Double Hashing Performance Evaluation

Kazi Shadman Sakib
Roll: FH-97
3rd Year Undergraduate Student
Department of Computer Science and Engineering
University of Dhaka

11th August, 2022

1 Description of Dataset

The full dataset preparation was done by using python programming language and it's built-in random module. The dataset was prepared according to the guideline given.

1.1 Set of Keys

By using the random module of python programming language, 10000 unique random numbers from integer range were generated. And these 10000 unique random set of keys were stored in a file named 'setOfKeys.txt' for later use.

1.2 Insertion Sequence

By using the random module of python programming language, a random insertion sequence for those 10000 set of keys were generated randomly and uniquely. And these 10000 unique random insertion sequence were stored in a file named 'insertionSequence.txt' for later use.

1.3 Search Sequence

By using the random module of python programming language, a random search sequence of 3000 numbers were generated. The keys of search sequence were generated with 0.7 probability of belonging to the key set and 0.3 probability of not being in the key set. That is, a random search sequence of 2100 numbers were generated that belongs to the key set and 900 random search sequence were generated that are not in the key set. Then, these 2100 keys and 900 keys were merged and randomly shuffled before storing them in a file named "searchSequence.txt". It should be also mentioned that the keys in the search set have 0.2 probability of repetition.

1.4 Insert-Search Sequence

By using the random module of python programming language, a random insert-search sequence with 0.7 probability of the operation being insertion and 0.3 probability of the operation being a search event were generated. That is, 7000 events were generated where the operation is an insert operation and 3000 events were generated where the operation is a search operation. Thus, they were shuffled randomly and was stored in a file named "insertSearchSequence.txt". It should be mentioned here that, in the insert-search sequence, "1"s are the insert operations and "0"s are the search operation.

2 Double Hashing Implementation and Details

The program starts with an "init()" function, which initiates our program by reading the four files of dataset that were generated after running "dataset.py" script from the "Dataset" directory. We store all relevant the data from the dataset to four arrays named, "setOfKeys[10000]", "insertSeq[10000]", "searchSeq[3000]" and "insertSearchSeq[10000]".

We start our operations using the "insertSearchSeq[10000]" array in a for loop. The conditions inside the for loop detects if it is an insert operation or a search operation by comparing if the value of "insertSearchSeq[i]" is 1 (insert operation) or 0 (search operation).

For the implementation of Double Hashing, we have selected two good hash functions, i.e, "h1(int key)" and "h2(int key)", that are used to build the hash table "hashTable[SIZE OF HASH]". The two hash functions are being used in two functions, named, "insertHash(int hashTable[SIZE_OF_HASH], int key)" and "searchHash(int hashTable[SIZE_OF_HASH], int key)".

Here,
hash1 (h1) = key % SIZE_OF_HASH
hash2 (h2) = 1 + (key % (SIZE_OF_HASH - 1))

The approach for double hashing insert is that, we first use the hash1 (h1) function and check if hashTable[hash1] has a 0 as it's value or not. If it has a value 0, then we insert the key into the hashtable and increase our insert probes count to the array "insertProbes[insertProbesCount]". If the value is not a 0, then, we use the double hashing formula, that is,

hash = (hash1 + (i * hash2)) % SIZE_OF_HASH;

We use a while loop with a condition of i¡SIZE_OF_HASH to perform the double hashing. We check if the hash table with the hash key has the value 0 or not. If it does not have the value 0, we increase i.

if it does have the value 0 then we found out spot for the key to be inserted into the hash function. We increase the insert probe count here too.

The searchHash(...) function searches the given key into the hash table and increments the search probe array, "searchProbes[searchProbesCount]" accordingly if the key is found. The searchHash(...) function follows the same technique as inserHash(...) function by using the hash1 function first. If we do not get our key using the hash1 function, then, we use the double hashing formula to search the key deeply into the hash table. The conditions of the while loop and inside the while loop are the same as inseartHash(...) funtion, but we gave an extra condition here to know that the key is not in the hash table. If at any moment we find a value 0 in the "hashTable[hash]", then we will know that the key is not in the hash table.

Thus, the Double Hashing implementation ends.

3 Data Structure Implementation and Details

In this part, we implemented the Binary Search Tree data structure to compare it with the double hashing in the means of performance evaluation.

Just like before, the program starts with an "init()" function, which initiates our program by reading the four files of dataset that were generated after running "dataset.py" script from the "Dataset" directory. We store all relevant the data from the dataset to four arrays named, "setOfKeys[10000]", "insertSeq[10000]", "searchSeq[3000]" and "insertSearchSeq[10000]".

We created a structure for node with data, it's left child and it's right child. Then we used that node structure to build our binarySearchTree structure to implement Binary Search Tree data structure.

As we start our insert-search sequence operations, the program uses this Binary Search Tree to insert or search for a given key.

For our insert operation we used "insert(int data)" function to call our main "*insertData(node *root,int data)" function, which mainly saves the inserted key. We used recursion technique for the insert operation to the binary search tree. This "*insertData(node *root,int data)" function creates a new node "*cur" for the insertion node and checks if the root is NULL or not. If the root is NULL, then, we add the "*cur" value to our root, which holds the data that is to be inserted. Then, again, if the root is not NULL the we check if the data/key that is to be inserted is greater than or lesser than the root data. If the key is greater than the root data, then by the definition of binary search tree we

go recursively to the right child of the root to find the NULL, i.e "root-¿right = insertData(root-¿right,data)". But if the key is less than that of the root data, then by the definition of binary search tree again, we go recursively to the left child of the root i.e "root-¿left = insertData(root-¿left,data)".

The recursion happens until the root is a NULL (which is our base case) and insert the new key to the binary search tree. It is to be mentioned that we recorded per operation steps count for to reach the main insertion root by using the counter variable.

For our search operation we used "search(int data)" function to call our main "searchData(node *root,int data)" function, which mainly searches the data/key given to it. We used recursion technique also for the search operation to the binary search tree.

We first increment the count as first step has been occured to search the key. Then, we check if the root is NULL or not. If the root is NULL then there exists no key what so ever. If the root data is equals to the data then we found our key, thus we return the count value, as we record this count value for the later use of comparing the performance of binary search tree with the double hashing technique. The count value of every per operation is recorded in the "searchProbes[searchProbesCount]" array for later use. If the root is not NULL and we have not found our data/key, we compare our data/key that is to be searched with the root data. If the searching key is greater than the root data then we recursively call "searchData(root-¿right,data)" to search the key. Again, if the searching key is less than the root data then we recursively call "searchData(root-¿left,data)". If anytime of the recursion we find a NULL root then we will know that our key is not is the tree. We count the steps of the search operation recursively too.

Thus, the Binary Search Tree implementation ends.

4 Comparative Performance

4.1 Comparative Performance Evaluation

The time complexity of Insertion or Searching in average case of Double Hashing is O(1) and in worst case is O(n). Where as, the time complexity of Insertion or Searching in average case of Binary Search Tree is O(log(n)) and in worst case is O(h), where h is the height of the Binary Search Tree.

There is no collision of data in Binary Search Tree, where as in hashing is some collision occurs we can use collision removal technique like double hashing in the hash table.

4.2 Visual Form

4.2.1 Insertion Operation

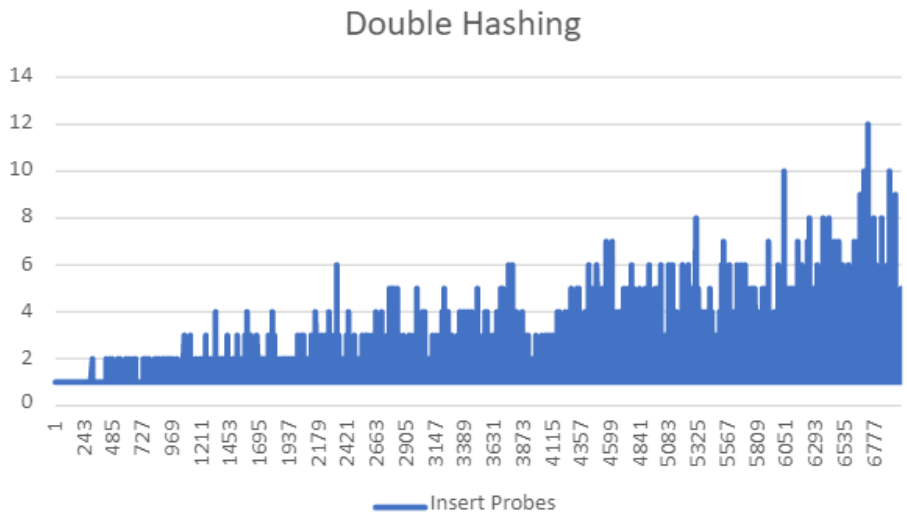

Figure 1: Double Hashing Insertion Probes

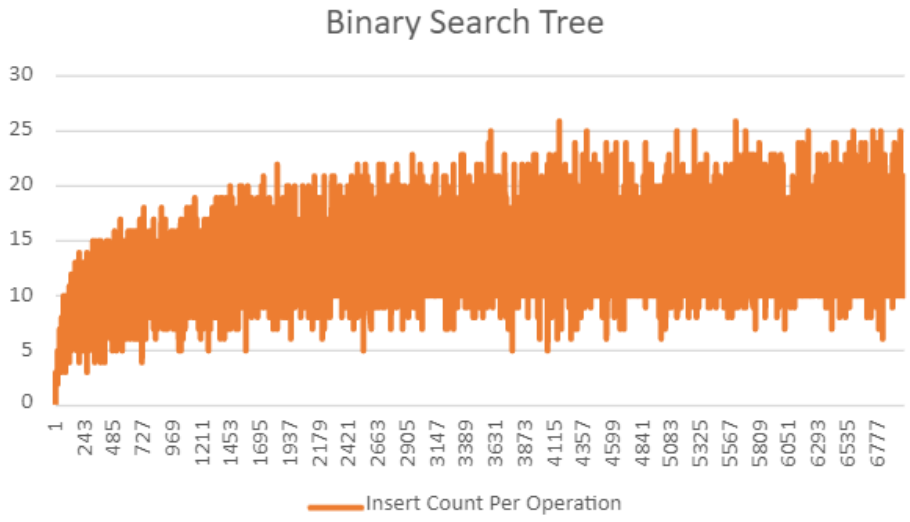

Figure 2: Binary Search Tree Insertion Counts

4.2.2 Search Operation

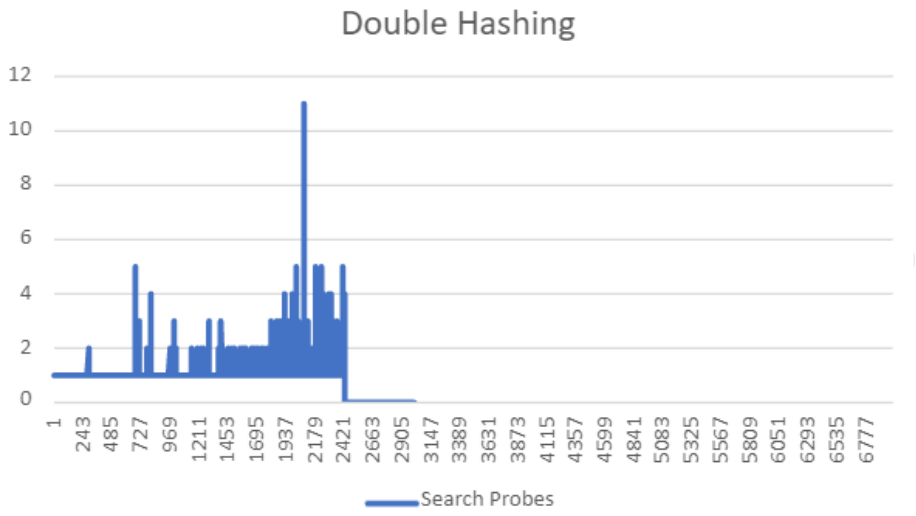

Figure 3: Double Hashing Search Probes

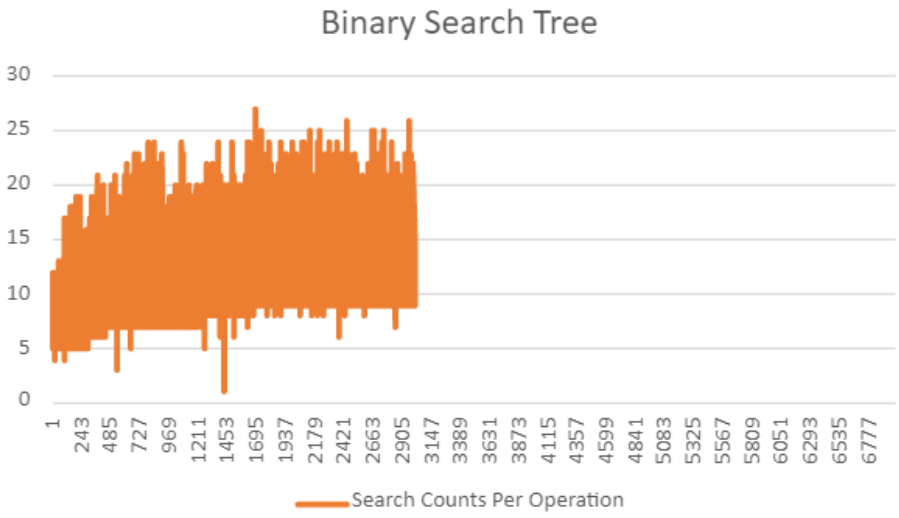

Figure 4: Binary Search Tree Search Counts

4.3 Tabular Form

Command to execute	Probe Count	Traverse Count
Search 24089	1	1
Insert 471	1	3
Insert 4196	1	5
Insert 8151	1	3
Insert 6397	1	4
Insert 1430	1	4
Search 90965	1	1
Insert 8239	1	3
Insert 157	1	4
Insert 6179	1	4

Result of the first 10 commands

Description	Double Hashing	Binary Search Tree
Summation	12834	170748
Maximum	13	36
Average	1.2834	17.0748

Statistics

5 Conclusion

In most circumstances, hash tables are faster O(1), but when they perform poorly, they can be quite awful. Although Binary Search Trees are slower in ordinary situations, binary search trees provide many benefits, including the ability to control the worst-case behavior.

The usage of BST and Hash Table depends on the need of the situation. If the input size is known then we can use the hash table and make some hash function that will generate the key uniformly. If the input size is not known to one in advance, then use the Hash Table. If we want to perform range search i.e. searching some key in between some keys, then we should go with Binary Search Tree because, in Binary Search Tree, we ignore that subtree which is impossible to have the answer.

Hash functions are the most important part of the Hash Table. So, if our Hash Function is such that it uniformly distributes the keys, then we should go with the Hash Table. But if we are finding it hard to make some Hash Function, then we go for Binary Search Tree.