

b. In the window that pops up **select** C++ File (.cpp) and name the file appropriately, for example main.cpp.

Note: For these instructions I will be using the C++ language, but you can use any language, and the concepts and sudo code will still apply.

3. Inside your main file, create a node.

Note: It is a best practice in programming to divide larger problems into smaller problems. In this case before coding an entire Binary Search Tree a small part of the tree is coded and then built on to create an entire tree.

a. To create a node, you first need to define a structure.
b. Define what the node's main characteristics are going to be.
    i. In this case; left pointer, right pointer, and key.
c. Give the struct a default constructor.

BST_basic

```
1   #include <iostream>
2   using namespace std;
3
4   struct Node {
5       //define node properties
6       int key;
7       Node* left;
8       Node* right;
9
10      //constructor
11      Node(int item) {
12          key = item;
13          left = right = NULL;
14      }
15  };
16
17  int main() {
18
19      return 0;
20  }
21
```

Figure 4: Basic Node Outline

d. Nodes are the very basic unit or building block of BSTs. Nodes hold a key (or value) and at most two pointers to child nodes.
    i. In Figure 4, there is a struct definition for a basic node. The data the node holds will be assigned to the key variable. The right and left child nodes are declared as pointers, so movement through the tree later will be easy. Lastly, to ensure a valid configuration upon a new node's creation, there is a constructor.

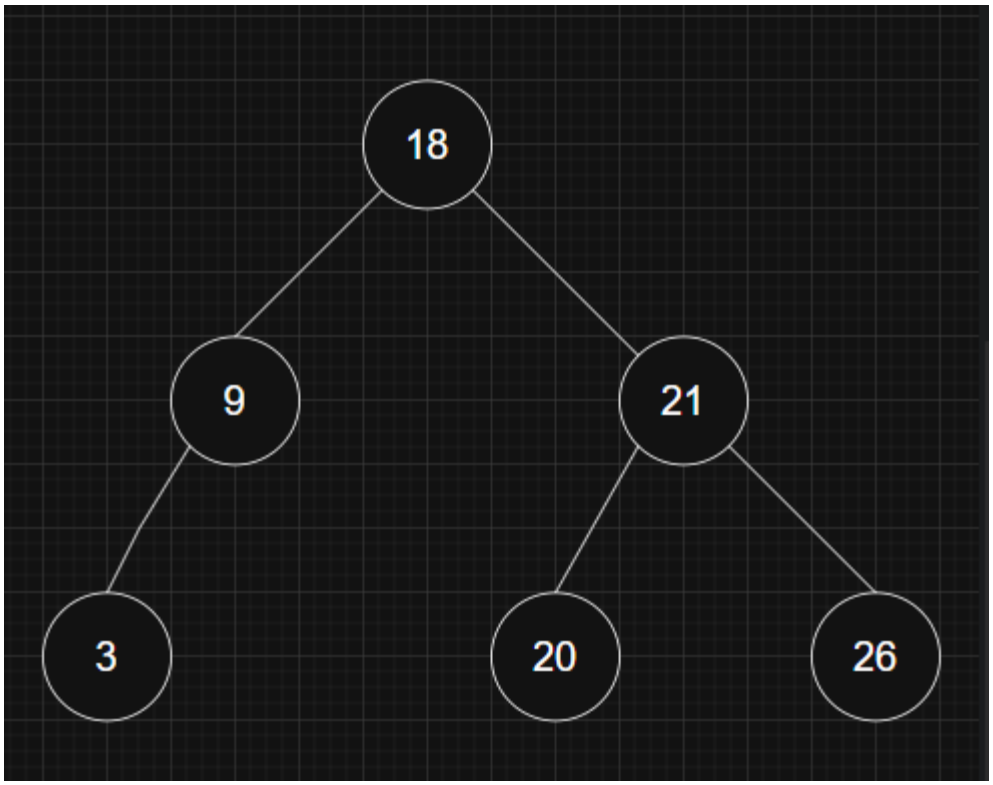

Figure 5: A Basic Binary Search Tree Example

e. In Figure 5 above, you can see there are three levels.

4. Create an insertion helper function.

Note: Example code provided in Figure 6.

a. Initialize a reference node with the root node.
b. Compare the reference node's key to the key to be inserted.
c. Move right in the BST if the reference node is less than key to be inserted.
d. Move left if the reference node's key is greater than the key to be inserted.
e. Steps c & d need to be repeated until a leaf node is reached, so you can use recursion.

```
//A helper function to insert a new node
Node* insert(Node* node, int key) {

    //endcase -> empty tree, return a new node
    if (node == NULL) {
        return new Node(key);
    }

    //endcase -> If the key is already present, return the node
    if (node->key == key) {
        return node;
    }

    //move left or right
    if (node->key < key) {
        node->right = insert(node->right, key);
    }
    else {
        node->left = insert(node->left, key);
    }

    return node;
```

Figure 6: Insertion function

Caution: New memory was allocated, and the cleanup for memory previously allocated for the empty tree was disregarded.

5. Create a traversal function.
    a. Inorder Traversal.
        i. Travel all the way to the left node.
        ii. Visit the root node and output the keys.
        iii. Travel all the way to the right node.

```
//Inorder Traversal
void printInorder(Node* node) {
    //check endcase
    if (node == NULL) {
        return;
    }

    //move left in tree
    printInorder(node->left);

    //output key
    cout << node->key << " ";

    //move right
    printInorder(node->right);
}
```

Figure 7: Inorder Traversal Function

   b. Preorder Traversal.
   c. Postorder Traversal.
6. Create a search helper function.
   a. First check if the root is null or equal to the key you are searching for.
   b. Then use a conditional statement to see if the key you are searching for is greater than the reference key.
   c. Use another conditional statement to cover if the key you are searching for is less than the reference key.

```
//Search function
Node* search(Node* root, int key) {

    //check endcase
    if (root == NULL || root->key == key) {
        return root;
    }

    //key is greater than reference key
    if (root->key < key) {
        return search(root->right, key);
    }

    //key is less than ref key
    return search(root->left, key);
}
```

Figure 8: A BST Search Function

7. Create a deletion helper function.
   a. Case 1: Node with only right child or no child.
      i. Create a temporary node and assign it to the right child.
      ii. Delete the original node.
         1. It is best practice and safer to delete these nodes when they are no longer being used. Leaving them could cause memory to run out very fast.
      iii. Return the temporary node.
   b. Case 2: Node with only left child.
      i. Create a temporary node and assign it to the child on the left.
      ii. Delete the original node.
      iii. Return the temporary node.
   c. Case 3: Both children are present.
      i. Find the node's successor.
      ii. Assign the current nodes key with the successor's key.
      iii. Delete the right child node.

```
//Delete Function
Node* delNode(Node* node, int key) {

    //If the root node does not have a key, just return it
    if (node == NULL) {
        return node;
    }

    if (node->key > key) {
        node->left = delNode(node->left, key);
    }
    else if (node->key < key) {
        node->right = delNode(node->right, key);
    }
    else {
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        }

        if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }

        Node* succ = getSuccessor(node);
        node->key = succ->key;
        node->right = delNode(node->right, succ->key);
    }
    return node;
```

Figure 9: A BST Deletion Function

8. Create a validation function (isBST).
    a. Check if the reference node's key is within the given range.
        i. If it is greater than the maximum return, it is false.
        ii. If it is less than the minimum return, it is false.
        iii. Use recursion to check subtrees.

```cpp
//Function to check BST within a given range
bool isBST(Node* node, int min, int max) {
    if (node == nullptr) {
        return true;
    }

    if (node->key < min || node->key > max) {
        return false;
    }
    return isBST(node->left, min, node->key - 1) &&
        isBST(node->right, node->key + 1, max);
}
```

Figure 10: Check function

9. Balance the BST.
    a. Create a function that sorts the tree nodes in order with an array.
    b. Build a balanced tree from a sorted array.
    c. Make a function that utilizes both functions created above.

```cpp
//sort a tree with inorder traversal
void storeInorder(Node* node, vector<int>& nodes) {
    if (node == nullptr) {
        return;
    }
    storeInorder(node->left, nodes);
    nodes.push_back(node->key);
    storeInorder(node->right, nodes);
}

//Builds a balanced BST from sorted array
Node* buildBalancedTree(vector<int>& nodes, int start, int end) {
    if (start > end) {
        return nullptr;
    }
    int mid = (start + end) / 2;
    Node* node = new Node(nodes[mid]);

    node->left = buildBalancedTree(nodes, start, mid - 1);
    node->right = buildBalancedTree(nodes, mid + 1, end);

    return node;
}

//fucntion to balance a BST
Node* balanceBST(Node* node) {
    vector<int> nodes;
    storeInorder(node, nodes);
    return buildBalancedTree(nodes, 0, nodes.size() - 1);
}
```

Figure 11: Balancing a BST

10. Find Minimum and Maximum.
    a. Case minimum:
        i. Check with a loop if the node's left child exists.
    b. Case maximum:
        i. Check with a loop if the node's right child exists.

```
//Find min
Node* findMin(Node* node) {
    if (node == nullptr) {
        return nullptr;
    }
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

//Find max
Node* findMax(Node* node) {
    if (node == nullptr) {
        return nullptr;
    }
    while (node->right != nullptr) {
        node = node->right;
    }
    return node;
}
```

Figure 12: Maximum & Minimum

11. Level-Order Traversal (BFS).
    a. Create an empty queue.

    Note: A queue is another data structure used to store and manage data. A queue follows a principle called First in, First out (FIFO). Although this data structure was not included in knowledge assumed to be known by the reader, it is out of the scope for these instructions. Simply for the reason that it would take too long to cover.

    b. Drop the root node into the queue.
    c. Use a loop to check the condition; while loop is not empty.
        i. Inside the loop, remove the current node from the queue.
        ii. Then drop the left child node into the queue.
        iii. Then drop the right child node into the queue.
12. Calculate Depth of BST.
    a. Use recursion to call the left child and then the right.
    b. Print the node.
13. Count Nodes / Leaves in BST.

    a. Use a conditional statement to check if the left child exists and the right child exists.
    b. Then add one to a counter variable.
    c. Lastly, move nodes.
14. Output a visual.
    a. Print the root node then move to the next line.
    b. Print the previous node's left child first, then the right child.
    c. Repeat the previous step with all nodes.

    Caution: printing to console inside a recursive traversal could result in excessive output.

15. Unit testing.
    a. Create test functions for all helper functions to ensure that independent code is working before using them in the main function.
    b. Inside test functions make sure to test for all outcomes.

Works Cited

Haggard, Gary, et al. *Discrete Mathematics for Computer Science*. Thomson Brooks/Cole, 2006.

Vahid, Frank, and Roman Lysecky. "Programming in C." zyBooks, www.zybooks.com/catalog/programming-in-c-2/.