Issued Date : Friday, May 13 2022 **Due Date :** Wednesday, May 25 2022

Special instructions : Must be done in teams of 8 people

Objective

The objective of this assignment is to compare how two different programming languages can solve a given problem.

Problem statement

You have to implement a **binary search tree** and a **hashing table** using two different programming languages for each algorithm, for a total of 4 programs. One of the programming languages must be Java, and the second one is the same as for assignment 1. You will also be evaluated on the choices of constructs and data structures selected from each programming language.

In order to achieve better comparison and testing of the programs, each implemented algorithm must read the list of elements to fill the binary search tree and the hash table from a file. Then the programs must read interactively at least 10 values to be searched where 5 must lead to successful searches. For each search element, you must record the number of accesses (e.g. $O_{(n)}$) and display the result in a log file. At the end of the process you must calculate the average search time for successful and unsuccessful searches for each implementation.
Just like in assignment 1, you must also record the memory usage of the data structures that store your data and the execution speed of the algorithms that populate the binary search tree and the hash table.

Hashing

A hash table is simply an array that is accessed using a hash function. For example, in Figure 1, *HashTable* is an array with 8 elements. Each element is a pointer to a linked list that stores all the data that map to this entry. The hash function for this example simply divides the data key by 8, and uses the remainder as an index into the table. This yields a number from 0 to 7. Since the range of indices for HashTable is 0 to 7, we are guaranteed that the index is valid.

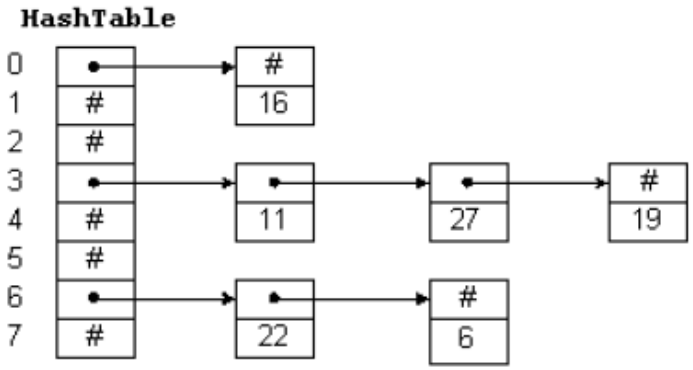

Figure 1: A Hash Table

To insert a new item in the table, we simply apply the hash function to the item key and then insert the item at the beginning of the list for that entry. For example, to insert 11, we divide 11 by 8 giving a remainder of 3. Thus, 11 goes on the list starting at HashTable[3]. To find an element, we hash the key for that element and chain down the correct list to see if it is in the table.

Entries in the hash table are dynamically allocated and entered on a linked list associated with each hash table entry. This technique is known as chaining. If the hash function is uniform, or equally distributes the data keys among the hash table indices, then hashing effectively subdivides the list to be searched. Worst-case behaviour occurs when all keys hash to the same index. Then we simply have a single linked list that must be sequentially searched. Consequently, it is important to choose a good hash function and also an appropriate size for the hash table.

Binary Search Trees

A binary search tree is a tree where each node has a left child and a right child. Either child or both children may be missing. Figure 2 illustrates a binary search tree. Assuming k represents the value of a given node, then a binary search tree also has the following property: all children to the left of the node have values smaller than k, and all children to the right of the node have values larger than k. The top of a tree is known as the root, and the exposed nodes at the bottom are known as the leaves. In Figure 2, the root node has value 20 and the leaf nodes have values 4, 16, 37, and 43. The height of a tree is the length of the longest path from the root to the leaf. For this example the tree height is 2.

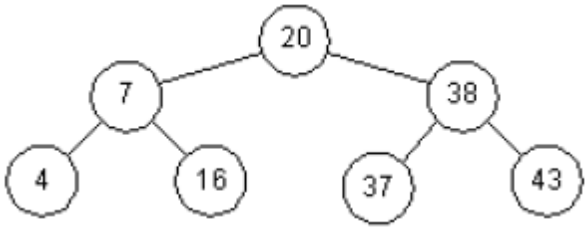

Figure 2: A Binary Search Tree

To search a tree for a given value, we start at the root and traverse down. For example, to search for value 16, we first note that 16 is less than 20 and we traverse to the left child. The second comparison finds that 16 is greater than 7, so we traverse to the right child. On the third comparison we succeed, thus it requires 3 accesses to find the search value.
Each comparison results in reducing the number of items to inspect by one-half. In this respect, the algorithm is similar to a binary search on an array. However, this is true only if the tree is balanced. For example, Figure 3 shows another tree containing the same values. While it is a binary search tree, its behaviour is more like that of a linked list, with search time increasing proportional to the number of elements stored.

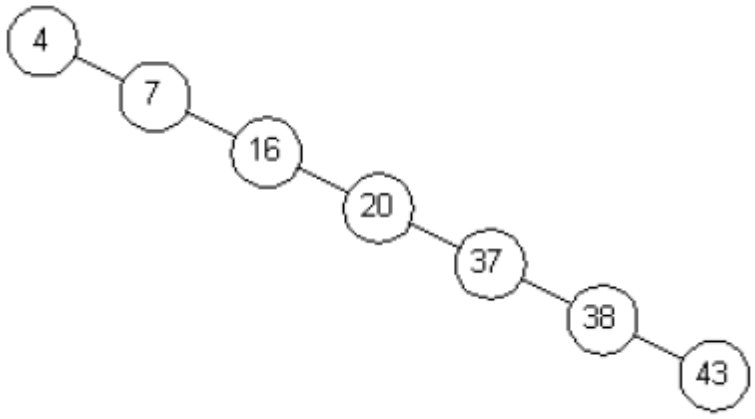

Figure 3: An Unbalanced Binary Search Tree

Let us examine insertions in a binary search tree to determine the conditions that can cause an unbalanced tree. To insert the value 18 in the tree in Figure 2, we first search for that number. This causes us to arrive at node 16 which is a leaf node. Since 18 is greater than 16, we simply add the node for 18 to the right child of the node of 16 (Figure 4).
Now we can see how an unbalanced tree can occur. If the data is presented in an ascending sequence, each node will be added to the right of the previous node. This will create one long chain, or linked list. However, if data is presented for insertion in a random order, then a more balanced tree is possible.

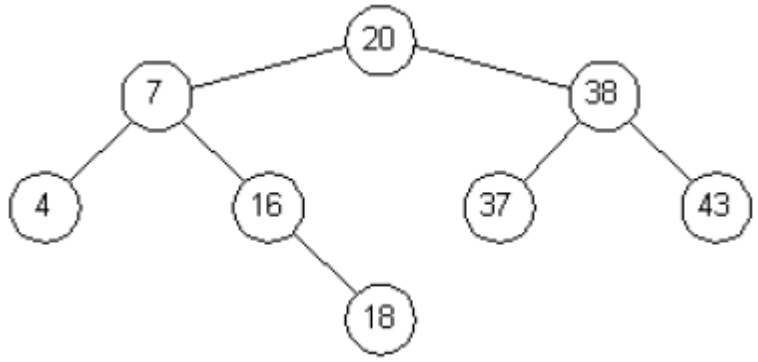

Figure 4: Binary Tree After Adding Node 18

Relationship with term paper

Each programming assignment should be done from the perspective of the comparative study to be made in the term paper. In the term paper, you will be asked to provide a comparative description of each of the programming languages you have used in your assignments. From the perspective of this assignment, you should be able to compare the performance of the searching algorithms for the two programming languages that you have selected.

Assignment submission requirements and procedure

You have to submit your assignment before midnight on the due date on moodle as "*programming assignment 2*". Late assignments are not accepted.
The file submitted must be a **.zip** file containing:

- all your code (i.e. 4 different programs) in text format and in executable format,
- one input file containing data to be read by the programs,
- one output file for each program containing the relevant statistics (i.e. number of accesses for each search value, the average search time for successful and unsuccessful searches, the memory usage and the execution speed),
- and instructions on how to compile and execute all your programs in a README file.

Evaluation Criteria

- Correctness of implementations (12 pts).
- Input and output files (2 pts).
- Output of relevant statistics enabling comparison (6 pts).