Binary Search Trees

Written by Rachel J. Morris, last updated Apr 5, 2018

About

Let’s go over the functionality of common Binary Search Tree functions.

The Node

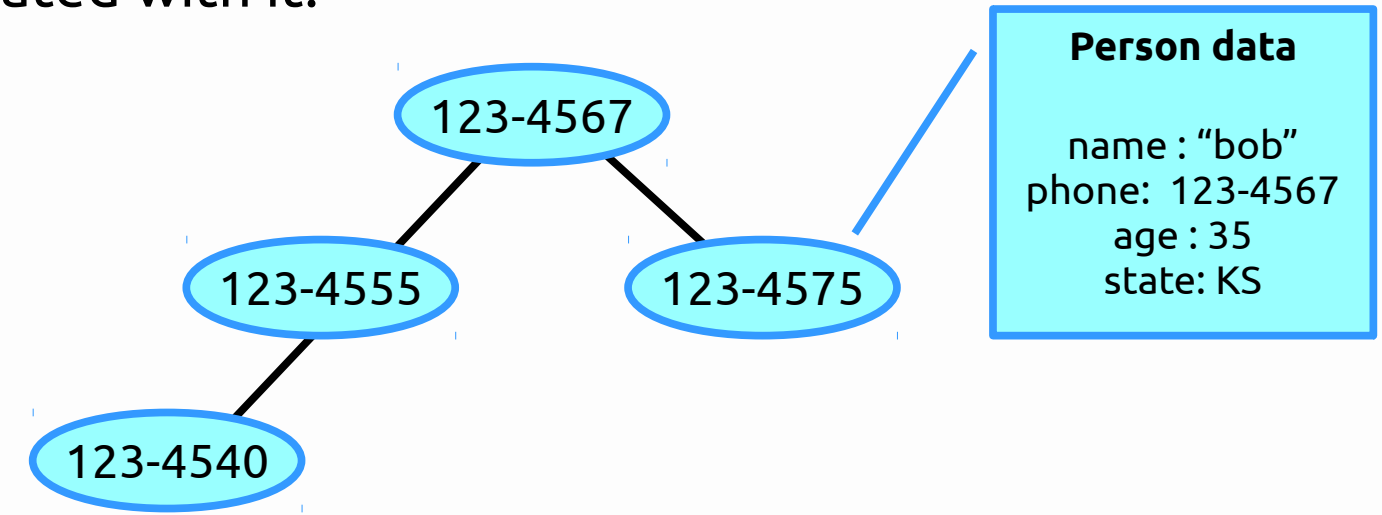
In our project, the Node contains:

`Node<TK,TD>* ptrLeft`	Pointer to the left child
`Node<TK,TD>* ptrRight`	Pointer to the right child
`TK key`	The **key** to access the node. In the tree, nodes will be sorted by key.
`TD data`	The **data** stored within the node. This is like the data in a linked list node.

Notes

The Node

The Binary Search Tree in Project 3 will organize the nodes by their keys; basically the look-up value. Then, if you access the node with that key, it will return the data associated with it.



In this example, the phone number is used as the “key” for a person. Within the node, Person data is stored, which contains more information.

Notes

The Node

For Project 3, we have the Node stored as a template with two template types: TK for the key, and TD for the data. That way we can store Nodes with different data types for the key and the value.

The key for project 3 will be a float – the price of a car.

The data for project 3 will be a CarData object, which contains a make, model, and price of a car.

Notes

The Binary Search Tree

At minimum, the Binary Search Tree needs a pointer to the root node.

There should also be functions to add, find, and remove nodes.

We will step through the functionality of each of these.

In the starter code, any recursive function has a public facing counterpart, which is responsible for calling the recursive version with the root node.

Notes

Push (public)

Scenario 1: If the root pointer is nullptr, there is nothing yet in the tree, so we can create new data and set it as the root.

New node is root!

Use the root pointer to create the new node,

```
m_ptrRoot = new Node<TK, TD>;
```

set is key and data,

```
m_ptrRoot->key = newKey;
m_ptrRoot->data = newData;
```

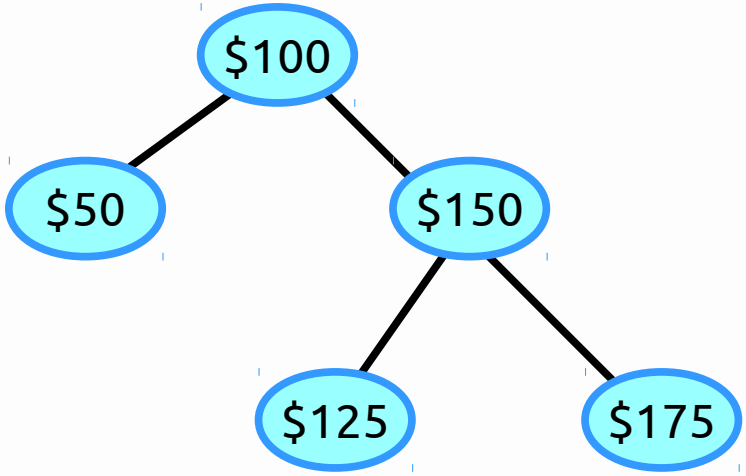
and increment the node count.

```
m_nodeCount++;
```

Notes

Push (public)

Scenario 2: The tree is NOT empty, so we need to find a place for the new data first. Call the RecursivePush function, passing in the key, the data, and the root pointer.

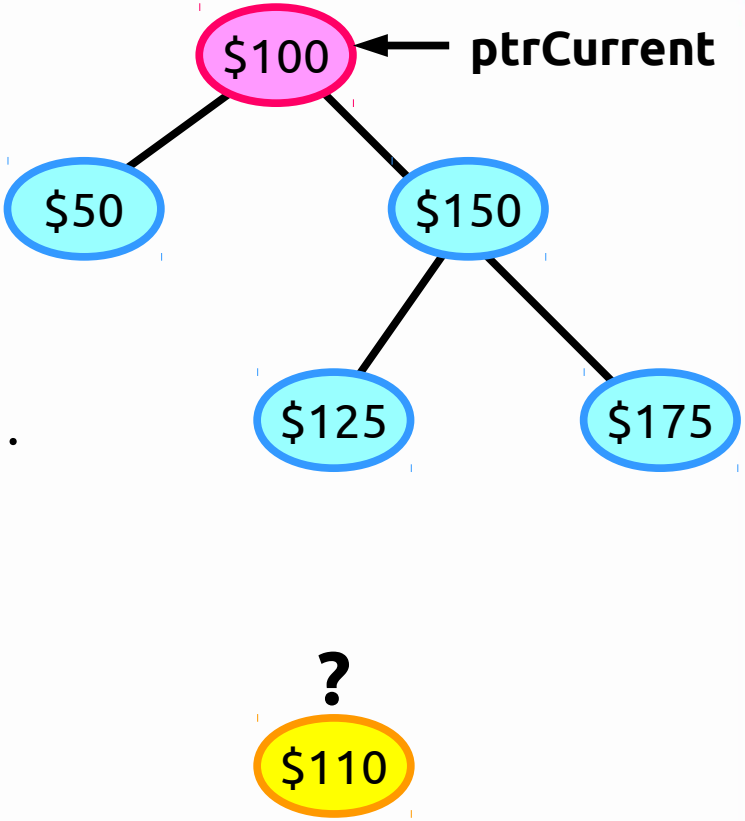



```
RecursivePush( newKey, newData, m_ptrRoot );
```

Notes

RecursivePush

For the recursive push, we are at some arbitrary node. We have the new Key and the new Data to pass in, so we have to find a location for it.



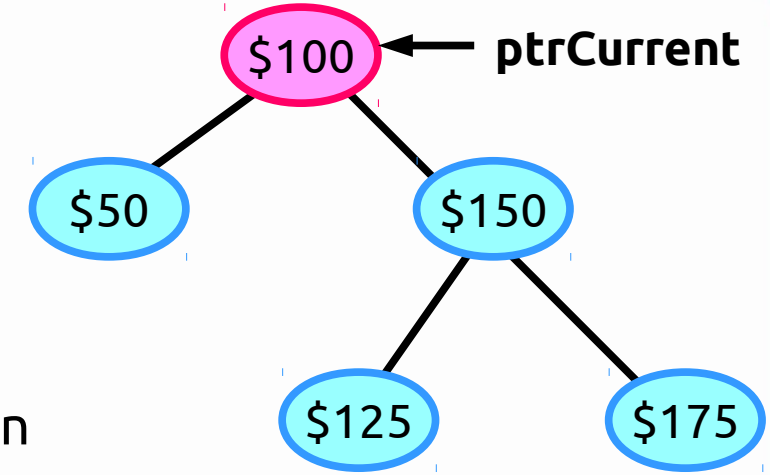
For the node we're currently on…

- Is our new item's key less than the current node?
  - No
- Is our new item's key greater than the current node?
  - **Yes**

Notes

RecursivePush

For the recursive push, we are at some arbitrary node. We have the new Key and the new Data to pass in, so we have to find a location for it.



Since our **newKey** is greater than **ptrCurrent->key**, we will look to the right.

- Is this position taken?
  - Yes
- OK, then what?
  - Recurse right.

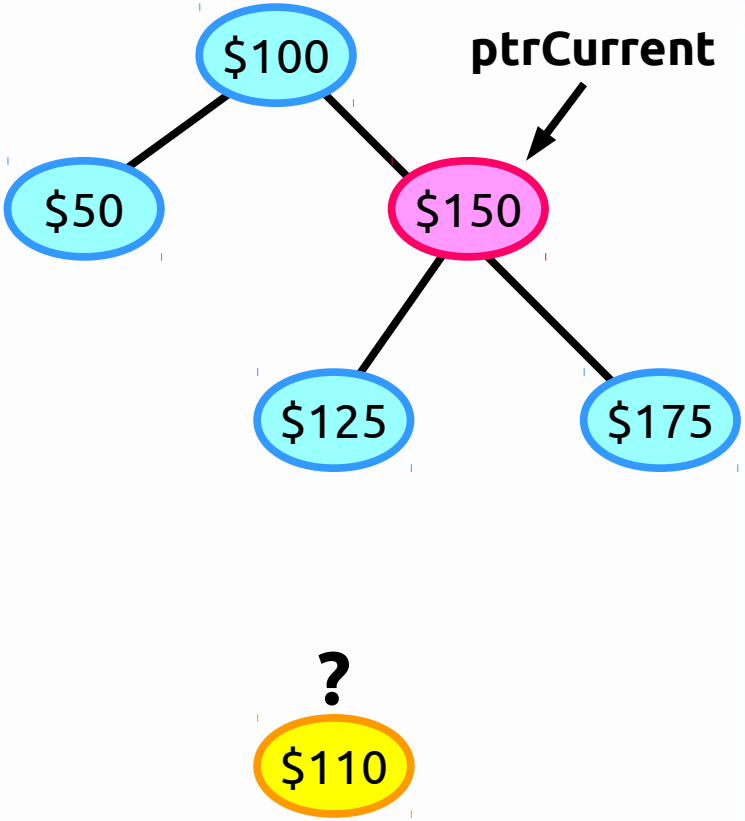

```
RecursivePush( newKey, newData, ptrCurrent->ptrRight );
```

Notes

RecursivePush

Now we have a new ptrCurrent.

- Is our new item's key less than the current node?
  - Yes
  - Is the position to the left taken?
    - Yes
    - Recurse left.



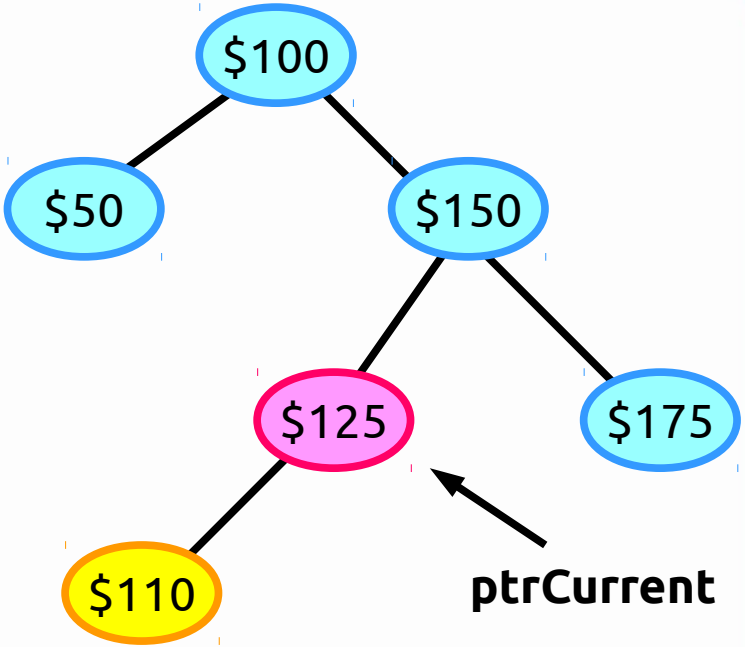

```
RecursivePush( newKey, newData, ptrCurrent->ptrLeft );
```

Notes

RecursivePush

Now we have a new ptrCurrent.

- Is our new item's key less than the current node?
  - Yes
  - Is the position to the left taken?
    - No
    - Insert the new node as **ptrCurrent's** left child!



Notes

RecursivePush

Is the **newKey** less than **ptrCurrent's key**?

- Yes:
  - Is there a node at the **left** child position?
    - Yes:
      - Recurse **left** to keep looking for a position.
    - No:
      - Place the new node as **ptrCurrent's** left child.
- No:
  - Is there a node at the **right** child position?
    - Yes:
      - Recurse **right** to keep looking for a position.
    - No:
      - Place the new node as **ptrCurrent's** right child.

Notes

Find Node

There are several functions that will traverse the tree, such as **FindNode**, **GetData**, and **FindParentOfNode**, these will mostly work the same way, with some small differences.

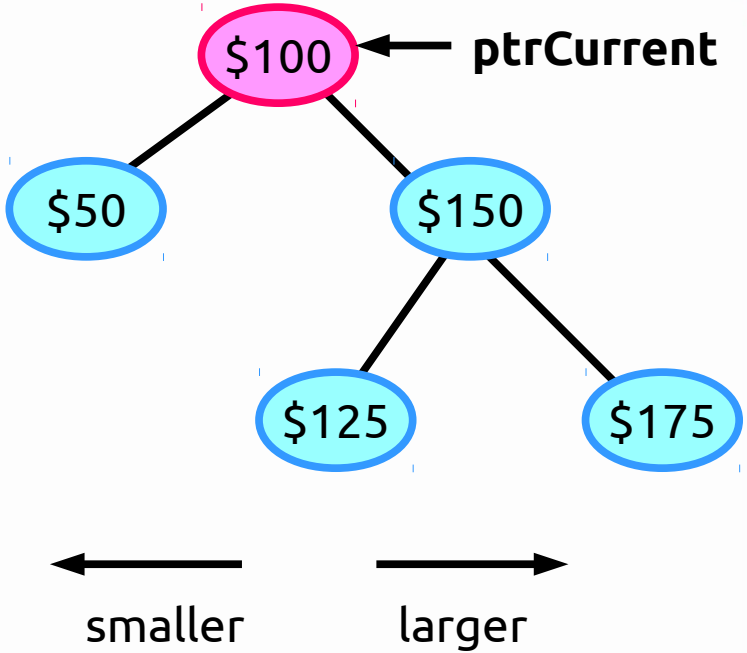
Notes

Find Node

We will have a **key** that we're searching for, and we begin iterating with **ptrCurrent** as the root.

For each **ptrCurrent**,

- If **ptrCurrent = nullptr,** we couldn't find the node, return false or nullptr.
- If **key = ptrCurrent→key,** return **ptrCurrent's** data.
- If **key < ptrCurrent→key,** traverse left
- If **key > ptrCurrent→key,** traverse right



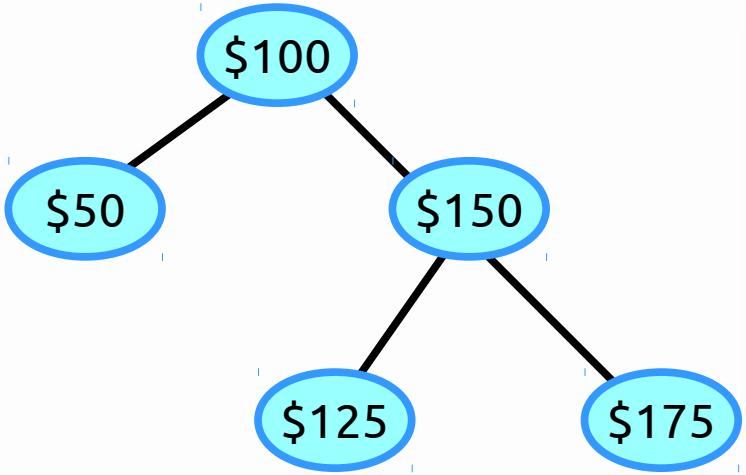
Notes

Traversals

PreOrder traversal:
- Display **ptrCurrent→key**
- Recurse left
- Recurse right

InOrder traversal:
- Recurse left
- Display **ptrCurrent→key**
- Recurse right

PostOrder traversal:
- Recurse left
- Recurse right
- Display **ptrCurrent→key**



Notes

GetMinKey / GetMaxKey

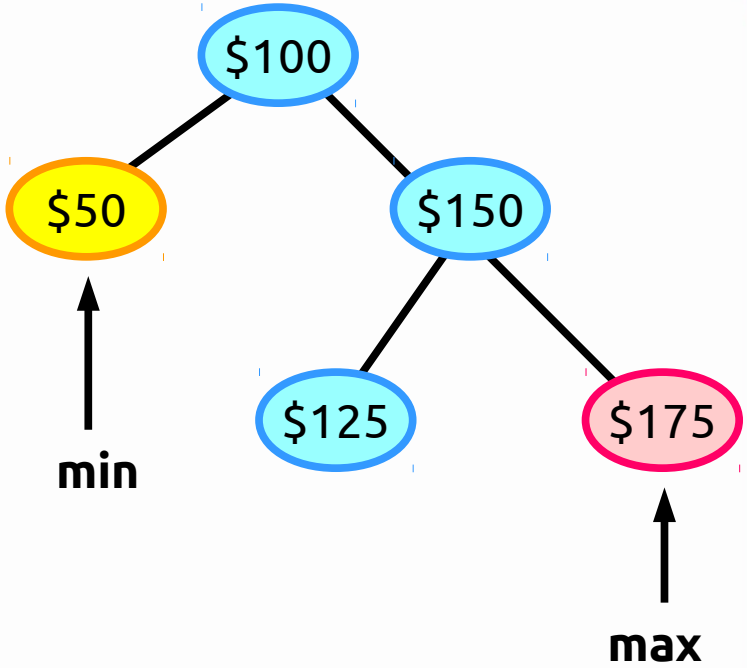
For the GetMinKey function, the smallest key will be all the way to the left. So basically…

ptrCurrent begins at the root.

- If **ptrCurrent→left** is not nullptr
  - Recurse left

- If **ptrCurrent→left** is nullptr
  - Return **ptrCurrent**'s key.

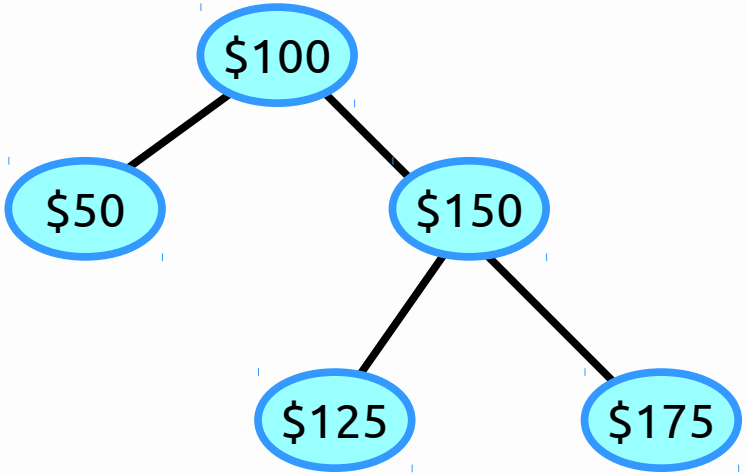
And vice versa for max – go all the way right.



Notes

GetHeight

The GetHeight function should return the height of the longest path from the root to a leaf.



- Terminating case: If **ptrCurrent** is nullptr, return a length of -1.

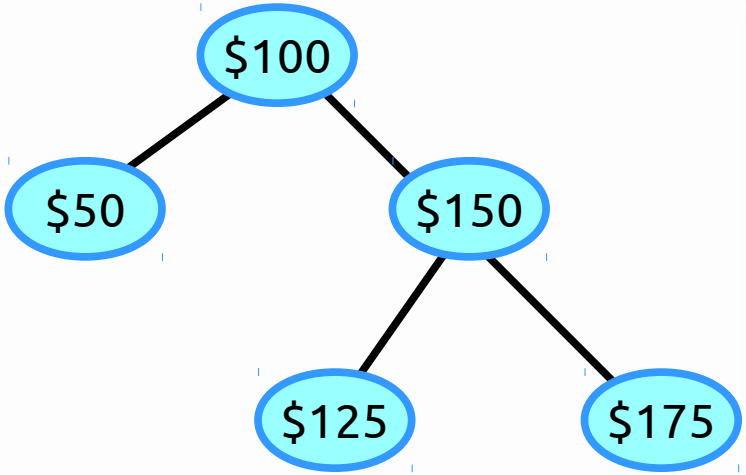
- Create **leftHeight** and **rightHeight** variables, set to 0.
- If the left child is not null, recurse left and set the return value to the **leftHeight** variable.
- If the right child is not null, recurse right and set the return value to the **rightHeight** variable.

- Return 1 + the larger height value (+1 to count self).

Notes

Delete

The Delete function is the trickiest. I've written the base part of the Delete function for you already, but you'll have to write the logic for deleting a node based on its children.



Notes

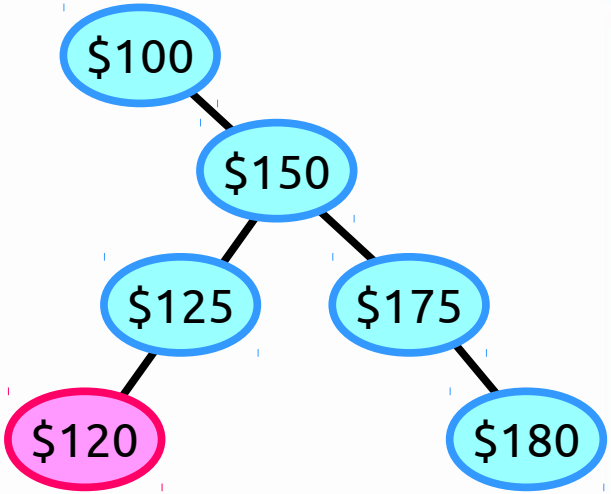
Delete

```
template <typename TK, typename TD>
void BinarySearchTree<TK,TD>::Delete( const TK& key )
{
    // Locate node
    Node<TK, TD>* deleteMe = FindNode( key );

    if ( deleteMe == nullptr )
    {
        cout << "deleteMe is nullptr" << endl << endl;
        return;
    }

    Node<TK, TD>* parent = FindParentOfNode( key );
    bool isLeftNode = ( parent->ptrLeft == deleteMe );

    if ( deleteMe->ptrLeft == nullptr && deleteMe->ptrRight == nullptr )
        DeleteNode_NoChildren( deleteMe, parent, isLeftNode );
    else if ( deleteMe->ptrLeft == nullptr )
        DeleteNode_RightChild( deleteMe, parent, isLeftNode );
    else if ( deleteMe->ptrRight == nullptr )
        DeleteNode_LeftChild( deleteMe, parent, isLeftNode );
    else
        DeleteNode_TwoChildren( deleteMe, parent, isLeftNode );
}
```

Notes

Delete – No children

deleteMe is a node with no children:



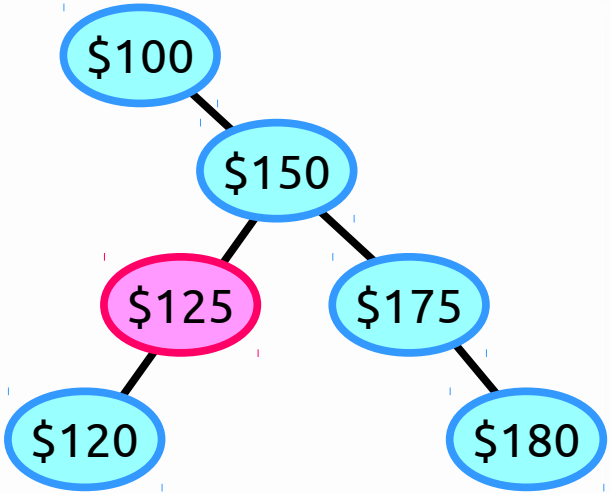
- If **deleteMe** is the root:
  - Set the root pointer to nullptr.
- Else if **deleteMe** is the left child of its parent:
  - Set the parent's left pointer to nullptr.
- Else if **deleteMe** is the right child of its parent:
  - Set the parent's right pointer to nullptr

- delete **deleteMe**, and reduce the node count.

Notes

Delete – Left child only

deleteMe is a node with only the left child:



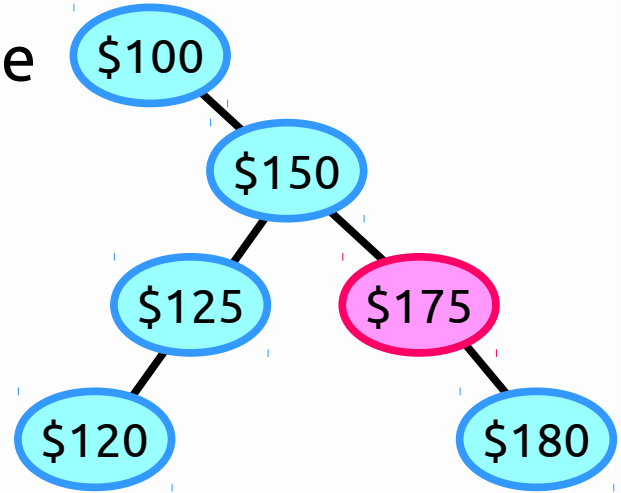
- If **deleteMe** is the root:
  - Set the root pointer to deleteMe's left child.
- Else if **deleteMe** is the left child of its parent:
  - Set the parent's left pointer to deleteMe's left child.
- Else if **deleteMe** is the right child of its parent:
  - Set the parent's right pointer to deleteMe's left child.

- Set deleteMe's left child to nullptr.
- delete **deleteMe**
- reduce the node count.

Notes

Delete – Right child only

deleteMe is a node with only the right child:



- If **deleteMe** is the root:
  - Set the root pointer to deleteMe's right child.
- Else if **deleteMe** is the left child of its parent:
  - Set the parent's left pointer to deleteMe's right child.
- Else if **deleteMe** is the right child of its parent:
  - Set the parent's right pointer to deleteMe's right child.

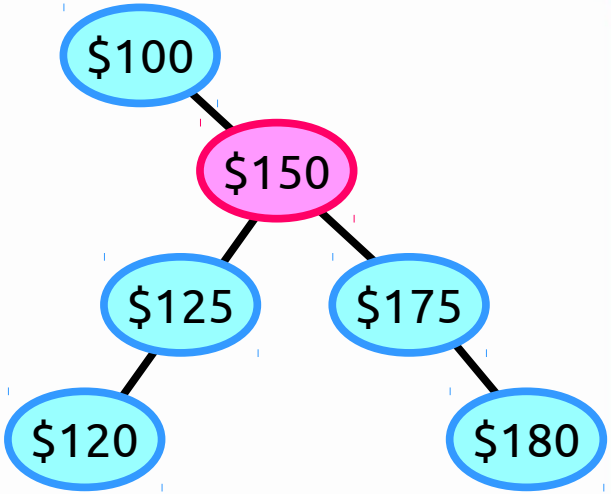
- Set deleteMe's right child to nullptr.
- delete **deleteMe**
- reduce the node count.

Notes

Delete – Left AND right child

deleteMe has both left and right children, we need to figure out what to do with both children.

1. We need three nodes to figure out how far to traverse:

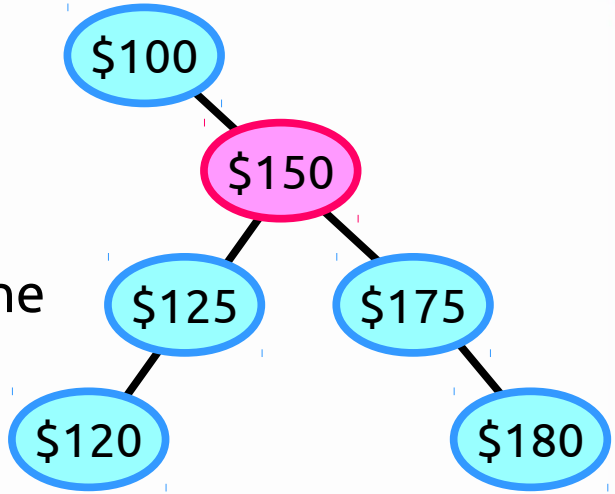



```
Node<TK, TD>* tempNode = deleteMe->ptrRight;
Node<TK, TD>* successor = deleteMe;
Node<TK, TD>* successorParent = deleteMe;
```

Notes

Delete – Left AND right child

deleteMe has both left and right children, we need to figure out what to do with both children.



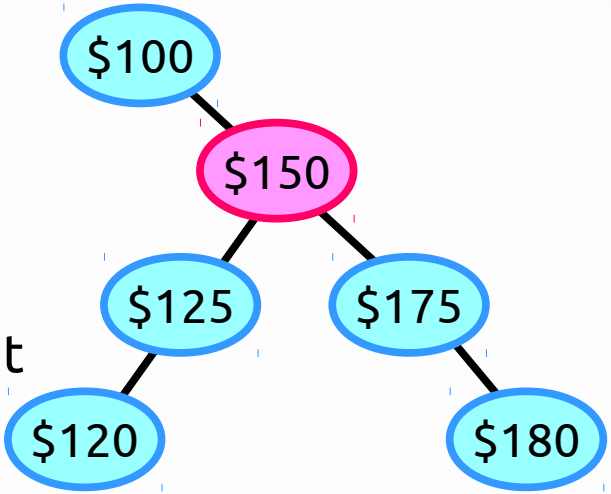
2. Keep traversing left until we get to the end of the tree.

```
while ( tempNode != nullptr )
{
    successorParent = successor;
    successor = tempNode;
    tempNode = tempNode->ptrLeft;
}
```

Notes

Delete – Left AND right child

deleteMe has both left and right children, we need to figure out what to do with both children.



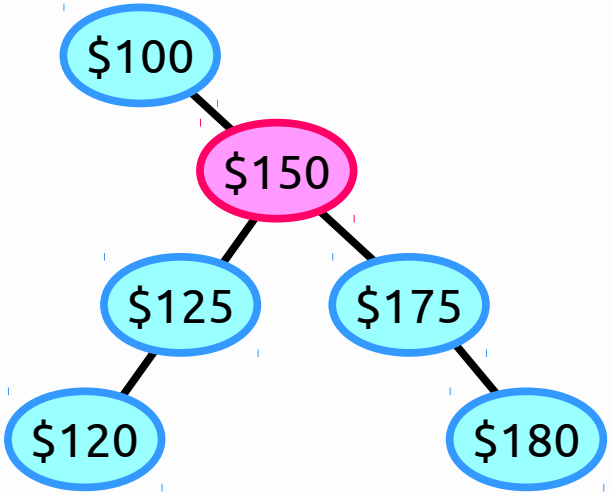
3.

- If the successor is <u>not</u> deleteMe's right child, then…
  - Set the successorParent's left child to the successor's right child.
  - Set the successor's right child to deleteMe's right child.

Notes

Delete – Left AND right child

deleteMe has both left and right children, we need to figure out what to do with both children.



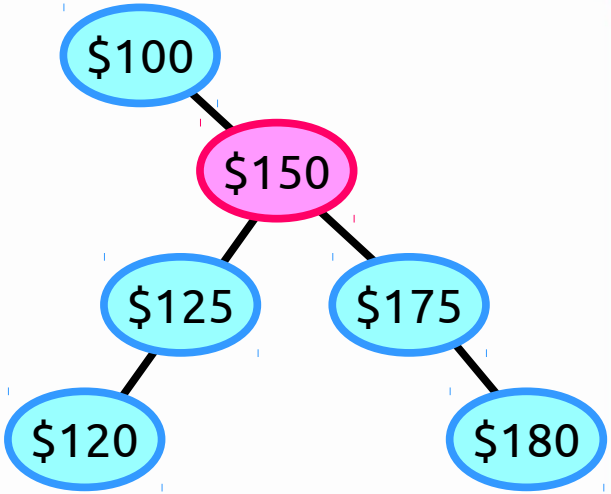
4.

- If deleteMe is the root…
  - Set the root to the successor
- Otherwise, if deleteMe is the left child of its parent…
  - Set deleteMe to the parent's left ptr
  - Set the parent's left ptr to the successor.
- Otherwise…
  - Set deleteMe to the parent's right ptr
  - Set the parent's right ptr to the successor.

Notes

Delete – Left AND right child

deleteMe has both left and right children, we need to figure out what to do with both children.



5. Set the successor's left child to deleteMe's left child.

Then we can delete:

- Set deleteMe's left ptr to nullptr.
- Set deleteMe's right ptr to nullptr.
- Delete deleteMe
- Decrease the node count

Notes

Conclusion

Make sure to also view the Doxygen documentation for instructions on how the functions should work.

```
#include <BinarySearchTree.hpp>
```

Public Member Functions

	BinarySearchTree () Initializes the node count to 0, and the root pointer to nullptr. More...
	~BinarySearchTree () Destroys the root node. More...
void	**Push** (const TK &newKey, const TD &newData) Creates a new node for the tree and assigns the data of that node to the parameter passed in. More...
void	**Delete** (const TK &key) Deletes the **Node** that contains the given data, if it exists in the tree. More...