Arthur Flores

Professor Riggins

CS310

Program 4

7 May 2015

Program 4 Reports

**Binary Search Tree:**

The binary search trees are a linked structure that was designed to solve some of the issues with linked lists. The binary search tree takes advantage of its architecture and speeds up the search, insertion, and deletion operations. The best performance for the binary search tree is O(log n) and the average performance is in fact O(log n) but for the worst case you find a O(n) performance. This lack of performance in the worst case is mainly due if the data that you are inserting, deleting, or searching for is already in sorted order. This completely defeats the purpose of the binary search trees structure.

Searching within a binary search tree is fairly simple. Since the search always starts at the root node it cuts the search from linked lists O(n) operation into a O(log n) because if the node you are searching for is bigger than the root then you go to the right node and if less than the root you got to the left node. This in theory cuts the search in half making it O(log n). You will find the worst case for searching when you have data that is already in sorted order. As you can see below with the BST there is a O(log n) complexity due to the slow positive slope from the start of 100000 to 1000000.

Insertion in the binary search tree is logical task that on average takes O(log n). You get this performance of O(log n) due to the fact finding the correct position to place the node is fairly simple and also very efficient. What insertion does is if the data in the node is less than the data at the root then it is placed to the left of that node otherwise it is greater than so it goes to the right of the root node. This node keeps going down the tree making the same comparison until it finds a null area to get dropped in to. O(log n) performance is in random unordered data. If the data is already ordered that is when we see the worst case of O(n) performance because each node will go to the left or right of the root node and the node after that. As you can see below in the insertion graph there is a slight positive slope indicating the O(log n) complexity.

Deletion in the binary search tree has a little bit more difficult task. In order to remove an element from the tree you need to search while keeping track of the parent node and if you had gone left in the tree. That extra information is critical for figuring out where to move the nodes below the node that is being deleted. This performance is all done in O(log n) with unordered data. The only way to get performance of O(n) is if the data is already in sorted order as previously mentioned in the other two performances. As seen below in the deletion graph there is a slight positive slope as well as the other graphs indication the O(log n) complexity.

**Hash Table:**

A Hash table is probably the best performer out of the three data structures that I am going to discuss. Hash table are arrays of linear lists. A hash table is an array-based structure so if you know the index you are looking for you can find it instantly. This gives a hash table O(1) performance on best and average cases for insertions, deletion, and search operations. This is an outstanding number. For worst case when the table is almost full it can have an O(n) performance, which is a huge downgrade. Making the table 30-50 percent larger than the max capacity can prevent this worst condition from happening.

Searching within a hash table is fairly easy because the key given by the user is the index of where the information for that particular key is stored. This gives instant access to information of some tables that have lots of data to be stored. Although to not waste space a hash table would shorten the keys to make chaining possible. Chaining is used where there is a linked list for every array index. This makes storing information still O(1) due to fairly little amount of the same keys. As seen below there is a O(1) complexity with the data time all being around the same time and not increasing throughout larger data.

Inserting within a hash table has a little bit of a tricky side to it. The concept of chaining is very relevant to insertion. While inserting in the hash table it looks at a particular number of key numbers. If you are inserting into the hash table and get a hash code for your insertion and it is the same for another key then this data will be stored in a linked list array of that particular hash code. This type of strategy is what gives the hash table a O(1) performance. As seen below this insertion method in the hash table proves the O(1) complexity due to the very linear graph that even goes down in slope.

Deletion within the hash table is very similar to the addition concept with chaining. Once given the key for the deletion you use that key to get the hash code which will take you directly to the index in the array where that hash code is and then it looks at the values within the linked lists and compares it to the one you want to delete. This is a very efficient process due to the fact that you do not have to shift or move anything once it is deleted. This deletion graph proves the O(1) complexity due to the non existing change in timing. Almost all points are around 80ms.

**Red Black Tree:**

For the red black trees they are very similar to binary search trees but there is one key difference. The red black trees solve the problem for the worst case in the binary search tree. The best, average, and worst case performance for the red black tree is O(log n). The only thing that is not so good is that red black trees imply an additional overhead storage to store the color in each node. A red black tree is an implementation of a balanced tree where all data to the left is less than the previous and greater will go to the right. The difference between a red black tree is that each node has a color to keep its balance.

Searching is done just like it would be done in a binary search tree. You would start at the root and compare data until you have found your node. Since there is no removal or insertions in the tree, the rebalancing of the tree becomes unimportant. This makes the extra storage very wasteful. Here shows the search method of getValue and it proves the O(log n) complexity with the slightly higher value throughout the datas doubling input.

Inserting into the red black tree is very similar to the binary search tree. The new node is always colored red therefore when going down the tree to insert the parent node of the newly inserted is red a violation occurs and the rebalance of the tree needs to be done. This rebalancing is a very quick task and still preserves the O(log n) performance. Below is the insertion graph which has a slight increase from the start of the line indication a O(log n) performance.

Deletion is pretty much the same as insertion. You go down the tree starting at the root and delete and move the nodes attached to the parent like in the binary search tree and if there are ever two red nodes in a row then the tree needs rebalancing. Rebalancing keeps the height of the tree at 2\*log n. Below shows that the 2 constant in front of the log n doesn’t really effect the data proving the O(log n) complexity for all of the methods.

Below are timing test for the BST, Hash table, and red/black tree which also prove each individual complexities. This data shows the timing test for unordered and ordered data. The BST is the most noticeable in the ordered data proving its worst case complexity of O(n) due to each child being a rightchild upon insertion.

|  |  |  |
| --- | --- | --- |
| **Random Ordered** | **100000 searches (ms)** | **1000 reverse lookups (ms)** |
| Hash Table | 86 | 330 |
| Binary Search Tree | 202 | 591 |
| Red/Black Tree | 118 | 438 |

|  |  |  |
| --- | --- | --- |
| **Ordered Data** | **100000 searches (ms)** | **1000 reverse lookups (ms)** |
| Hash Table | 79 | 332 |
| Binary Search Tree | 6008 | 549 |
| Red/Black Tree | 117 | 302 |

Throughout testing multiple amount of array sizes, I have come to a conclusion that for the BST I can perform around 300,000 amount of searches in 1 second. In the Hash table I concluded with around 500,000 amount of searches per second. Lastly, I came to a conclusion of around 500,000 amount of searches per second for the Red/Black Tree. These timings mean that the Binary Search Tree timing is the fastest out of all the methods for doing n amount of searches in a second.