Dan Le

CMPSC 24

PA02 Report

Part 1: Searching

Code for search()

```
// search returns true if title is in BST, false otherwise
bool MovieList::search(std::string title){
    return search(title, root);
}

// recursive helper for search
bool MovieList::search(std::string title, Movie* n){
    if(n){
        if(title == n->getTitle()){ // return true if n's title is the same as target title
            return true;
        }
        if(title.compare(n->getTitle()) < 0){  // go to n's left subtree if title is less than n's title
            return search(title, n->left);
        }
        else{
            return search(title, n->right);  // go to n's right subtree otherwise
        }
    }
    return false;
}
```

Dataset	Number of runs (W)	Minimum Time (micro seconds)	Maximum time (micro seconds)	Average (micro seconds)
20 Ordered	50	54	91	62.24
20 Random	50	39	100	45.10
100 Ordered	50	3989	4988	4647.56
100 Random	50	963	1996	1157.66
1000 Ordered	50	424384	451794	428647.62
1000 Random	50	14959	15481	14353.32

Analysis of Searching in a BST

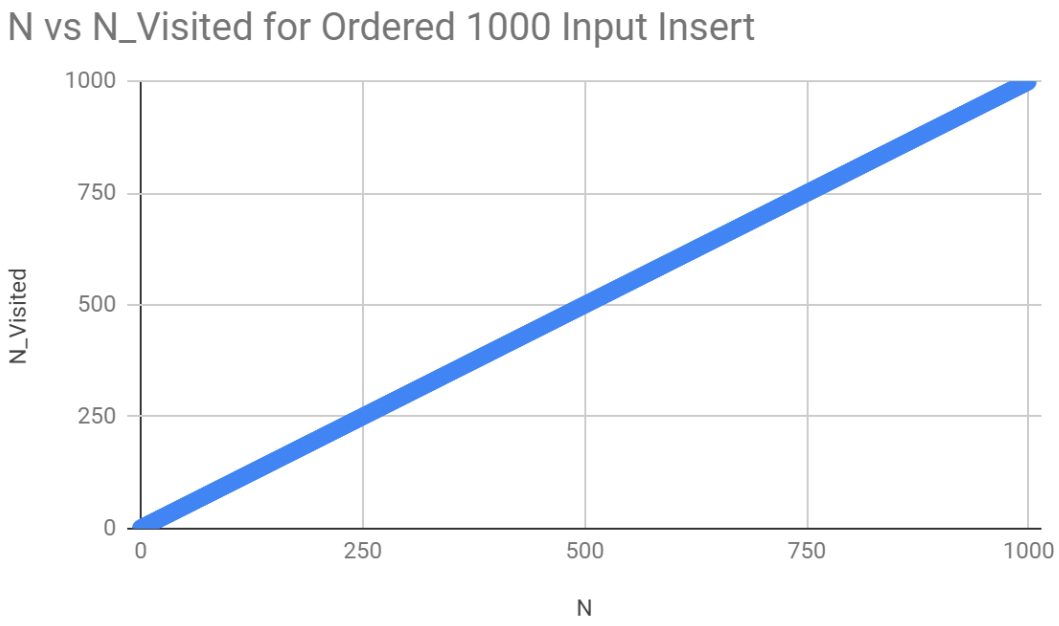
The ordered datasets take a longer time to search than the random datasets, especially for 1000 inputs. The 1000 ordered took around 100 times longer to run than the 100 ordered, while the 1000 random only took around 10 times longer to run than the 100 random.

I expected that the ordered datasets would take longer to search for a specific movie. The worst case scenario for the ordered datasets is when the movie being searched for is the most recent insertion into the binary search tree. Every node would need to be visited, making search() very inefficient in this case. For smaller binary search trees, this would not be much of an issue. However, for big datasets, this would not be a very efficient way to search.

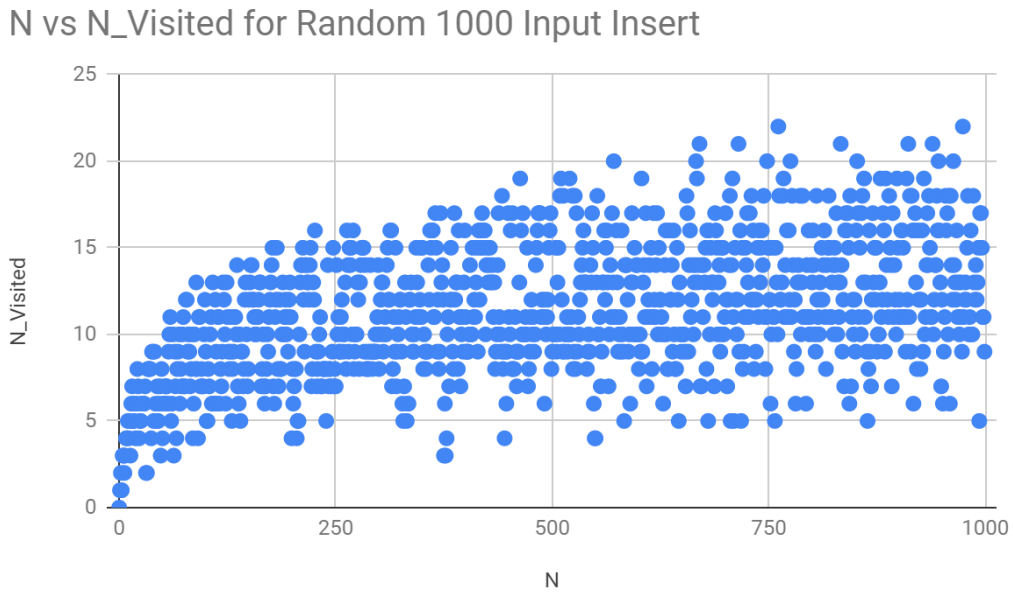
The worst case scenario is better for random cases. When searching in a random dataset, the root node would initially be compared to the movie being searched for. Then search() would either return true if the root was the target movie or recursively traverse down the left or right subtree of the root. This means that not every node would need to be visited. The worst case scenario is when the target movie is at the lowest depth of the binary search tree. Unlike the ordered case, the lowest depth in the random datasets can contain more than one node. Searching would thus be more efficient, and the total runtime would not increase as much for bigger datasets.

Part 2: Inserting

Scatter Plots of Ordered vs Random for 1000 movies



The number of nodes in the binary tree before inserting a new node (N) was the same as the number of nodes visited before inserting a new node (N_Visited) for ordered input.



The number of nodes visited before insertion gradually increased as the number of nodes in the binary search tree before inserting a new node (N) increased from 0 to 1000 for random input.

Code for Insert() function

```
// inserts newMovie into BST
bool MovieList::insert(Movie& newMovie){
    if(!(root)) {
        root = new Movie;
        root->setTitle(newMovie.getTitle());
        root->setRating(newMovie.getRating());
        node.push_back(0);
        nodes_visited.push_back(0);
        total_nodes++;
        return true;
    }
    else {
        insert( &newMovie, root);
    }
}
```

```
// recursive helper for insert (always call root as n outside of this function)
bool MovieList::insert(Movie& newMovie, Movie* n){
    if(newMovie.getTitle() == n->getTitle()) {
        return false; // return false if movie with same title is already in BST
    }
    else if (newMovie.getTitle() < n->getTitle()){
        if(n->left) {
            return insert( &newMovie, n->left);
        }
        else {
            n->left = new Movie(newMovie.getTitle(), newMovie.getRating());
            n->left->parent = n;
            node.push_back(total_nodes);
            total_nodes++;
            nodes_visited.push_back(getDepth(newMovie.getTitle()));
            return true;
        }
    }
    else {

        if(n->right) {
            return insert( &newMovie, n->right);
        }
        else {
            n->right = new Movie(newMovie.getTitle(), newMovie.getRating());
            n->right->parent = n;
            node.push_back(total_nodes);
            total_nodes++;
            nodes_visited.push_back(getDepth(newMovie.getTitle()));
            return true;
        }
    }
}
```

Big O Analysis for the insert() function

The insert() function uses recursion in order to insert a node into the binary search tree. For the ordered case, there would be no left children because every node inserted would simply be the right child of the node before it. However, this would mean that every node in the binary search tree would have to be visited until a node with no right child is found, which is the end of the tree. The runtime complexity for this implementation in the ordered case is O(N), where N is the number of nodes in the tree before insertion.

For the random case, the insert() function has a better runtime than for the ordered case. Left children would have to be considered in this case as well. Each node's information would be compared to the information to be inserted. Traverse down the tree until there is an available spot for the node to be inserted, and insert the node into that spot. Not every node in the tree would have to be visited, but the worst case is when the function would need to traverse to the lowest level of the tree to insert the new node. Thus, the runtime complexity for the random case would be O(log(N)).

The scatter plots also support the big Os for the insert function. For the ordered case, the scatter plot is a straight line where the N is the same as N_Visited, so the big O would also be O(N). For the random case, N_Visited continues to increase at a decreasing rate as N increases, meaning that its big O would be O(log(N)).