**DSA Assignment Report**

[**Application Description** 1](#_Toc33916605)

[**Basic Requirements** 2](#_Toc33916606)

[**Option 1: Searching for a Value** 3](#_Toc33916607)

[**Option 2: Adding a Value** 5](#_Toc33916608)

[**Option 3: Removing a Value** 6](#_Toc33916609)

[**Option 4: Displaying of Values in Ascending Order** 8](#_Toc33916610)

[**Option 5: Display value of kth node through Level-By-Level Traversal** 9](#_Toc33916611)

[**Option 6 & 7: Display Total Number of Items and Height** 10](#_Toc33916612)

[**Additional Requirements** 11](#_Toc33916613)

[**Option 8: Display the Tree** 12](#_Toc33916614)

[**Option 2 & 3: Self-Balancing of Tree** 12](#_Toc33916615)

[**User Validation** 14](#_Toc33916616)

[**Robustness** 14](#_Toc33916617)

[**Data Structures and Algorithms Implemented** 14](#_Toc33916618)

[**Binary Search Tree** 15](#_Toc33916619)

[**AVL Tree** 15](#_Toc33916620)

[**Queue** 15](#_Toc33916621)

[**Class Diagram** 16](#_Toc33916622)

[**References** 18](#_Toc33916623)

# **Application Description**

Our application is developed to create a Binary Search Tree with L levels and 2L-1 nodes. Afterwards, the various functions of the application that the user is able to use are searching for values, adding of values, removing of values, displaying the value of the kth node in the tree and display the tree, where these are the basic features of the application. Furthermore, this application also has additional features such as displaying of the tree in console and the tree having the ability to self-balance. Further additional features are implemented to ensure that the application runs smoothly with User Validation that make sure that the application is robust.

This report aims to explain the basic and additional features, data structures implemented, class diagram as well as the roles and contributions of the DSA assignment.

# **Basic Requirements**

The following are the basic requirements we have implemented in the application. Furthermore, our group has also added additional functions such as displaying the total number of items and height in the Binary Search Tree, as well as checking if the tree is balanced. In this part of the report, we will be explaining how each function works in our application. These functions showcase the fundamental operations that a Binary Search Tree is able to do. At a later part of the report, there will be additional requirements that our group has also implemented.

**Initialisation: Creation of a Binary Search Tree (BST)**

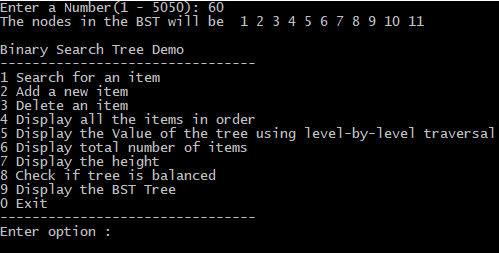
A Binary Tree is a data structure in which every node in the tree can have at most 2 child (either 0 child, 1 child or 2 child).

A Binary Search Tree (BST) is also an application of Binary Tree, in which all its subtrees are divided into two segments, the left subtree and the right subtree. All the nodes in the left subtree will have a lower value than the root, while all the nodes in the right subtree will have a higher value or equal to the root.

The BST also has a search operation whereby whenever an element is to be searched, it will start searching from the root node. If the data is less than the user input value, it will search for the value in the left subtree. Otherwise, it will search for the value in the right subtree.

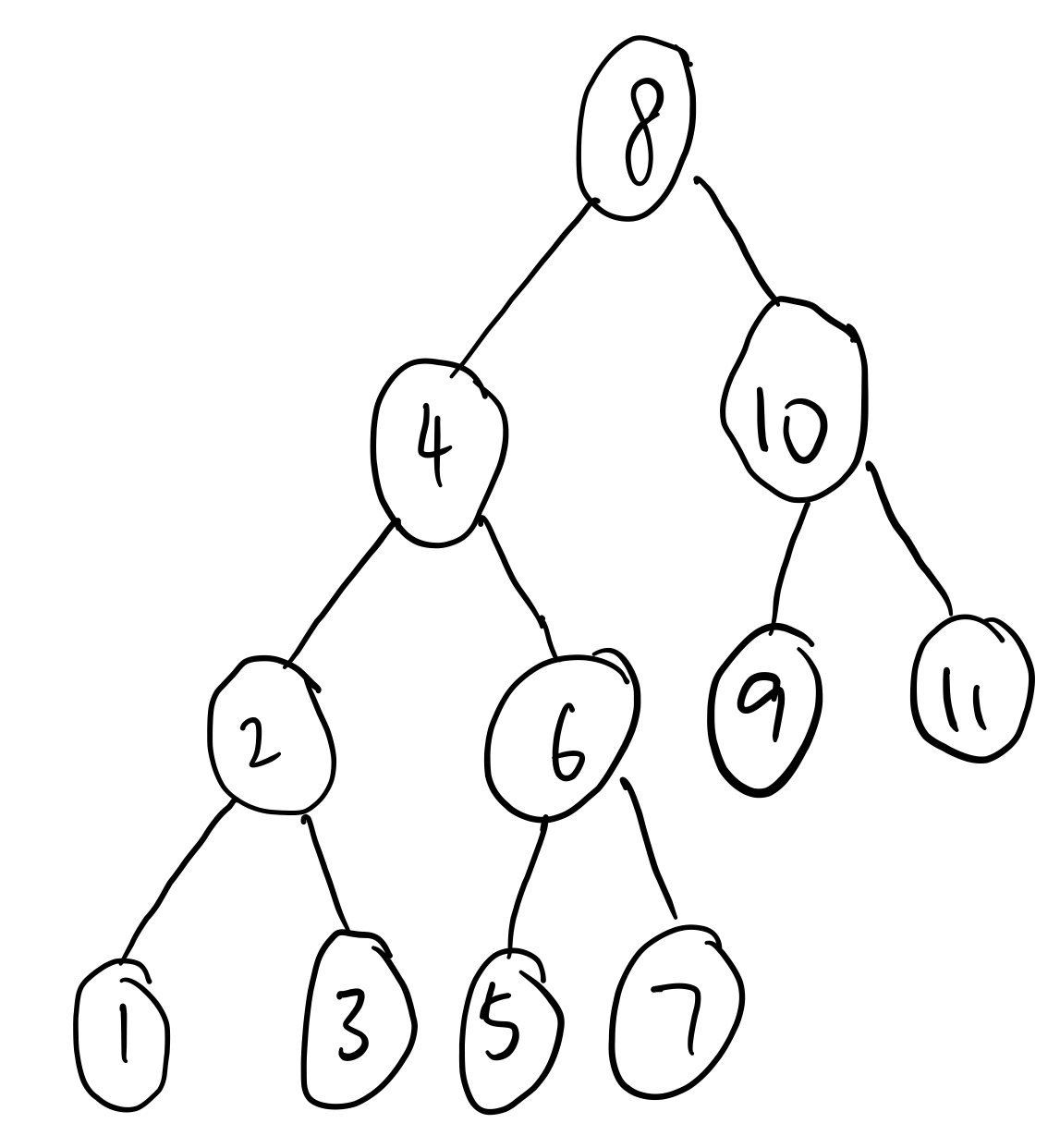
In the main program, the user will be immediately prompted for a integer value to create a BST. The value has to be an integer, whereby a sum of consecutive numbers will be initialized For this assignment, we have limited the consecutive numbers to range from 1 to 100, therefore the maximum user input is 5050. A randomiser will be initialized through vector (dynamic array) that helps to randomize the values, and the program will display a sum of consecutive numbers whose sum is larger than the integer given by the user.

For example, if the user has entered an input of 60, the consecutive numbers will be ranged from 1 to 11, where its sum will add up to 66, which is the smallest possible integer greater than the user input integer. The nodes in the BST for the rest of the application session will be ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘10’ and ‘11’.



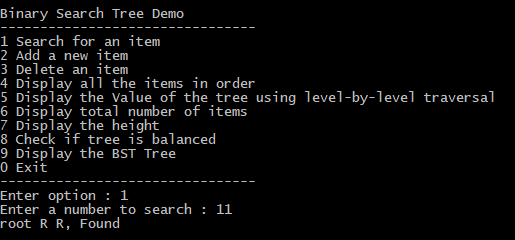
## **Option 1: Searching for a Value**

Our first option is to allow the user to input a value, whereby the application will search for it in the BST. When searching for the user input integer, the application first compare the value to the root. If both of the values match, it will return root. However, if the user input value is greater than root’s, it will recur for the right subtree of the root node. Otherwise, it will recur for the left subtree.

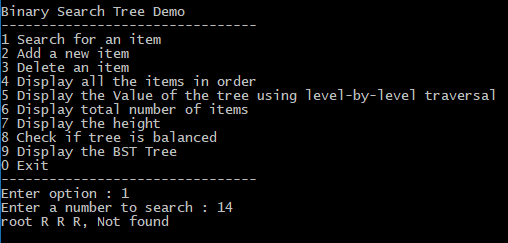


For example, based on the illustration shown above, the BST contain values of ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘10’ and ‘11’, searching for the value ‘11’ will return us with the output of “root R R, Found”. This means that the application first compared the value with the root. If the value does not match the root, if not, it will compare if the value is greater than the root, where if it is true, it will recur for the right subtree of the root node, otherwise it will recur for the left subtree.

In this case, the value ‘11’ is greater than the root value of ‘8’, therefore it will recur for the right subtree. The value of the right subtree is ‘10’, where ‘11’ is still greater, hence it will further recur for the right subtree, where it will find the value ‘11’.

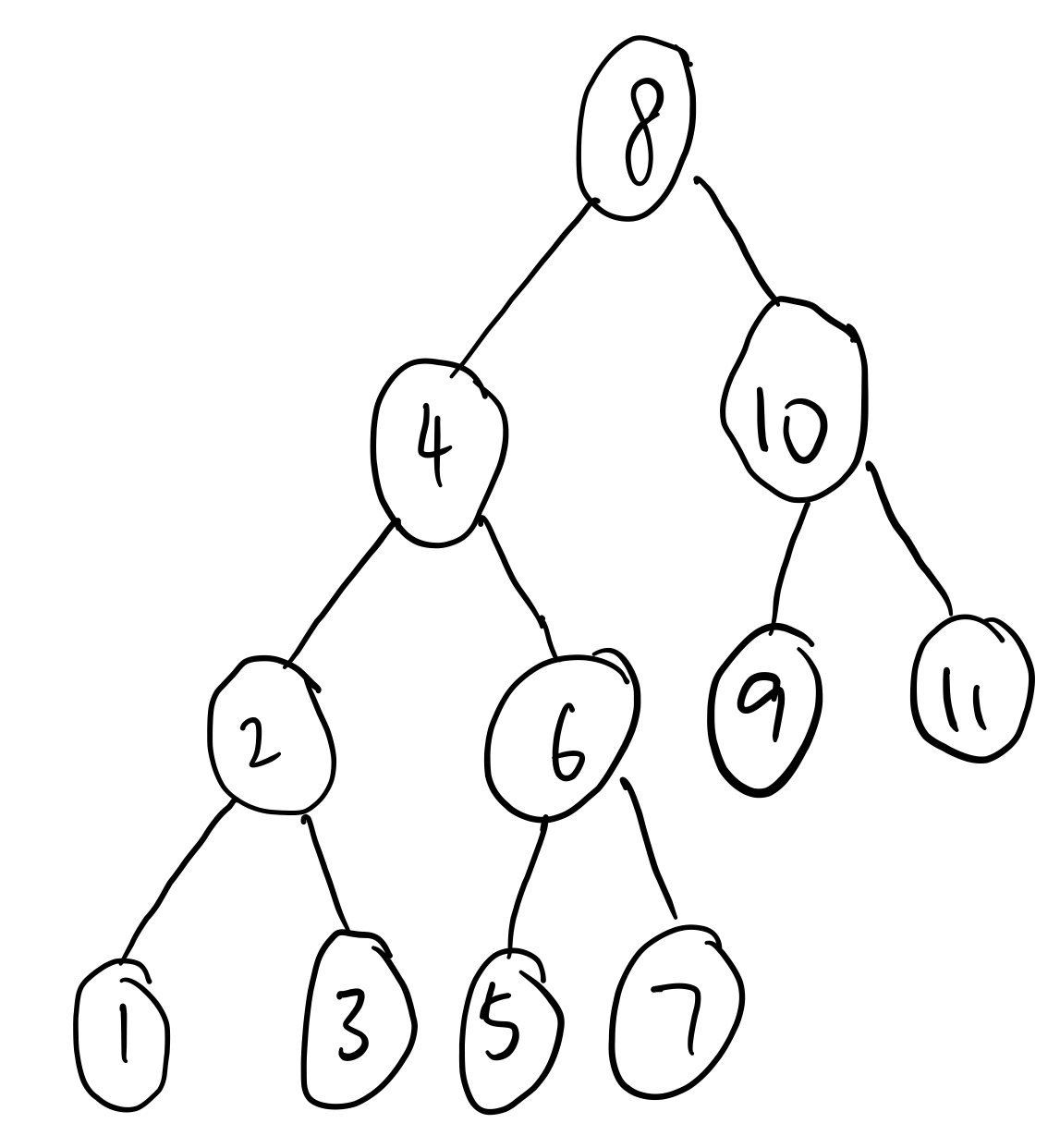


In another scenario where the user chose to search for the value of ‘14’. As the current BST does not contain this value, the application will return the output of “root R R R R, Not found”. The value ‘14’ is greater than the root value of ‘8’, therefore it will recur for the right subtree. As the value of the right subtree is ‘10’, ‘14’ is still greater. It will then further recur for the right subtree, where ‘14’ is still greater than the value ‘11’. It will recur for the right subtree once again, but because there is no child node, it will return null and hence not found.

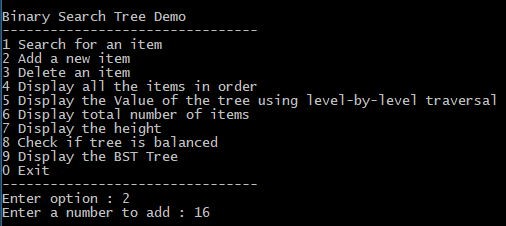


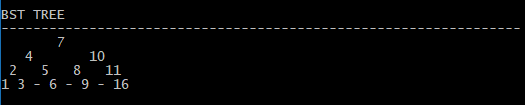
## **Option 2: Adding a Value**

Our second option is to allow the user to insert a value into its current list of values. The application will first ensure that the root is not null and the value does not exist in BST, before adding. The new value will always be inserted at a leaf node. A leaf node is a node that does not have a child. Therefore, the application will start searching from root till it reaches a leaf node.



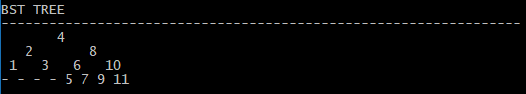
For example, based on the illustration shown above, if a new value ‘16’ is being added into BST, it will start off at the root, where ‘16’ is greater than ‘8’, so it will search in the right subtree. As ‘16’ is also greater than ‘10’, it will search in the right subtree of ‘10’, where ‘16’ is also greater than ‘11’. However, ‘11’ is a leaf node, therefore ‘16’ will be inserted as a right child of ‘11’.

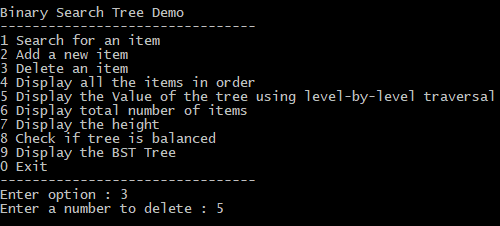


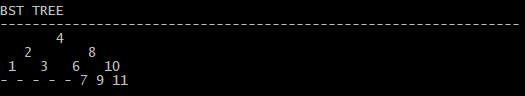


## **Option 3: Removing a Value**

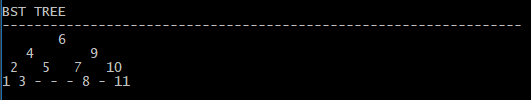
Our third option is to allow user of the application to remove an existing value from the BST. When deleting a node, there are many different possibilities that may occur. Referring to the three screenshots shown below, the first possibility is that if the node to be deleted is a leaf, it will just simply remove from the tree as there is no implication.

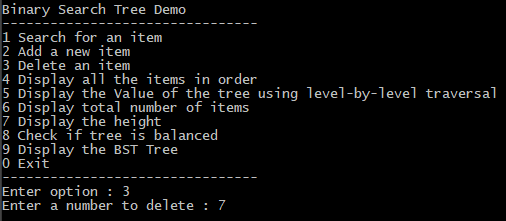


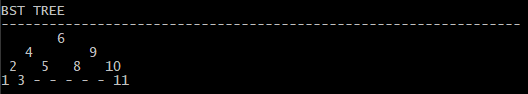




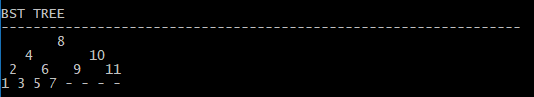
The second possibility is that if the node to be deleted has only one child, the application will copy the child to the node and delete the child. Refer to the three screenshots below for an example.

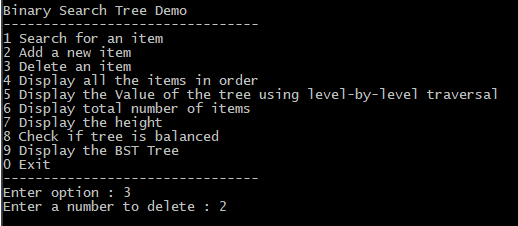


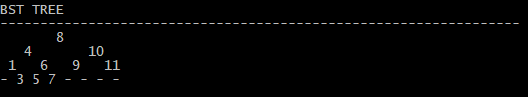




The third possibility is that if the node to be deleted has two children, the application will be required to find the successor through the rightmost child in the node’s left subtree. It will then copy its contents to replace the node’s item, before moving from its original position to node’s current position. However, a successor is only required when the right child of the node is not empty. Refer to the three screenshots below for an example.

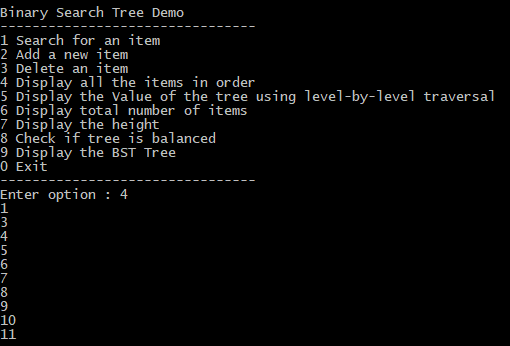






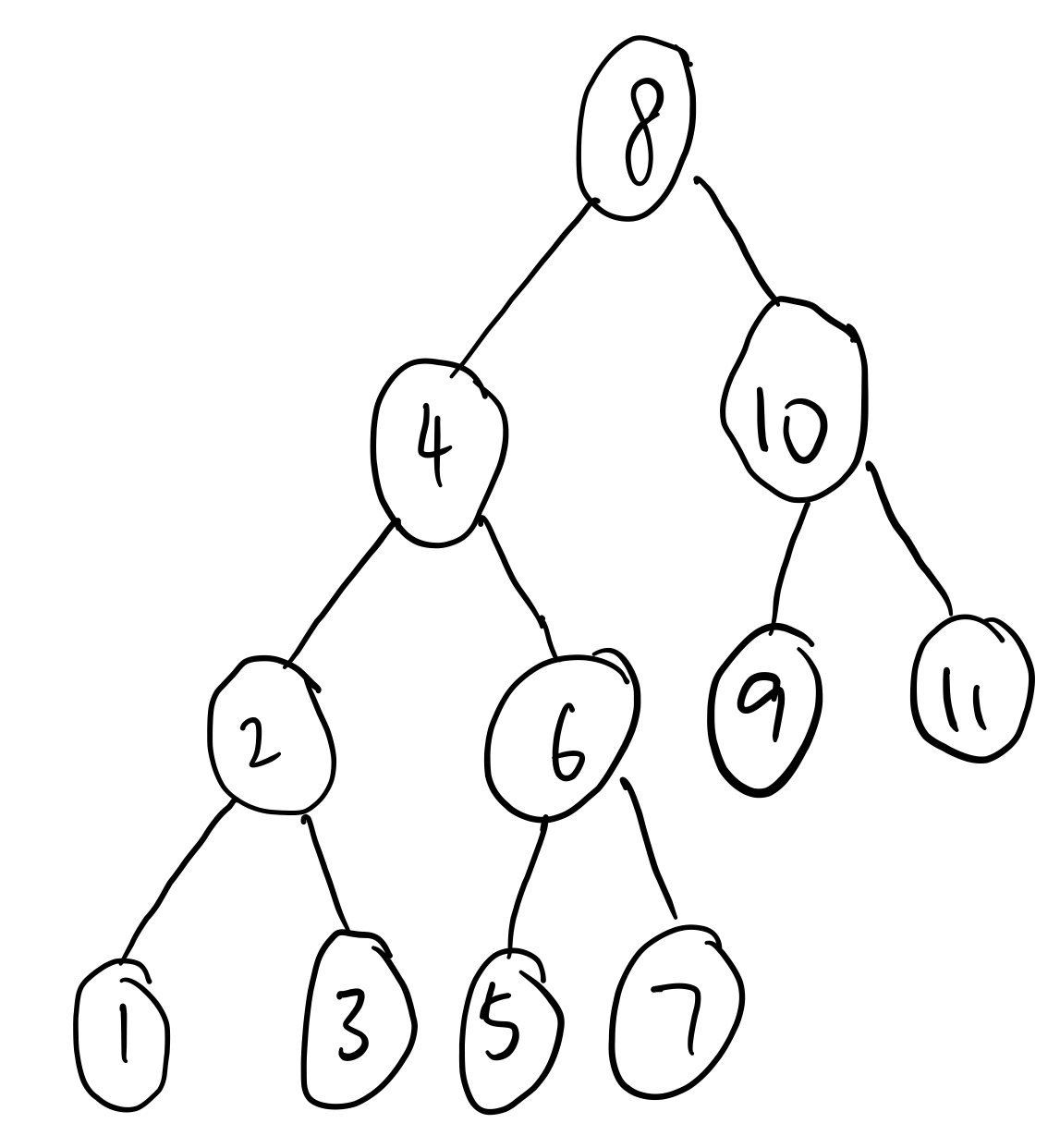
## **Option 4: Displaying of Values in Ascending Order**

Our fourth option allows the user to display the current values in the BST in ascending order, which means from the lowest to the highest value. The function used in the application is through inorder(), where it will traverse the BST in order through recursion to get the values in ascending order.

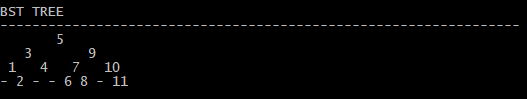


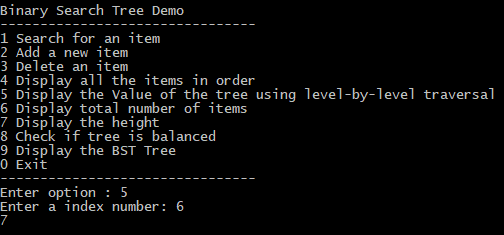
## **Option 5: Display value of kth node through Level-By-Level Traversal**

Our fifth option allows the user to search for an index in the BST to display the value that belongs to the index. As we are implementing the Queue Data Structure for this feature, there are five functions, levelOrder(), createQueue(), enQueue(), deQueue() and newNode(). The application will first create an empty queue q, where the temp\_node is the root. Starting from the root, the application will loop while temp\_node is not null. During each loop, the application will print temp\_node -> data, where it will then enqueue temp\_node’s children (left child then right child) to q. Lastly, it will dequeue a node from q and assign it’s value to temp\_node.



For example, referring to the illustration shown above, if the user entered an index number of 3, the application will return the value of ‘10’. This is because through the level order traversal, ‘10’ is the third index. Another example will be based off the two screenshots shown below, where if the user entered an index number of 6, it will return the value of 7.



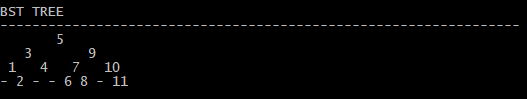


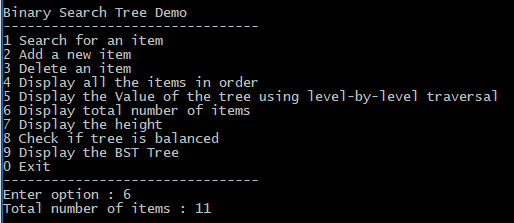
## **Option 6 & 7: Display Total Number of Items and Height**

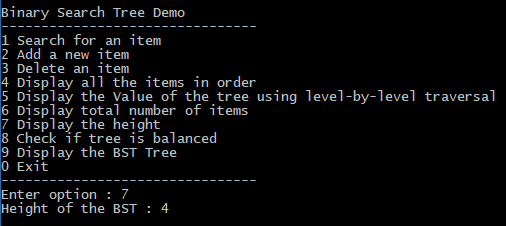
Our sixth and seventh option of the application helps to display the total number of items and height of the BST respectively. In the first screenshot shown below, it display the current state of the BST.

Firstly, the number of items is displayed through recursive function countNodes(), where it will count as it traverse through the tree.

Next, the height is displayed through the function getHeight() where it calculate the heights of the left subtree and right subtree, which will then calculate the maximum of the value of both heights, returning the value as h.





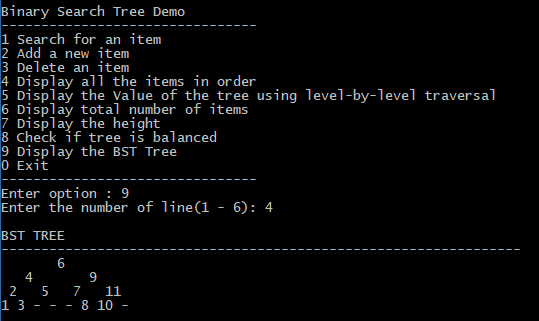


# **Additional Requirements**

The additional requirements of the application include displaying of BST in console, the BST being able to self-balance as well as the user validations that are implemented to ensure that the application is robust. The following are the explanations on each implementation of feature.

## **Option 8: Display the Tree**

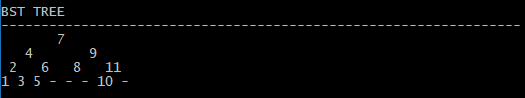
The BST is displayed in the console through two functions, printLevel() and printLevelOrder(), where the nodes in BST’s left and right subtrees are clearly shown. For example, in a scenario where BST contains values of ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘10’ and ‘11’, it will print the tree in the console that looks like this in the screenshot given below.

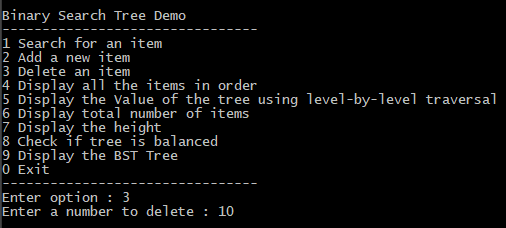


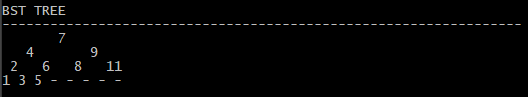
## **Option 2 & 3: Self-Balancing of Tree**

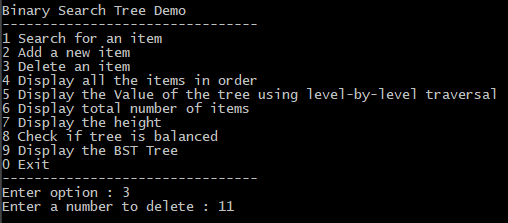
It is very important to have a balanced BST because the key reason to why a BST offers such great performance is because it allows us to ignore irrelevant values. Thus, by decreasing the number of comparisons a program has to perform to find a data element. This additional feature is implemented in Option 2 and 3, which are the adding of an item and deleting of an item respectively.

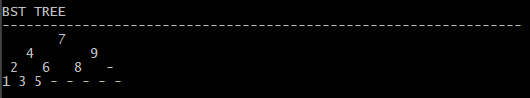
For example, referring to the screenshots given below, if the items ‘10’, ‘11’ and ‘8’ are removed from the BST, it will cause the tree to be unbalanced. However, because of this feature being implemented, the BST will rebalance itself.

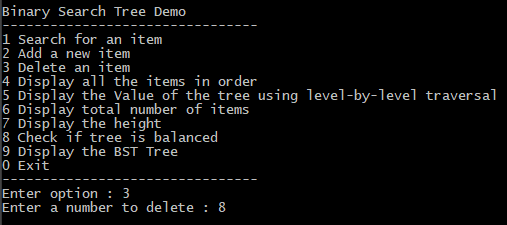


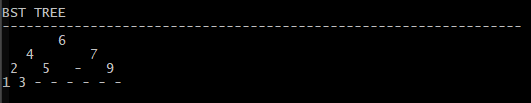












## **User Validation**

User validation is crucial in any application because it will also ensure that the application is robust. It also ensures that the user input is sensible and reasonable, and helps to prevent the application from crashing easily. For this application, we have validated a few user inputs. When the application console first start up, it will prompt the user to enter a number. A user validation is implemented where it ensures that the number must be ranged from 1 to 5050, where it must not be a decimal nor a negative value. Without this user validation in place, any negative or non-integer value input will cause the application to go on loop. Other user validations include the application menu, where the option number that the user enters must be an integer and in the range of 0 to 8. There are still plenty of validations that are found in the application, which helps to ensure that the user has an enjoyable crash-free experience with the application.

## **Robustness**

Robustness is essential in any application because no user will like to use an application that succumbs to crashing easily. Therefore, through various user validations, we ensure that this application has the ability to withstand adverse conditions and rigorous testing.

# **Data Structures and Algorithms Implemented**

In our application, three main data structures are implemented, such as Binary Search Tree, AVL Tree and Queue. The main reason as to why we used different data structures to implement different functions is because each one of them have their own pros and cons. As such, we made sure that we only used the advantages of each data structure to code this application.

## **Binary Search Tree**

BST stores item in nodes, so that searching, insertion and deletion of the nodes can be done efficiently and dynamically. As it uses simple implementation, all the functions can be done easily and quickly. The main feature of the BST is that it’s left subtree contains only values less than the root while the right subtree contains values greater than the root.

Therefore, when needed to find a specific element, it does not require to check the elements one by one - it will simply traverse the tree from root to leaf, to compare the items stored in the nodes to the element it is searching to continue its search in the left or right subtrees.

## **AVL Tree**

AVL tree is a self-balancing BST where the difference between heights of the left and right subtrees cannot be more than one for all the nodes in the tree. It is essential to implement AVL tree because most of the BST operations take O(*log n*) time where h is the height of the BST. The cost of these operations may become skewed for an unbalanced BST. This implementation is used in Option 2 and 3 of our application because the BST may be unbalanced each time a new item is added or removed from the tree, therefore there is a need for the BST to self-balance.

## **Queue**

Queue is a collection of items such that it is on a First In First Out (FIFO) basis. This is used such that the new items can only be added to the back of the queue and can only be removed from the front of the queue. This Data Structure is being implemented in Option 5 of our application. The queue in our application is implemented using an array with a maximum size of 500. The algorithm is that for each node, the node is first visited and it’s child nodes are placed in a First In First Out (FIFO) queue.

# **Class Diagram**

|  |
| --- |
| BST |
| - root: BinaryNode\* |
| +search(ItemType target): BinaryNode\*  -search(BinaryNode\* root, ItemType target): BinaryNode\*  +getHeight(): int  -getHeight(BinaryNode\* t): int  +diff(BinaryNode\* t): int  +rightright(BinaryNode\* parent): BinaryNode\*  +leftleft(BinaryNode\* parent): BinaryNode\*  +leftright(BinaryNode\* parent): BinaryNode\*  +rightleft(BinaryNode\* parent): BinaryNode\*  +rebalance(BinaryNode\* t): BinaryNode\*  +check(ItemType target): BinaryNode\*  -check(BinaryNode\* root, ItemType target): BinaryNode\*  +insert(ItemType target): void  -insert(BinaryNode\* &t, ItemType item): void  +remove(ItemType target): void  -remove(BinaryNode\* &root, ItemType target): void  +inorder(): void  -inorder(BinaryNode\* t): void  +preorder(): void  -preorder(BinaryNode\* t): void  +postorder(): void  -postorder(BinaryNode\* t): void  +countNodes(): int  -countNodes(BinaryNode\* t): int  +isEmpty(): bool  +levelOrder(int target): void  -levelOrder(BinaryNode\* n, int target): void  +createQueue(int \*front, int \*rear): BinaryNode\*\*  +enQueue(BinaryNode \*\*queue, int \*rear, BinaryNode \*new\_node): void  +deQueue(BinaryNode \*\*queue, int \*front): BinaryNode\*  +newNode(int data): BinaryNode\*  +printLevelOrder(int depth): void  -printLevelOrder(BinaryNode\* root, int depth): void |

1..1

1..\*

|  |
| --- |
| <interface>  BinaryNode |
| +item: int  +left: BinaryNode\*  +right: BinaryNode\* |

# **References**

[**https://www.geeksforgeeks.org/level-order-tree-traversal/**](https://www.geeksforgeeks.org/level-order-tree-traversal/)

[**https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/**](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)

[**https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/**](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)

[**https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/**](https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/)

[**https://www.crondose.com/2016/08/binary-search-trees-balanced/**](https://www.crondose.com/2016/08/binary-search-trees-balanced/)

[**http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html**](http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html)