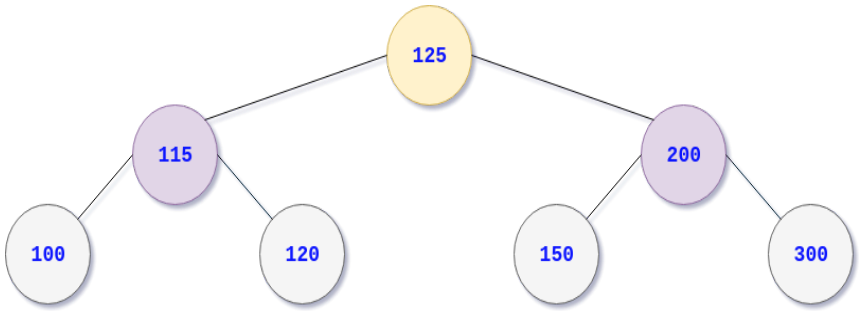
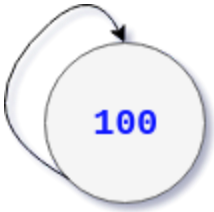
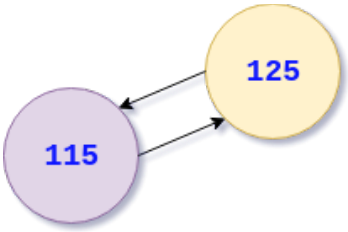
Binary Search Tree

A Binary Search Tree is a tree in which every node's data is always greater than or equal to the data in its left subtree and always less than the data in its right subtree. For example, if a node contains the data 125, all numbers less than or equal to 125 will be placed in its left subtree and all numbers greater than 125 will be placed in its right subtree.



With the aforementioned properties, a Binary Search Tree cannot have a node link to itself or a child node link back to the parent node. The following two images are **NOT** allowed in a Binary Search Tree:





To create a simplistic node, a `struct` is normally used in CSII (although you are able to use a `class`). Here is a simple node with the data being an integer:

```
struct Node
{
    int data;
    Node* leftChild = nullptr;
    Node* rightChild = nullptr;
};
```

In order to chain nodes together, each **Node*** will be dynamically allocated. Here they are initialized to `nullptr` as that is how to check whether or not a child is present. That is, a **Node*** will point to `nullptr` is no child is present and **Node*** will point to an address if a child is present.

To start, a separate **Node*** must be made to point to the root node, usually called **head**, **rootPtr**, or **nodePtr**. This should initially point to `nullptr`. The insertion process will firstly connect the pointer to the root node to the newly created root node. From there, the insertion process will connect the Binary Search Tree via the **leftChild** and **rightChild** node pointers.

The insertion, deletion, and display functions may utilize recursion. Since there are multiple ways of implementation, one method of these functions will be given. See **BinarySearchTree.cpp** for an example program reading and deleting nodes via a file that conform to the following descriptions.

Insertion

The Binary Search Tree will be empty at first. The insertion function will return a **Node*** and take a **Node*** by reference and an **int** as arguments.

```
Node* insert(Node* &nodePtr, int d)
```

The base case is when the **Node*** argument that is being passed in is pointing to **nullptr**. In this case, a new node is created:

```
if(nodePtr == nullptr) {
      Node* nodePtr = new Node;
      newNode->data = d; //d is the incoming data argument
      return nodePtr;
}
```

Note that you do not have to initialize the left or right child as they are already initialized to **nullptr** in the **struct**. Any new node will be a leaf.

After the root node has been created, any subsequent calls to the insert function will utilize recursion by comparing the incoming data with the nodes already in the Binary Search Tree. The first node to be compared will always be the root node. Depending on the value of the incoming data, the next node to be compared will either be the **leftChild** or **rightChild** if they exist. If a child is pointing to **nullptr**, that is where a new node will be created with the incoming data. Here are the two types of recursive calls:

1) If the incoming data is less than or equal to the current node, traverse the left subtree by setting the node's **leftChild** equal to the recursive call. For example:

```
if(d <= nodePtr->data)
    nodePtr->leftChild = insert(nodePtr->leftChild, d);
```

2) If the incoming data is greater than the current node, traverse the right subtree by setting the node's **rightChild** equal to the recursive call. For example:

```
if(d > nodePtr->data)
    nodePtr->rightChild = insert(nodePtr->rightChild, d);
```

For example, suppose the root node is **125**, its **leftChild** is **115** and the incoming data is **120**. **120** is compared with **125**. Since **120** is less than **125**, make a recursive call to the **leftChild**. **120** is now compared with **115**. Since **120** is greater than **115**, make a recursive call to the **rightChild**. The **rightChild** is pointing to **nullptr**. The base case is activated and a new node is created containing the data **120**. The address of this node is returned as the **rightChild** to the node containing **115**. The address of the node containing **115** is returned as the **leftChild** to the root node. Lastly, the address of the root node is returned to the root pointer. Since this function's **Node*** is passed by reference, the initial function invocation can be standalone.

Deletion

The deletion function will return a **Node*** and take a **Node*** by reference and an **int** as arguments.

```
Node* deleteNode(Node* &nodePtr, int d)
```

The deletion function has a similar base case as the insert function. If the incoming **Node*** is pointing to **nullptr**, The Binary Search Tree is either empty or the incoming data to be deleted does not exist. In that case, simply **return nodePtr;**

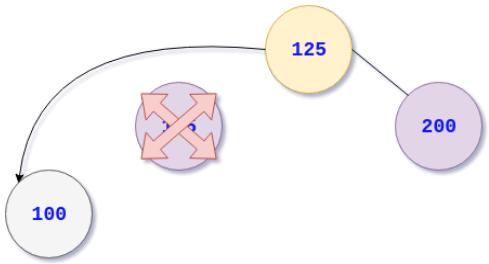
Utilize recursion if the data to be deleted is not equal to the data in the current node. Move to the **leftChild** if the data to be deleted is less than the current node. Move to the **rightChild** if the data to be deleted is greater than the current node.

```
if(d < nodePtr->data)
      nodePtr->left = deleteNode(nodePtr->left, d);
else if (d > nodePtr->data)
      nodePtr->right = deleteNode(nodePtr->right, d);
```

Now, if the data to be deleted is equal to the data in the current node, a few scenarios need to be checked.

1) If the node is a leaf node, simply delete the node and return **nullptr**.
2) If the node contains a **leftChild** only, create a temporary **Node*** and set it to the **leftChild**. Delete the current node and return the temporary **Node***. This will successfully delete the node and reconnect the Binary Search Tree by making the parent node point to the current node's (the now deleted node's) **leftChild**.
3) If the node contains a **rightChild** only, create a temporary **Node*** and set it to the **rightChild**. Delete the current node and return the temporary **Node***. This will successfully delete the node and reconnect the Binary Search Tree by making the parent node point to the current node's (the now deleted node's) **rightChild**.

Here is an example of deleting a node with only a **leftChild**. The same method is done when you delete a node with only a **rightChild**.



4) If the node contains a **leftChild** and **rightChild**, you either have to find the largest data in the left subtree or find the smallest data in the right subtree. This will preserve the properties of a Binary Search Tree. In this example, finding the lowest data in the right subtree is utilized.

Create a **Node*** to find the node that contains the smallest data. This will be found by setting the **Node*** to the current node's **rightChild** and looping through the left subtrees until you come across a node that has no **leftChild**.

```
Node* findSmallest = nodePtr->right;
while(findSmallest->left)
    findSmallest = findSmallest->left;
```

You will now have one pointer pointing to the node to be deleted and one pointer pointing to the node with the smallest data in its right subtree. Swap the data between the two nodes and delete the node **findSmallest** is pointing to. Since you swapped the nodes, the node to be deleted is now being pointed to by **findSmallest**. This is desirable, because the node can be deleted successfully by using the first or third method of deletion. Remember, there will never be a **leftChild**.

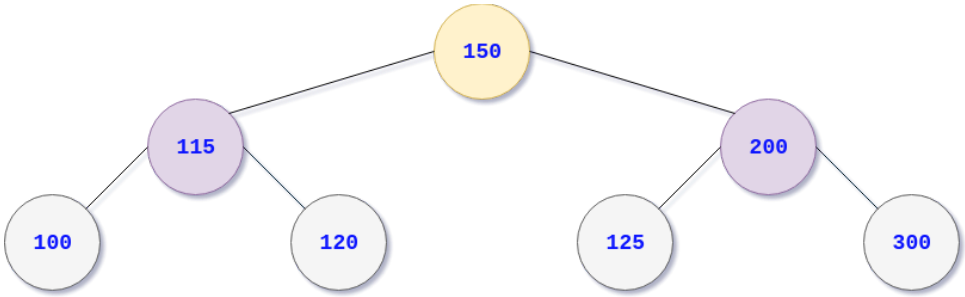
```
int temp = nodePtr->data;
nodePtr->data = findSmallest->data;
findSmallest->data = temp;
```

This means you will need to recursively call the delete function once more.

```
nodePtr->right = deleteNode(nodePtr->right, findSmallest->data);
```

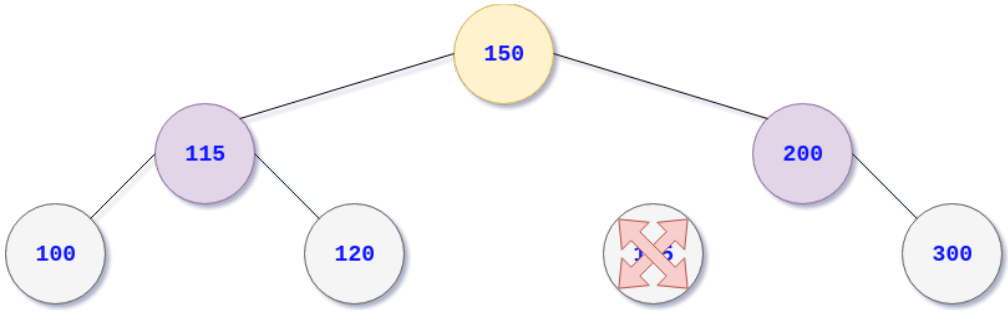
This recursive call focuses on the right subtree of **nodePtr** and will locate the new node to be deleted, which, again is the node pointed to by **findSmallest**. The following images on the next page shows this method of deleting a node with two children.

For example (using the Binary Search Tree from the first page), if you want to delete the root node containing the value 125, traverse its right subtree as far left as possible finding the node with the smallest value. In this case it is the node containing 150. Swap the values as such:



150 is now the root node and 125 is now the node where the smallest value in the right subtree (of the root node) was found. From there, 125 will be deleted using the first method by simply deleting the node since the node does not have a **rightChild**. In order to keep the Binary Search Tree intact, start the recursive call from the root node's right subtree to locate the data to be deleted (which is 125).

This recursive call will compare 125 with 200 and move on to the **leftChild** of the node containing 200. From there, 125 will be compared with 125 (content of that node) and trigger the deletion method as seen above. The node is successfully deleted and returns **nullptr** as the **leftChild** to the node containing 200. The address of the node containing 200 is then returned to the original function invocation in this step. The new Binary Search Tree will now look like this:



Display

Please refer to the **Tree Traversals** page to see recursive functions to display your Binary Search Tree. There are three common methods as fully described in that section.