Project: Binary Search Tree

Samael Newgate

University of Advancing Technology

Documentation

**Binary Tree:** A binary tree is a hierarchical data structure where each node can only contain up to two children maximum. These children are the node’s left and right sub nodes when following the data structure. These nodes don’t have to follow any order.



**Binary Search Tree:** Unlike the binary tree, the binary search tree has a particular order it has to follow. After the root, the top value in the tree(8 is the root in the case of the following example). Any value that is lower goes to the left, any value that is higher goes to the right. This means only values that can be used in a greater than or lower than manner can be inserted. Because of this, binary search trees provide a fast and efficient way of sorting and retrieving data.

200px-Binary_search_tree.svg

**Binary Search Tree Class Structure:**

template<typename T> class bst

{

public:

struct node {

node\* parent;

node\* left\_child;

node\* right\_child;

T data;

node(T new\_data)

{

parent = nullptr;

left\_child = nullptr;

right\_child = nullptr;

data = new\_data;

}

};

bst()

{

root = nullptr;

}

~bst()

{

while (root != nullptr)

{

remove(root->data);

}

}

node\* get\_root()

{

return root;

}

bool insert(T data\_insert)

{

if (root == nullptr)

{

node\* new\_node = new node(data\_insert);

root = new\_node;

return true;

}

else

{

if (data\_insert > root->data)

{

return insert\_helper(data\_insert, root->right\_child, root);

}

else if (data\_insert < root->data)

{

return insert\_helper(data\_insert, root->left\_child, root);

}

else

{

return false;

}

}

}

node\* find(T data\_find)

{

if (root == nullptr)

{

return nullptr;

}

else

{

node\* current = root;

while(current != nullptr && current->data != data\_find)

{

if (current->data > data\_find)

{

current = current->left\_child;

}

else

{

current = current->right\_child;

}

}

return current;

}

}

void remove(T data\_delete)

{

node\* delete\_node = find(data\_delete);

if (delete\_node == nullptr)

{

return;

}

else

{

if(delete\_node->right\_child == nullptr && delete\_node->left\_child == nullptr) {

if (delete\_node == root)

{

root = nullptr;

delete delete\_node;

}

else if(delete\_node->data > delete\_node->parent->data)

{

delete\_node->parent->right\_child = nullptr;

delete\_node->parent = nullptr;

delete delete\_node;

}

else

{

delete\_node->parent->left\_child = nullptr;

delete\_node->parent = nullptr;

delete delete\_node;

}

}

else if (delete\_node->right\_child != nullptr && delete\_node->left\_child == nullptr)

{

if (delete\_node == root)

{

root = delete\_node->right\_child;

delete\_node->right\_child = nullptr;

delete delete\_node;

}

else if (delete\_node->data > delete\_node->parent->data)

{

delete\_node->parent->right\_child = delete\_node->right\_child;

delete\_node->parent = nullptr;

delete\_node->right\_child = nullptr;

delete delete\_node;

}

else

{

delete\_node->parent->left\_child = delete\_node->right\_child;

delete\_node->parent = nullptr;

delete\_node->right\_child = nullptr;

delete delete\_node;

}

}

else if (delete\_node->right\_child == nullptr && delete\_node->left\_child != nullptr)

{

if (delete\_node == root)

{

root = delete\_node->left\_child;

delete\_node->left\_child = nullptr;

delete delete\_node;

}

else if (delete\_node->data > delete\_node->parent->data)

{

delete\_node->parent->right\_child = delete\_node->left\_child;

delete\_node->parent = nullptr;

delete\_node->left\_child = nullptr;

delete delete\_node;

}

else

{

delete\_node->parent->left\_child = delete\_node->left\_child;

delete\_node->parent = nullptr;

delete\_node->left\_child = nullptr;

delete delete\_node;

}

}

else if(delete\_node->right\_child != nullptr && delete\_node->left\_child != nullptr)

{

node\* replacement\_node = find\_min\_branch(delete\_node->right\_child);

if (replacement\_node->data > replacement\_node->parent->data)

{

replacement\_node->parent->right\_child = nullptr;

}

else

{

replacement\_node->parent->left\_child = nullptr;

}

delete\_node->data = replacement\_node->data;

delete replacement\_node;

}

else

{

return;

}

}

}

T maximum()

{

if (root == nullptr) {

return NULL;

}

else

{

node\* current = root;

while (current->right\_child != nullptr)

{

current = current->right\_child;

}

return current->data;

}

}

T minimum() {

if (root == nullptr)

{

return NULL;

}

else {

node\* current = root;

while (current->left\_child != nullptr)

{

current = current->left\_child;

}

return current->data;

}

}

private:

node\* root;

bool insert\_helper(T data\_insert, node\* check\_node, node\* parent\_node)

{

if (check\_node == nullptr)

{

node\* new\_node = new node(data\_insert);

if (data\_insert > parent\_node->data)

{

parent\_node->right\_child = new\_node;

}

else

{

parent\_node->left\_child = new\_node;

}

new\_node->parent = parent\_node;

return true;

}

else if (data\_insert > check\_node->data)

{

return insert\_helper(data\_insert, check\_node->right\_child, check\_node);

}

else if (data\_insert < check\_node->data)

{

return insert\_helper(data\_insert, check\_node->left\_child, check\_node);

}

else

{

return false;

}

}

node\* find\_min\_branch(node\* branch)

{

while(branch->left\_child != nullptr)

{

branch = branch->left\_child;

}

return branch;

}

};

**Member functions:**

|  |  |
| --- | --- |
| node\* parent | Node pointer for the parent |
| node\* left\_child | Node pointer for the left child |
| node\* right\_child | Node pointer for right child |
| T data | Sets data to datatype T |
| node() | Node constructor |
| bst() | Bst contructor |
| ~bst() | Bst deconstructor |
| node\* get\_root() | Returns the root of the tree |
| bool insert(T data\_insert) | Insert a value into the tree |
| node\* find(T data\_find) | Find a specific data value in the tree |
| void remove(T data\_delete) | Deletes a specific data value in the tree |
| T maximum() | Finds the maximum value in the tree |
| T minimum() | Finds the minimum value in the tree |
| node\* root | The node pointer of root |
| bool insert\_helper() | Helper function for insert |
| node\* find\_min\_branch(node\* branch) | Finds the minimum branch(lowest left node) |

**Usage:**

Binary search trees are great if you’re trying to sort data that has a value that can be measured in a greater or less than manner. Due to its structure being searching and sorting friendly. If the value you were looking for is greater than the root then you go to the right, if it’s lower, then you go to the left. This makes it fast and efficient to search and sort data. But the main purpose of using a binary search tree is because it extends the capability of a normal array.