PA Report: Building and Analyzing properties of a Binary Search Tree

Minh Luu
CSCE 221-509

Date: March 16th, 2020

Objectives:

In this assignment, we are introduced to another data structure called binary search tree. We will write the data structure in C++ by implementing various properties. We will have two classes: Node for each node of the tree and BSTree for tree itself. Besides learning about the data structure, the BSTree member functions require an extensive use of recursive programming so it was also a great opportunity to learn about recursion. By the end of this assignment, I will be able to add another data structure and programming technique to my toolbox and hopefully use it on various applications in computer science soon.

Procedure:

This programming assignment is divided into three parts: building a binary search tree, populating the class according to the given instructions, and analyzing its search time cost.

The data structure is relatively simple: Each tree starts with a single root that contains data as well as references to left and right nodes. Each left and right node can also contain data and references to its own left and right node, respectively. This pattern continuous recursively, forming a tree. Beside the basic functions like constructors, size, height, insert, and search, my program extends this idea further by adding each node search time and average search time to analyze three different ways of inserting values into the tree and how they affect search performance.

The building part was quite straightforward. I just built all the functions required in the given code file. Due to the nature of a binary search tree, I had to create helper functions for executing many actions recursively such as insert, search, update search time, and print to console and text files. The final output files are exported to data-output-files folder with each node being print with its search time and an endl. An empty placeholder is created at the beginning so I can commit my codes to GitHub for storing. When you do make clean, it will also clean all output files.

For individual search cost, I have a helper function, inspired by the level by level print function, that takes in two parameters: Node for the root and int for level (aka the search time). I assign the current level to the current node and then recursively call its left and right child with level incremented by 1 until the child(ren) is a nullptr. Since I must visit every single node to assign/update its search time, the running time is O(n).

For average search cost, both the average search time function and total search time function are pre-implemented. I can see that we use the total search time function to recursively go down the tree and add all nodes' search times together. Then, we just convert the total search time to a float and divide it by the tree's size and we get the average search time. Once again, the total search time function requires going to every single node to add its search time to the total search time, so the running time is O(n).

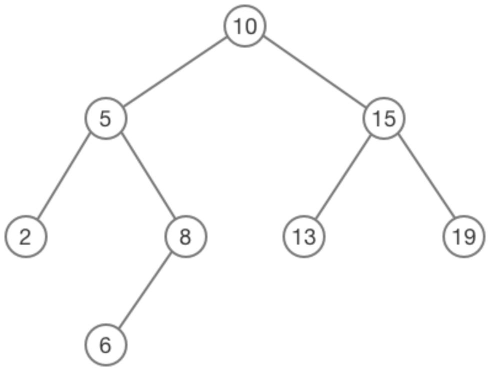

Fig 1: A basic binary search tree. Source: brilliant.org

Populating a tree and testing all the functions I created was the next part. This part was also straightforward because most of the code was already pre-written for me and all I had to do was building a function for output the tree in order into a .txt file and adding some other small snippets of code to complete the program. My program is compiled via a make file and everything can be run using the command make all and then ./main via a Linux environment with g++ and gdb installed.

Finally, I analyzed the search time for three scenarios: linear insertion, perfect insertion, and random search time. I will talk about the results in the next section.

Results:

Below are the results we gathered during the lab with a brief description for each figure:

l

# of nodes	average search time
1	1
3	2
7	4
15	8
31	16
63	32
127	64
255	128
511	256
1023	512
2047	1024
4095	2048

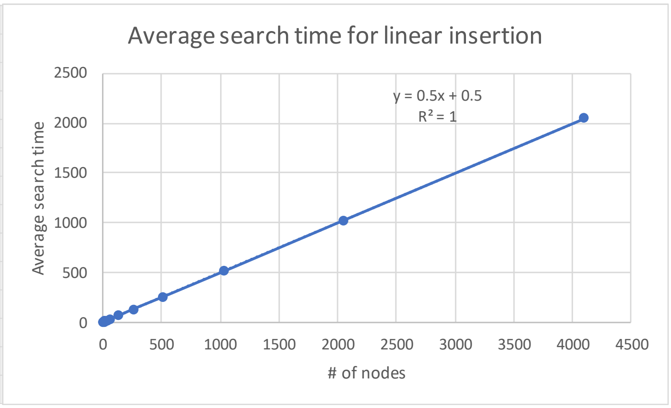

Fig 2: Table and graph for the average search time vs # of nodes for linear insertion

p

# of nodes	average search time
1	1
3	1.66667
7	2.42857
15	3.26667
31	4.16129
63	5.09524
127	6.05512
255	7.03137
511	8.01761
1023	9.00978
2047	10.0054
4095	11.0029

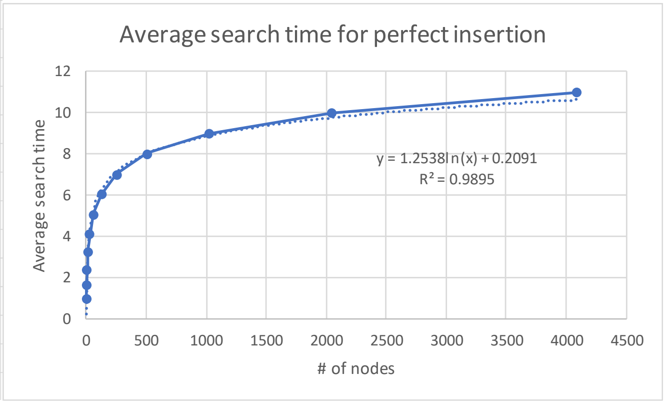

Fig 3: Table and graph for the average search time vs # of nodes for perfect insertion

r

# of nodes	average search time
1	1
3	1.66667
7	2.71429
15	3.73333
31	6.3871
63	7.66667
127	7.59055
255	9.06667
511	10.3033
1023	12.2463
2047	13.3972
4095	14.0237

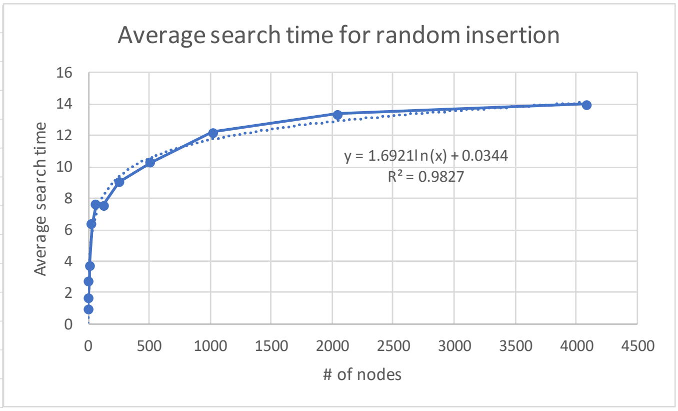

Fig 4: Table and graph for the average search time vs # of nodes for random insertion

Discussion:

The individual search cost is O(n) for linear insertion tree, while it is O(log(n)) for a perfect insertion and most cases.

On one hand, when we insert the values in an increasing order, the total search time is 1 + 2 + 3 + … + (n – 1) + n. That sum is equivalent to $\frac{n(n+1)}{2}$ and the average search cost would be $\frac{n(n+1)}{2n}$ that means the running time is O(n). Looking at figure 2, we can see that this is indeed the case and the equation we got is $y = 0.5x + 0.5$ and it is a perfect correlation.

On the other, if the values are somehow inserted to create a perfect tree, the total search time is $\sum_{d=0}^{log_2(n+1)-1} 2^{\mathrm{d}}(\mathrm{d}+1) \simeq (n+1)log_2(n+1) - n$ and the average search time is $\frac{(n+1)log_2(n+1)-n}{n} = log_2(n+1) + \frac{log_2(n+1)}{n}$ – 1. Therefore, the running time is O(log(n)). Figure 3 shows just that. The average search time grows logarithmically with the correlation being almost perfect.

These behaviors remind me of quicksort which has similar big O notations for the worst and the best case like the total search times above. It also has, statistically speaking, very close to the best case when it comes to the running time of a random case, and this leads me to the random insertion of values into a tree. As we can see in figure 4, the average search time of random insertion has a very close resemblance to the perfect insertion case. Unless intentional, it is extremely unlikely that we insert values in a tree in a linear fashion, so for most cases, binary search tree retains its high searching performance close to a perfect scenario.

Conclusion:

In conclusion, I have successfully implemented a binary search tree class. Through search time analysis, I have figured out the advantages of binary search tree compared to other data structure such as linked list: search time of almost all cases is O(log(n)). I was also able to observe its behaviors with my own eyes which help me at remember many concepts for future considerations when working in the industry. Last but not least, since the structure of the tree correlate directly to its search performance, I understood why we have to use versions of binary search tree which support self-balancing like AVL, Red Black, and Scapegoat trees.