4190.407 Algorithms HW 3-2

Yeonjun Kim

2024-13755

Dept. of Computer Science and Engineering

2025-10-19

In this assignment, I implemented the binary search tree (BST) and the hash table.

Before running the program, you should place **`input.txt`** and **`op.txt`** in this directory. In **`input.txt`**, you write the elements to push into the data structure, separated by a new line(**`\n`**). In **`op.txt`**, you write one data operation at a line. e.g. **`INSERT 42`, `SEARCH 42`, `DELETE 42`**. If either of these files are not present, the program will not work properly.

Theoretical Comparison

The two have different theoretical complexities, as shown in **Table 1**. N denotes the number of elements in the data structure.

Table 1 (expected) time complexity comparison in big-O notation.

	Binary Search Tree		**Hash Table**	
	Average	**Worst Case (Unbalanced)**	**Average**	**Worst Case (Bad hashing)**
Search	$O(\log N)$	$O(N)$	$O(1)$	$O(N)$
Insert	$O(\log N)$	$O(N)$	$O(1)$	$O(N)$
Delete	$O(\log N)$	$O(N)$	$O(1)$	$O(N)$

While binary search tree has time complexity of $O(\log N)$ on average, hash table boasts $O(1)$ on average. However, the performance of BST depends on how balanced the tree is. At worst case, that is, when every node has only one child, the time complexity grows up to $O(N)$. The performance of hash table also depends on the hashing function. When hashing is implemented properly with a universal hash family, it generally exhibits a good performance. However, when one uses a deterministic or bad hash function, the complexities can grow up to $O(N)$.

Experimental Setup

The hash function was implemented by following algorithm. (same as the lecture slides)

Algorithm. Generating a hash function from a universal hash family

```
p <- least_prime_greater_than(M)
a, b <- random_int_between(1, p-1)

def hash(key):
    i1 <- (a*key + b) % p
    i2 <- i1 % M
    return i2
```

In my experiments, I generated testcases with N=5000.

- To generate the 'random' case, I sampled from N integers from [0, 10N]. That is, I used `random.sample(range(1, 10*N),N)`.
- To generate the 'sorted' case, I used `sort()` in Python.
- To generate the 'duplicated' case, I duplicated 20% of the array elements into another random index. So, 40% of the array elements were duplicates.
- For runtime measurements, I used `time.process_time()` in Python.

Performance Comparison

Table 2 runtime comparison results (in milliseconds).

		Binary Search Tree	**Hash Table**
Random	**Search**	0.000837	0.000653
	Insert	0.000903	0.001163
	Delete	0.001092	0.000830
Sorted	**Search**	0.181067	0.000893
	Insert	0.180668	0.001367
	Delete	0.222734	0.001060
Duplicated	**Search**	0.001322	0.000983
	Insert	0.002047	0.001453
	Delete	0.001912	0.001100

Analysis & Conclusion

In this experiment, I could observe the important characteristics of the BST and the hash table. In the sorted case, the running time for the binary search tree grew dramatically, whereas the running time didn't differ very much in the duplicated case. Meanwhile, the hash table showed a rather robust performance in all the random, sorted, and duplicated cases.

This experiment result suggests that a good binary search tree should be well balanced, as the time complexity can grow up to $O(n)$ when the input is given in sorted manner. Red-Black tree is one of the ways to address this kind of problem. Also, the hash table is very powerful regardless of the input order, given that the hashing is implemented well.