Data Structure & Algorithms Assignment

By

Ng Jih Bin (18093674)

Muhammad Afiq Aiman (18022145)

# **Table of Contents**

[**Table of Contents**](#_km4s4ffs371y) **2**

[**Introduction**](#_5dgsnljoa3k7) **3**

[Binary Search Tree](#_cs9ly0aoala4) 3

[Hashing](#_aqnfxy44ttq6) 3

[Programming Language and Tools](#_v5wvizl800cm) 3

[**Problem 1 Implementation**](#_cy1fqqvlpkhg) **4**

[Implementation](#_dw5be97iqyn0) 4

[Insert](#_unqg39qj73at) 5

[Search](#_a97u9k7pr3qe) 7

[Delete](#_i515ty2t4q8h) 8

[Tree Traversal](#_p4rci9xmu22q) 9

[User Interface](#_zb7hmfyd27db) 10

[Limitations](#_hv9dwtbpy3qk) 10

[**Problem 2 implementation**](#_w99dcylbiwnv) **12**

[Implementation](#_4x5meh2nee71) 12

[Hash Table](#_69p1kr8bot4a) 12

[Search](#_ar4bvltuld8c) 13

[Insert](#_norypb5suqu7) 14

[Delete](#_w1i761ny8n26) 15

[Display Hash Table](#_f6kezokxswa1) 16

[User Interface](#_ozrd69yktjau) 16

[Limitation](#_sahwevxoo5z8) 17

[**Individual member contribution**](#_l61fszyfluvk) **18**

[Muhammad Afiq Aiman](#_scrkjj2yyp40) 18

[Ng Jih Bin](#_otx2snr09m0z) 18

[**Challenges**](#_uqyj1kbvkwyp) **18**

[Testing](#_shttt9bgfv5o) 18

[Hashing](#_7t8kg9hs1jmb) 19

[**Conclusion**](#_4b95qdu3qys8) **19**

[**References**](#_9cmu14yc7rjw) **20**

# 

# 

# **Introduction**

The two problems that were chosen in this assignment are Binary Search Tree (BST) and Hashing.

## **Binary Search Tree**

According to Ruzankin (2019), Binary Search Tree is a tree data structure that is made up of connected node, whereby the left subtree contains nodes that have a lesser key than the root node, and the right subtree contains nodes that have a greater key than the root node. BST can be used in many applications like indexing and multi-level indexing, and helpful in implementing various searching algorithms and maintaining sorted streams of data (*Binary Search Tree: Introduction, Operations and Applications*, n.d.).

## **Hashing**

Hashing is a process done to improve the speed of inserting and searching elements in a hash table, it is done by using hash function(s) to compute indices, also known as hash codes, to map keys or values into a hash table (Konheim, 2010). There are many applications to hashing, applications like message digest, password verification, Compiler Operation, and etc (*Applications of Hashing,* 2018). The data structure of hashing used are ArrayList and LinkedList.

## **Programming Language and Tools**

Both problems will be implemented using Java programming language and the tool used in this assignment is Replit which is a collaborative browser-based IDE that supports more than 50 programming languages, using Replit can help our team to do the coding together online.

# **Problem 1 Implementation**

## **Implementation**

The first problem chosen is Binary Search Tree, where there are a few operations like searching, inserting, deleting, and tree traversals. The tree and operations are implemented into three classes, the BST class, the Node class, and the main class for user interaction.

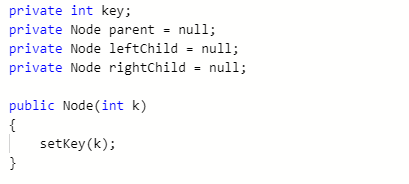


Figure 1: Variables of Node Class

In the Node class, there are four variables, key, parent, leftChild, and rightChild as shown above in Figure 1. There are individual setters and getters for each variable as shown in Figure 2.

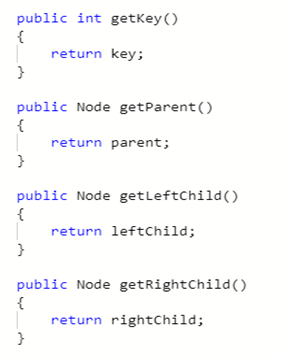
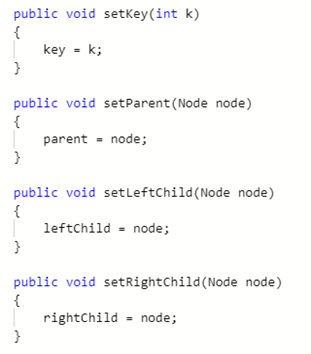


Figure 2: Setter and Getter For Each Variable

### 

### **Insert**

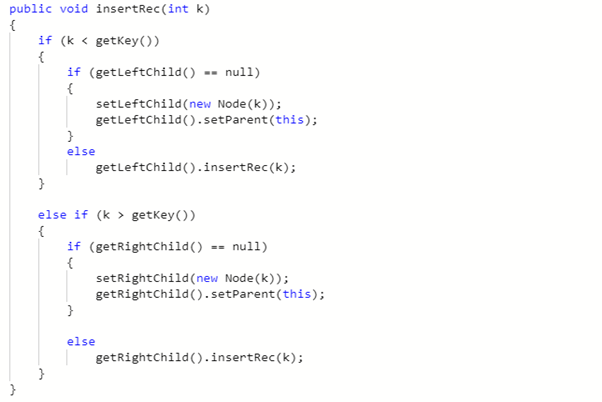


Figure 3: insertRec() Method

To insert a node into the BST, the insert() method as shown in Figure 4 in BST class will call the insertRec method as shown in Figure 3 in Node class. In the insertRec method, there are multiple if-else statements used. If the integer k value, which represents the number to be inserted, is lesser than the current key value, the integer k will be inserted into the left child of the current key value. However, if the left child is null or empty, set the left child with the integer k as a new node and set that node as the parent node. The same concept can be applied to the right child, if integer k value is larger than the current key value, the integer k will be inserted into the right child of the current key value. However, if there is no right child, set the right child with the integer k as a node and set that node as the parent.

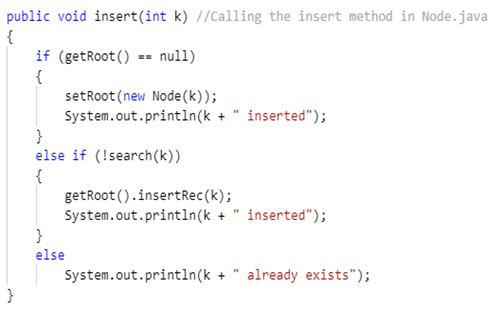
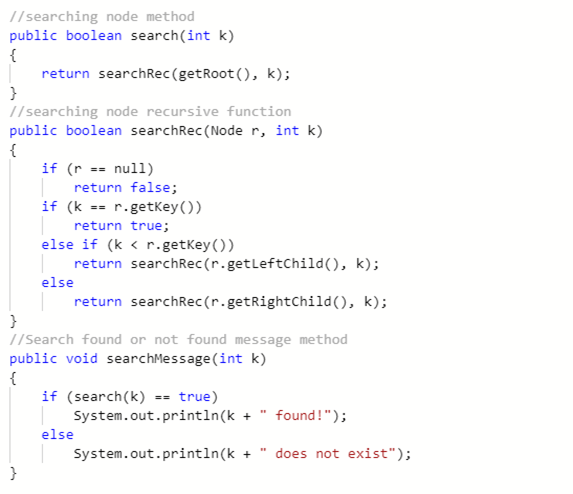


Figure 4: insert() Method

The insertRec method from Node.java will then be called inside the insert method in BST.java. As shown in Figure 4, if the root of the tree is empty, we just set the root node with a new node and the program will print a message saying “inserted”. However, if the root of the tree already existed, the new node to be inserted will be searched first and if the new node to be inserted does not exist in the tree, we call the insertRec method to insert the new node into the tree which will prompt the program to print a message saying “inserted”, but if the new node does exist in the tree, the system will print a message “already exists”.

### **Search**

****

**Figure 5: Searching Operation**

The searching operation is implemented using a recursive function as shown in figure . In the searchRec() method, a recursive function is utilized where if root(r) is empty, it returns false, but if the integer k, the value to be searched is equal to the root of the tree, it returns true. The same method is used for searching the left and right child, meaning if the integer k is not equal to the root but less than the root, the algorithm will search the left subtree of the root in a recursive function, but if the integer k is larger than the root, the algorithm will search the right subtree of the root in a recursive function. The recursive function is then called in the search() method.

Another method is then created to determine whether the node to be searched is found or not. Here we call the search() method and input the integer value to be searched, and if the search turns out to be true, the program will print a message saying “found!”. However, if it turns out to be false, the program will print a message saying “does not exist”.

### **Delete**

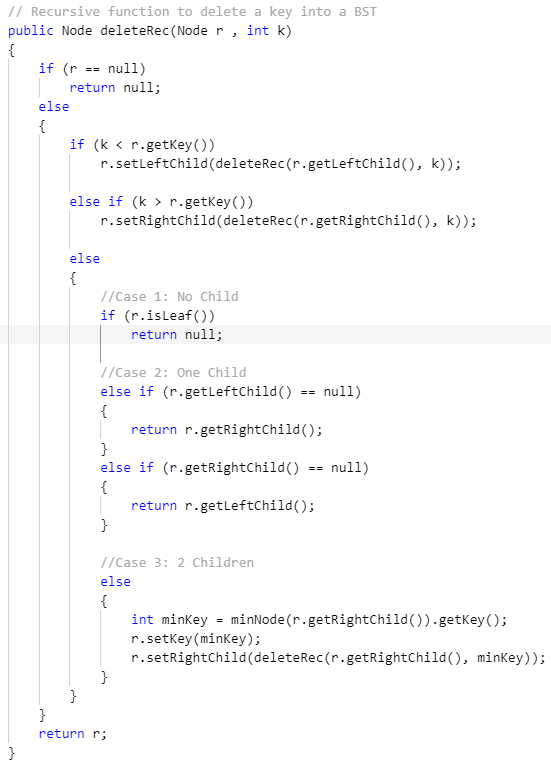


Figure 6: deleteRec(), Recursive Delete Function

To implement the delete operation for a BST, we first implemented a recursive delete function. As shown on Figure 6, we must first check if there is a node in the root of the tree, if there is no node then return null. However, if there is a node, then the node to be deleted or integer k must be checked if it's in the left subtree or right subtree. This can be done by setting a condition where if integer k is lesser than the root node, check the left subtree and if the integer k is larger than root node, check the right subtree tree. Once that is done, we delete the found node and rearrange the tree.

There are three conditions to rearranging the tree after deletion, if there are no children or in this case, the isLeaf() method in Figure 7 under the deleted node, we set the deleted node to null. The second condition is when there is one child under the deleted node, if the left child is empty, we replace the deleted node with the right child, if the right child is empty, we replace the deleted node with the left child. The last condition is when there are two children under the deleted node. In this case, the program will look for the node with the smallest value from the right subtree of the node to be deleted, and copy that value to the node to be deleted. Then, the node from which the value is copied from will be deleted using the same recursive function.

The delete recursive function is then called in the delete method, where if the searching of the node to be deleted or integer k is found, we set the root with the delete recursive method and print a message saying “deleted”. However if the search for the integer k is not found then we print a message saying “not found!”.

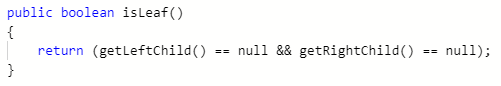


Figure 7: isLeaf() Method in Node.java

### **Tree Traversal**

The last operation is tree traversal, there are 3 traversals that are implemented, inOrder, preOrder, and postOrder. All three traversals use their own individual recursive function but with different ordering. As shown in Figure 8, we get the recursive function to arrange the left child first, then the node, and lastly the right child. After that the recursive function is called in the inOrder() method and printed. The same method can be used for both preOrder and postOrder but they are arranged differently. In preOrder, the node is arranged first, then left child, and lastly right child. For postOrder, the left child is arranged first, followed by the right child and lastly the node.

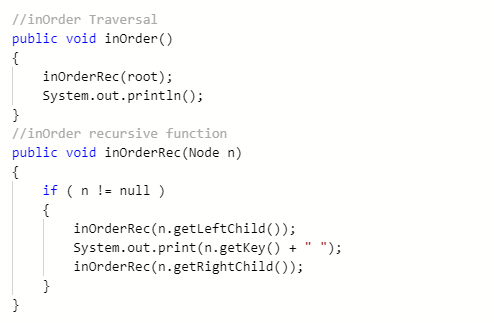


Figure 8: inOrder Tree Traversal

### **User Interface**

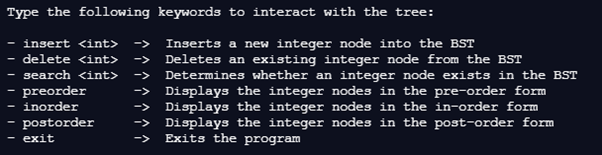


Figure 9: User Interface for User to Interact

For users to communicate with the program, we created a user interface that simulates a command-line interface. By creating an interactive user interface, we can carry out the BST operations while the program is running without having to end the program and change the values in the main.java every time the user wants to carry out a new operation like inserting new value or deleting new value.

## **Limitations**

Although our program implemented most of the operations, there are still some limitations to the program. First, the program cannot show a visualisation of the Binary Search Tree or shows the inserting and deleting process. It can only show a representation of the tree through the tree traversal. Secondly, the program cannot insert the same key again once it is inserted in the tree. Lastly, the program cannot handle all errors when there is rigorous testing with all possible inputs of value.

# 

# **Problem 2 implementation**

## **Implementation**

The second problem is Hashing, where there are a few operations like inserting, deleting, and searching a word in the hash table, as well as displaying the entire hash table. To avoid collisions in the hash table, the separate chaining technique is implemented. The hashing program is implemented in two classes, the HashTable class and Main class.

### **Hash Table**

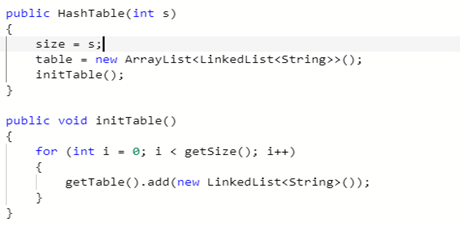


Figure 10: HashTable.java constructor

To create the hash table, we first use an ArrayList to create the hash table and a LinkedList to implement the separate chaining technique to handle collisions as shown in Figure 10. The initTable() method is used to instantiate an empty LinkedList for every index in the ArrayList, to avoid the problem of getting null values because when the array was created there were no LinkedLists in the ArrayList.

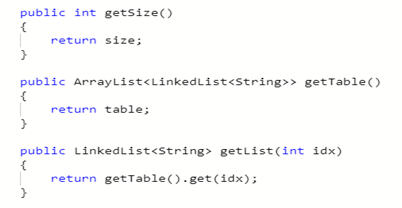


Figure 11: Getters

As shown in Figure 11, a getter for size is used to get the size of the hash table, the getTable() is used to return the table, and the getList() getter is used to get the LinkedList at an index in the ArrayList.

### **Search**

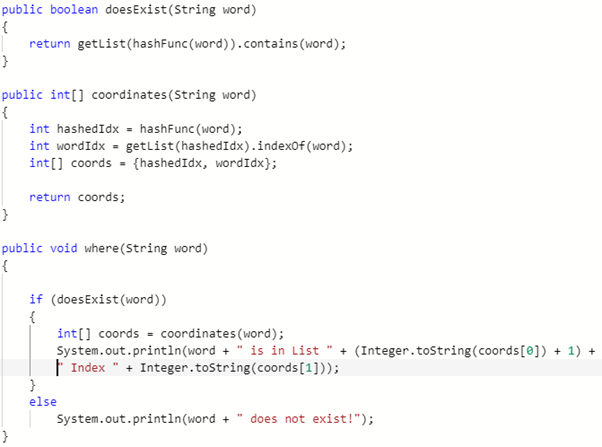
****

Figure 12: Search Method

To search for a String word in the hash table, we implemented a doesExist() method to ensure that the word in the argument exists in the hash table. Then, we implemented the coordinates() method to get the position of the String word in the hash table. Lastly, we implemented the where() method which will display the position of the String word is it exist in the hash table and print a statement saying the String word “is in the list” and its position “Index”, however if the String word does not exist, the program will print word “ does not exist”.

### **Insert**

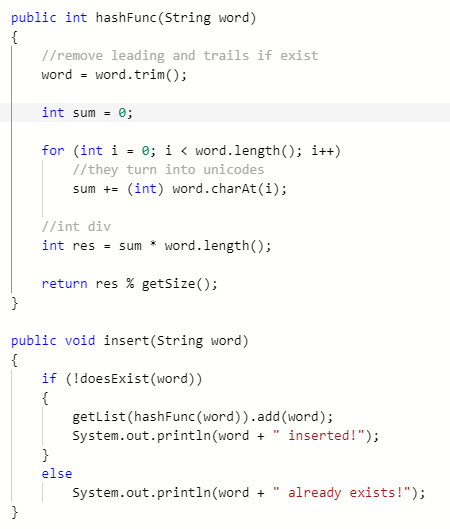


Figure 13: Insert and Hash Function

To insert a String word into the hash table, a hash function must first be used which is implemented using the hashFunc() method. As shown in Figure 13, the hash function used in this program, first we get the unicode of each character of the String word to be inputted which is accumulated in variable “sum”, then we divide sum by the length of the word which is finally modulo with the size of the hash table to get the hash code. To get the size of the hash table, we use the getSize() getter shown in Figure 11.

Then, to insert the String word into the hash table, we first check if the word is already in the hash table using the doesExist() method. If the word already exists in the hash table, we print a statement saying, “already existed!”. If the word does not exist, we use the insert() method to insert the hash code of the String word into the respective position of the hash table and print “inserted!”.

### **Delete**

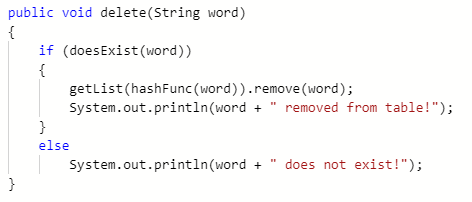


Figure 14: Delete Method

For the delete method, we first use the doesExist() method to check if the String word to be deleted is in the hash table. If it is in the hash table, we remove it from the array and print “removed from table!”. If the word is not in the hash table, we print a statement saying, “does not exist!”.

### **Display Hash Table**

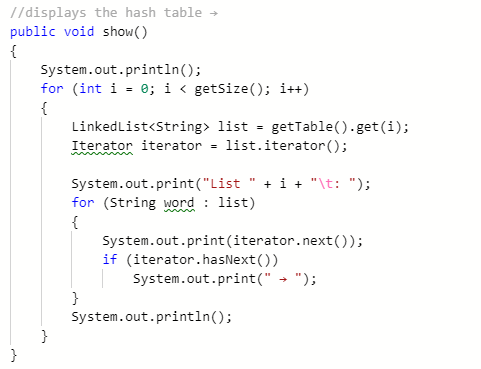


Figure 15: show() Method

The last operation of Hashing is to display the hash table, this can be done using the show() method as shown in Figure 15. In the show() method, we start by iterating through each LinkedList in the ArrayList. For every LinkedList, we print out all the words in the LinkedList and separate them with a right arrow symbol.

### **User Interface**

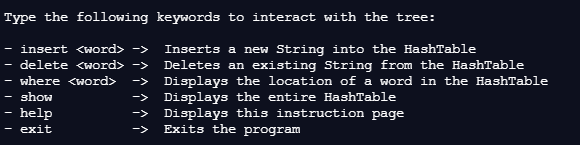
****

Figure 9: User Interface for User to Interact

Similarly to the BST implementation, we create a user interface that simulates a command-line interface for users to interact with. This interactive user interface provides easier access to the Hashing operations without having to end the program to make changes to the code like inserting a new String value or deleting a String value.

## **Limitation**

Although our program covers most of the operation in hashing, there are some limitations. Our program cannot change the hash table array size when the program is running, it can only be done manually in the coding itself before running the program. Another limitation is the data inputs, our program cannot run other data inputs except for String data inputs. The last limitation is that it cannot use other techniques to handle collision, our program can only use separate chaining techniques and none other.

# 

# **Individual member contribution**

## **Muhammad Afiq Aiman**

* **Code designer & manager & programmer (BST & Hashing, both UIs) (initiative)**
* Code tester
* Code editor & finaliser
* Document editor & writer

## **Ng Jih Bin**

* **Document designer & manager & writer (initiative)**
* Code programmer (BST)
* Resource finder
* Code tester
* Code editor & finaliser
* Referencer

Excellent cooperation utilising our individual strengths and complementing each others’ weaknesses. We help each other out in our distributed tasks. Good chemistry.

# **Challenges**

There are some challenges like that we faced when doing this assignment. Just like any other projects we have done in the past, bugs and errors in our codes will always continue to “bug” us every now and then. The toughest challenge was looking for reliable resources to learn from and understanding them. Other challenges include code testing, deciding the best structures to use, and creative thinking. However, with patience and excellent Good-searching skills (and thanks to Stackoverflow and very helpful Indian YouTube tutorials), we were able to overcome this challenge and learn many new things along the way.

## **Testing**

Firstly, there are many test cases that had to be tested on each program which would take a long time to test when changing values in the coding itself to compile and run. To solve this, we implemented a simple interface where users can enter keywords such as “insert 2” or “preorder” to have the instruction executed while the program is running for an easier testing phase.

## 

## **Hashing**

Another challenge we faced is during the hashing implementation. We ran into an error called generic array creation when we first used traditional java fixed arrays, thus to solve this, we switched to java.util.ArrayList to store the LinkedList.

We also encountered an error with the LinkedList, this error causes nulls value because when the array was created, there were not any LinkedList in the ArrayList. To solve this we used the initTable() method as mentioned in the implementation of hashing hash table.

# **Conclusion**

In conclusion, we have completed an implementation of Binary Search Tree (BST) and Hashing. In the BST implementation, there are operations like inserting a node, deleting a node, searching a node, and displaying the tree through the tree traversal. In the Hashing implementation, we have some basic operations like inserting a String value into the Hash table, deleting a String value from the Hash table, searching for a particular String value from the Hash table, and lastly, displaying the Hash table.

# 

# **References**

Ruzankin, P. S. (2019). *A fast algorithm for constructing balanced binary search trees*.

Konheim, A. G. (2010). *Hashing in Computer Science : Fifty Years of Slicing and Dicing* (1st ed.). John Wiley & Sons, Incorporated.

GeeksforGeeks. (2018, November 30). *Applications of Hashing*. [https://www.geeksforgeeks.org/applications-of-hashing](https://www.geeksforgeeks.org/applications-of-hashing/)/

*Binary Search Tree: Introduction, Operations and Applications*. (n.d.). from <https://afteracademy.com/blog/binary-search-tree-introduction-operations-and-applications>

mycodeschool. (2014, May 1). *Delete a node from Binary Search Tree* [Video]. YouTube. https://www.youtube.com/watch?v=gcULXE7ViZw