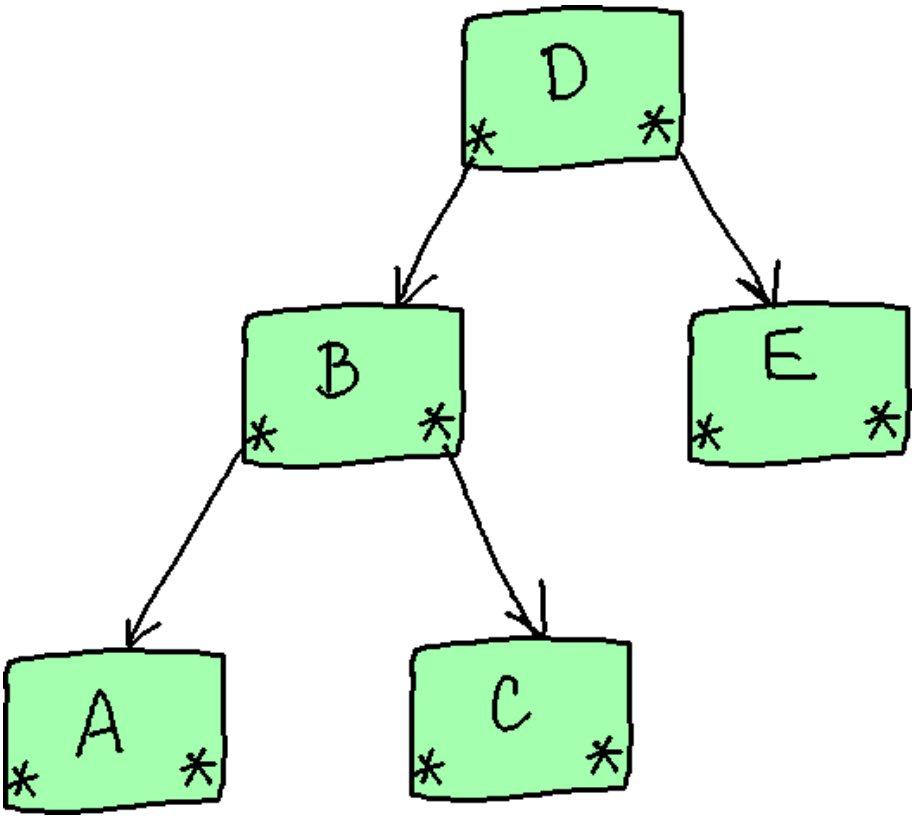
Rachel's Data Structures Notes
Binary Search Trees



An overview compiled by Rachel Singh

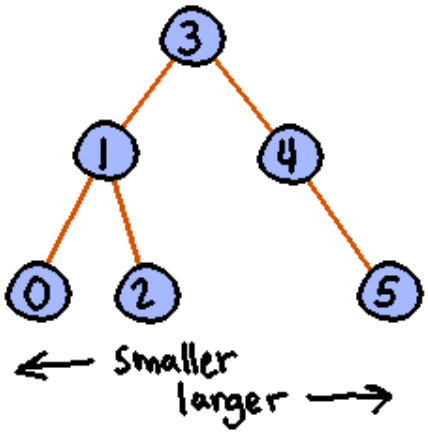
Last updated November 8, 2020

Contents

Topic 1

Binary Search Trees

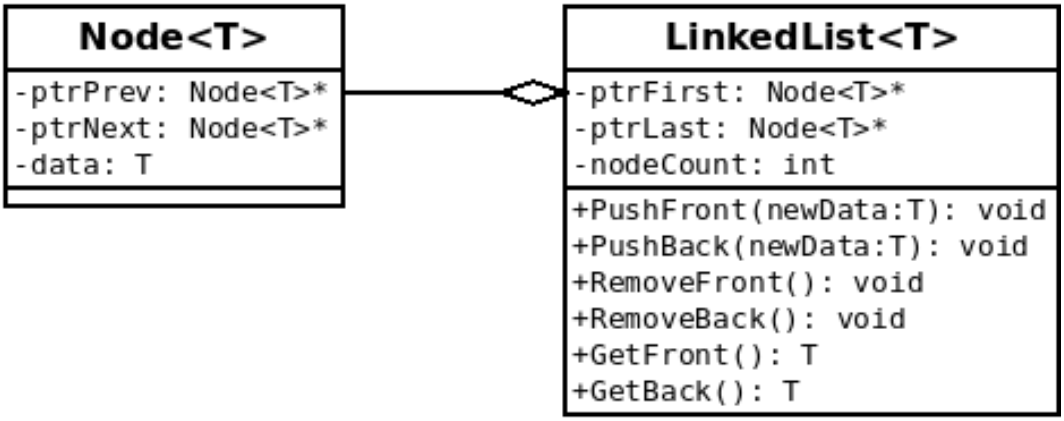
1.1 Introduction to Binary Search Trees



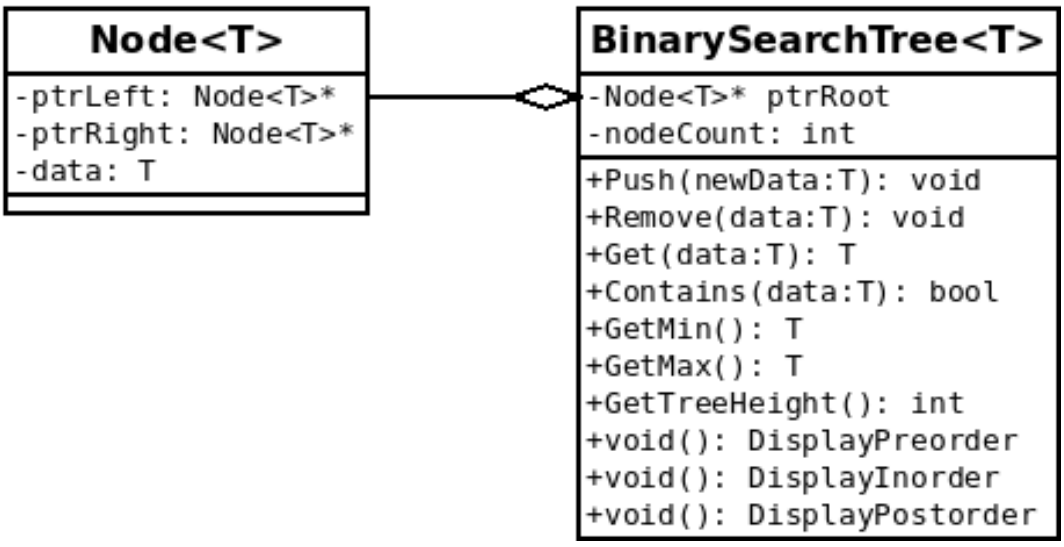
Binary search trees are a type of data structure that keep the data it contains **ordered**. The ordering process happens when a new piece of data is entered by finding a location for the data that adheres to the ordering rules. With a binary search tree, **smaller values** are stored to the left and **larger values** are stored to the right.

This means that when we're searching for data, when we land at a node we can figure out whether to traverse *left* or *right* by comparing the node to what we're searching for.

1.2 Architecture of a Binary Search Tree



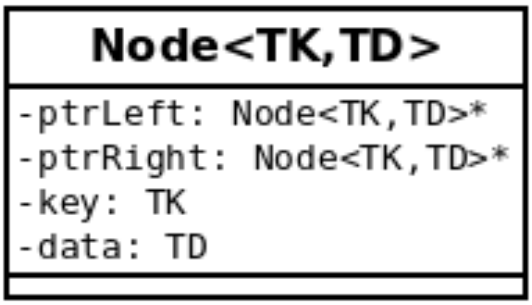
With a **Linked List**, we need to implement a Node structure and a LinkedList class, where the Node stores the data and the LinkedList provides an interface for users to add, remove, and search for data and stores a pointer to the **first** and **last** elements.



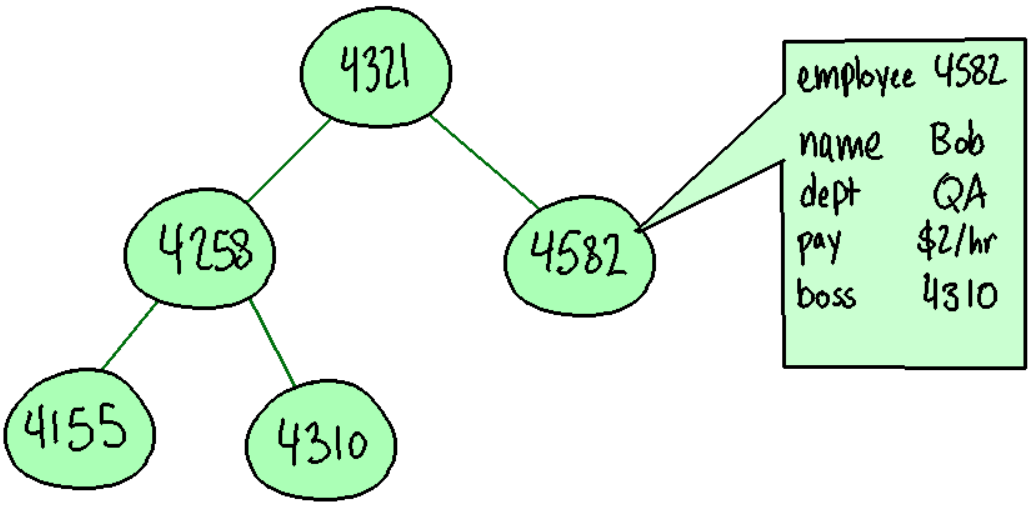
Similarly for a **Binary Search Tree**, we need another type of Node to store the data, as well as the BinarySearchTree structure that acts as an interface and keeps a pointer to the **root node**.

We might also think of a Binary Search Tree as being ordered based on the data's **key** - some sort of unique identifier we assign to each node - and then containing additional data (a value) within the node.

For example, if we create our Node with two templated types like this:



our **key** could be a unique lookup (e.g., "employee ID"), and the **data**/value could be another structure that stores more employee data (name, department, etc.)



What's the significance of the hierarchy in this tree? Nothing, really - the point of a binary search tree is that we're assuming the **keys** we will be pushing into the tree will be in a somewhat **random order**, and using the BST structure will help keep things ordered and somewhat faster to search through.

1.2.1 BinarySearchTreeNode in C++:

```
template <typename TK, typename TD>
class Node
{
public:
    Node()
    {
        ptrLeft = nullptr;
        ptrRight = nullptr;
    }

    Node( TK newKey, TD newData )
    {
        key = newKey;
        data = newData;
        ptrLeft = nullptr;
        ptrRight = nullptr;
    }

    ~Node()
    {
        if ( ptrLeft != nullptr  ) { delete ptrLeft; }
        if ( ptrRight != nullptr ) { delete ptrRight; }
    }

    Node<TK, TD>* ptrLeft;
    Node<TK, TD>* ptrRight;

    TD data;
    TK key;
};
```

The node I've written here contains a **key**, which nodes will be ordered by, and **data**, which can contain more information.

As with any structure utilizing **pointers**, the pointers should be initialized to `nullptr` in any constructors.

The **destructor** here will trigger the deletion of any child nodes, creating a chain reaction to clean up the entire tree if the root node is deleted.

1.2.2 BinarySearchTree in C++:

```
1  template <typename TK, typename TD>
2  class BinarySearchTree
3  {
4  public:
5      BinarySearchTree();
6      ~BinarySearchTree();
7
8      // Basic functionality
9      void Push( const TK& newKey, const TD& newData );
10     bool Contains( const TK& key );
11     TD& GetData( const TK& key );
12     void Delete( const TK& key );
13
14     // Traversal functions
15     string GetInOrder();
16     string GetPreOrder();
17     string GetPostOrder();
18
19     // Additional functionality
20     TK& GetMinKey();
21     TK& GetMaxKey();
22     int GetCount();
23     int GetHeight();
24
25 private:
26     // (more here)
27
28 private:
29     Node<TK, TD>* m_ptrRoot;
30     int m_nodeCount;
31 };
```

A Binary Search Tree, just like other data structures, can store more functionality than this, or less if needed.

There are additional **private methods** that would be implemented. This declaration is just showing the **public (interface) methods** and the **private member variables**. I will talk about the private helper methods in depth later.

1.3 Efficiency of a Binary Search Tree

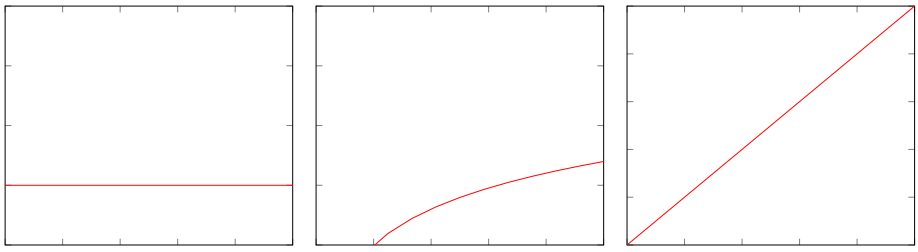
The Binary Search Tree ends up being a good compromise between choosing **faster random access but slow search/insert/delete** (like with a dynamic array) and **faster inserts/deletes but slow search/access** (like with a linked list).

The Binary Search Tree ends up being slower than $O(1)$ (instant) but faster than $O(n)$ (linear) for all of its operations:

Structure	**Random access**	**Search**	**Insert**	**Delete**
Dynamic Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Binary Search Tree	$O(log(n))$	$O(log(n))$	$O(log(n))$	$O(log(n))$

Why is this? By the nature of its tree structure, as we traverse the tree we're essentially **cutting out half the tree** each time we choose to go *left* or *right*. Halfing the nodes to search *each cycle* means we have the opposite of exponential growth: A logarithmic function.

Constant growth $O(1)$ | Logarithmic growth $O(log(n))$ | Linear growth $O(n)$



1.4 The Binary Search Tree and Recursion

Many of the BinarySearchTree functions will be **recursive**, starting at the root node and recursing down. Because of this, the Push function (and many others) that would actually do the work would look like this:

```
1 void RecursivePush(
2     TK newKey,
3     TD newData,
4     Node<TK, TD>* ptrCurrent );
```

However, we don't want the user *outside of the BinarySearchTree* to have to call Push and pass in the tree's node. They shouldn't even have access to any `Node` objects...

```
1 myTree.Push( 'a', "apple", ???? ); // What do I pass in?
```

That's why we have the **public Push function**...

```
1 void Push( TK newKey, TD newData );
```

And a **private RecursivePush function**...

```
1 void RecursivePush( TK newKey, TD newData,
2     Node<TK, TD>* ptrCurrent );
```

Where the user calls the **public Push** and that function makes the first call to **RecursivePush**, passing in the root node to begin operations on.

```
1 void Push( TK newKey, TD newData )
2 {
3     RecursivePush( newKey, newData, m_ptrRoot );
4 }
```

1.4.1 Private recursive functions

- `void Push( TK newKey, TD newData )` calls
  `RecursivePush( newKey, newData, m_ptrRoot );`
- `bool Contains( TK key )` calls
  `return RecursiveContains( key, m_ptrRoot );`
- `string GetPreOrder()` calls
  `return RecursiveGetPreOrder( m_ptrRoot );`
- `string GetInOrder()` calls
  `return RecursiveGetInOrder( m_ptrRoot );`
- `string GetPostOrder()` calls
  `return RecursiveGetPostOrder( m_ptrRoot );`
- `Node<TK,TD>* FindNode( TK key )` calls
  `return RecursiveFindNode( key, m_ptrRoot );`
- `TK GetMaxKey()` calls
  `return RecursiveGetMaxKey( m_ptrRoot );`
- `TK GetMinKey()` calls
  `return RecursiveGetMinKey( m_ptrRoot );`
- `int GetHeight()` calls
  `return RecursiveGetHeight( m_ptrRoot );`

1.5 Functionality of a Binary Search Tree

1.5.1 The Constructor

The constructor of the BinarySearchTree should set the `m_ptrRoot` pointer to `nullptr` and initialize the `m_nodeCount` to 0.

1.5.2 The Destructor

The destructor of the BinarySearchTree will check to see if `m_ptrRoot` is not null - if it's not null, we will free that memory with the `delete` command.

1.5.3 Push

Within **Push**, first check to see if the tree already contains a node with the given `newKey` by calling the `Contains` method. If that key is already present, I would throw an exception (for this design, we are assuming the keys are unique identifiers).

If the key is *not* already in the tree, then we are concerned with two scenarios:

1. The `m_ptrRoot` is nullptr.
2. The `m_ptrRoot` *is not nullptr.*

If the root is null, this is where we put our new node and set up its data:

```
1 m_ptrRoot = new Node<TK,TD>( newKey, newData );
2 m_nodeCount++;
```

If the root is already storing some data, then we call `RecursivePush`, passing forward the `newKey`, `newData`, and the `m_ptrRoot` as the starting point.

RecursivePush

Within the RecursivePush function, we need to be concerned with several scenarios:

1. The `newKey` is **less than** the `ptrCurrent->key`,
   and `ptrCurrent->ptrLeft` **is** `nullptr`:
   Store the new data here.

2. The `newKey` is **less than** the `ptrCurrent->key`,
   and `ptrCurrent->ptrLeft` **IS NOT** `nullptr`:
   Recurse left.

3. The `newKey` is **greater than** the `ptrCurrent->key`,
   and `ptrCurrent->ptrRight` **is** `nullptr`:
   Store the new data here.

4. The `newKey` is **greater than** the `ptrCurrent->key`,
   and `ptrCurrent->ptrRight` **IS NOT** `nullptr`:
   Recurse right.

Setting up the new node is a matter of just allocating space and incrementing the `m_nodeCount`:

```
1 // Storing data to the left of the current pointer
2 ptrCurrent->ptrLeft = new Node<TK,TD>(newKey, newData);
3 m_nodeCount++;
```

And recursing just requires passing through the same key and data, but a new node to look at:

```
1 // Recurse left
2 RecursivePush( newKey, newData, ptrCurrent->ptrLeft );
```

1.5.4 GetNodeWithKey

1.5.5 GetMinKey

1.5.6 GetMaxKey

1.5.7 GetHeight