Linked List Runtime Analysis

- Addperson() function has a runtime of O(1) as the worst case because a person gets added to the head of the newNode

```
void LinkedList::Addperson(int id, string name, string dob, string a, string a2, string a3, int a4){//Add function
Node * newNode = new Node(id, name, dob, a, a2, a3, a4);//new data allocated
newNode->next = Head;
Head = newNode;
cout << "Thank you for your information " << newNode->name<< " has been added to the list"<<endl;
}
```

- DeletePerson() has a runtime of O(n) because it will have to search through every id to ensure they match and delete the person

```
void LinkedList::DeletePerson(int id){//delete function requires id input
    Node * curr = Head;
    Node* prev = nullptr;

    while(curr != nullptr){

        if (curr->ID == id){
            if(prev != nullptr){
                prev->next = curr->next;// redirecting pointers to ensure no data is lost or misplaced
            }
                else {
                    Head = curr->next;
                }
                delete curr;
            cout << "The person with id: "<< id << " has been deleted from the List"<< endl;
            return;
        }
        prev = curr;
        curr = curr->next;
    }
    cout << "Person with this id is not found in the list"<<endl;
}
```

- All the following function utilize O(n) as the worst runtime because of the while loop they utilize.

```
Node* LinkedList::SearchPersonN(string n){// function that returns a Node Pointer after searching for specific name
    Node * curr = Head;
    while (curr != nullptr){
        if(curr->name == n){
            return curr;
        }
        else {
            curr = curr->next;

        }
    }
    return nullptr; // not found in list

}
```

```
void LinkedList::UpdateInfo(int id){// Function to update information while prompting user
   Node* UpdatedNode = SearchPerson(id);
   cout << "Current information: "<< endl;
   cout << "Name: " << UpdatedNode->name<<endl;
   cout << "ID: " << UpdatedNode->ID<<endl;
   cout << "Date of Birth: " << UpdatedNode->DOB<<endl;
   cout << "Address: " << UpdatedNode->address<<endl;
       if (UpdatedNode != nullptr) {
           string n, dob, a,a2,a3;
           int a4;
           cout << "Please enter new Name: ";
           cin >> n;
           cout << "Please enter new DOB: ";
           cin >> dob;
           cout << "Please enter new Address(Street): ";
           cin >> a;
           cout << endl;
           cout << "Please enter new Address(City): ";
           cin >> a2;
           cout << endl;
           cout << "Please enter new Address(State): ";
           cin >> a3;
           cout << endl;
           cout << "Please enter new Address(Zip Code): ";
           cin >> a4;
           cout << endl;
```

```
void LinkedList::DisplayEveryone(){//display function
    Node* curr = Head;
    while(curr != nullptr){
        cout << "Name: "<< curr->name<<endl;
        cout << "ID: "<< curr->ID<<endl;
        cout << "Date of birth: "<< curr->DOB<< endl;
        cout << "Address(Street): "<<curr->address << endl;
        cout << "Address(City): "<<curr->a2 << endl;
        cout << "Address(State): "<<curr->a3 << endl;
        cout << "Address(Zip Code): "<<curr->a4 << endl;

        curr= curr->next;
    }
}
```

Binary Search Tree Runtime Analysis

- Regarding the BST functions of Search() and Insert() will have a worst runtime of O(n) but depending on the size of the Binary Search Tree it will have an average runtime of O(log(n))

```
class BinarySearchTree {
private:
    TreeNode* root;

    // insert function
    TreeNode* insertRec(TreeNode* root, const NodeData& data) {
        if (root == nullptr) {
            return new TreeNode(data);
        }

        if (data.id < root->data.id) {
            root->left = insertRec(root->left, data);
        }
        else if (data.id > root->data.id) {
            root->right = insertRec(root->right, data);
        }

        return root;
    }

   // search funtion for integers
   bool searchRecInt(TreeNode* root, int id) {
       if (root == nullptr) {
           return false;
       }

       if (root->data.id == id) {
           return true;
       }
       else if (id < root->data.id) {
           return searchRecInt(root->left, id);
       }
       else {
           return searchRecInt(root->right, id);
       }
   }
```

- The following Delete() function will have an worst-case runtime of O(n) because it cycles down the tree to find the node

```
// function for deletion for ints and strings
TreeNode* deleteRecByName(TreeNode* root, const string& name) {
    if (root == nullptr) {
        return root;
    }
```