Data Structures and Algorithm Assignment

Michael Lu Han Xien, Pierre Corazo Cesario

School of Engineering and Technology, Sunway University
Subang Jaya, Malaysia
[REDACTED]
[REDACTED]

Table of Contents

Abstract- Different data structures are widely used in computing depending on the application. This report goes over the basic ideas of how a binary search tree and Huffman code work. Then, the implementation of the two is detailed through explanation of the code and features used in Java. Figures such as examples, sample code and terminal outputs are included in the implementation section to help readers better understand each member's implementation. Limitations of each implementation were also included so that the code may be improved upon in the future. Each member's contribution towards the assignment and challenges faced were also detailed at the end.

I. Introduction

Computers can store, access and manipulate data in many different ways. However, this is dependent on which data structure is used to store the data, as each data structure has its advantages, disadvantages and algorithms which are suited to specific types of applications. For example, a hash table can be used for database indexing while Dijkstra's algorithm can be effectively used to find the shortest route in network routing protocols [1], [2].

For this assignment, CSC1203 students were tasked with implementing two algorithms and their respective data structures from a choice of seven, namely:

- Shortest Pathfinding
- Hashing
- Binary Search Tree
- Red-Black Tree
- Huffman Coding
- String Matching (except naïve)
- Dijkstra algorithm

With the help of the Java language, Binary Search Tree (BST) and Huffman Coding were implemented. As for Integrated Development Environments (IDE), BST was implemented using JetBrains' Intellij, and Huffman Coding was implemented using Microsoft's Visual Studio Code due to each group member's preferences.

A. Binary Search Tree

A BST is a specific type of binary tree with special rules. To understand its features and inner workings, the regular binary tree must first be explained.

A binary tree is, as its name suggests, a type of tree data structure, but only consists of two children per node. In each node, there are 3 elements which are the element or key, and L and R, which are pointers for the left and right node subtrees respectively [3]. These subtrees can be referred to as 'children' and the original node as 'parent'. Each node can have a minimum of no children, making it a ‘leaf node’, and a maximum of two children, hence the name binary tree.

It is when further constraints are added to a binary tree that it becomes a BST. The four added constraints are:

- The left subtree must only contain keys that are of lesser value than the parent node’s key.
- The right subtree must only contain keys that are of greater value than the parent node’s key.
- Duplicate key values are not allowed.
- Both left and right children must also be binary search trees.

Due to these special properties and its organisation, BSTs are much faster than regular binary trees at searching, insertion and deletion. In Big O notation, the average time complexity for searching, insertion and deletion is O(log n) and O(n) for average and worst case respectively. As a result, BSTs can be used in situations where elements must be stored in order or to find what elements are within a range. An interactive visual simulation of a BST can be found at the University of San Francisco’s website [4].

B. Huffman Coding

Huffman coding is a lossless data compression algorithm that functions by assigning variable length character codes to the input characters [5]. The Huffman algorithm is generally broken down into two major parts which involves [6]:

1. Creating a Huffman tree from the input character.
2. Traversing down the Huffman tree and assigning the character codes .

To obtain the best compression possible, the algorithm assigns the most frequently used character with the smallest character code whilst assigning the least frequently used character with a larger character code. To do so, the two major parts of the algorithm follow their own fundamental rule:

1. When creating the Huffman tree, the character with higher frequencies will be placed higher up in the tree while the characters with the lower frequencies will be placed lower down in the tree.
2. When traversing and assigning the character node, the algorithm begins to traverse down the tree from the root with an empty character node string and adds a new character to the character code string at every new depth.

The Huffman code algorithm also ensures that the generated character codes are prefix codes. Prefix codes is a coding system in which a character code is never equal to the prefix of another character code [7]. This special property helps ensure that there will be no difficulty in decoding a Huffman encoded message.

The results produced from the Huffman coding algorithm are a Huffman table mapping the original character code and the new Huffman character code, and the Huffman encoded message. The time complexity of the Huffman coding algorithm is O(nlogn) where n is the number of unique characters [6].

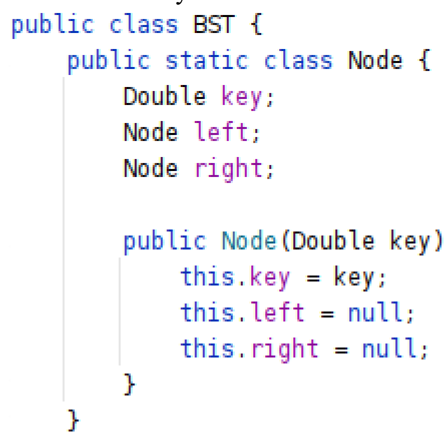
II. Binary Search Tree Implementation

The goal of implementing Binary Search Tree is to allow the user to perform insert, delete or search operations into a tree with their desired keys through the terminal. For all three of these operations, the terminal will print out how the algorithm traverses through the tree to execute the function. Then, once the user is satisfied with their modifications to the tree, they can choose to print it into the terminal in the form of preorder, inorder or postorder traversal functions.

To achieve this, a new class Node must be created with the appropriate attributes. As for functions, insert, delete, search, preorder, inorder and postorder functions are the main requirements. To facilitate user input, an appropriate function for taking in data and exception handling must also be created. Additional functions can be made to support the main functions’ functionalities and adhere to object-oriented programming (OOP) code reuse principles such as Don’t Repeat Yourself (DRY) [8].

A. Node Data Structure

As illustrated in Fig. 1, a new class *Node* was made as the data structure for the binary search tree.



```
public class BST {
    public static class Node {
        Double key;
        Node left;
        Node right;

        public Node(Double key) {
            this.key = key;
            this.left = null;
            this.right = null;
        }
    }
```

Fig. 1. Node class data structure with constructor.

It contains three attributes which are key, left and right. Key uses the Double data type to allow users to input decimals if they so desire, while left and right are of the Node class since they are essentially pointers to the children nodes. In other words, no objects except the *root* node are initialised with proper names. Rather, the program relies on pointers and memory locations to other Nodes within the tree.

To facilitate the creation of new nodes for insertion, an appropriate Node constructor class was made. The desired Double *key* is taken as an argument and the *Nodes* left and right are initialised as null. Then, a new null object of name *root* was initialised as the start or root of the tree.

B. Tree Operation Functions

For each of the operation functions, two functions were made:

1. The function to be called.

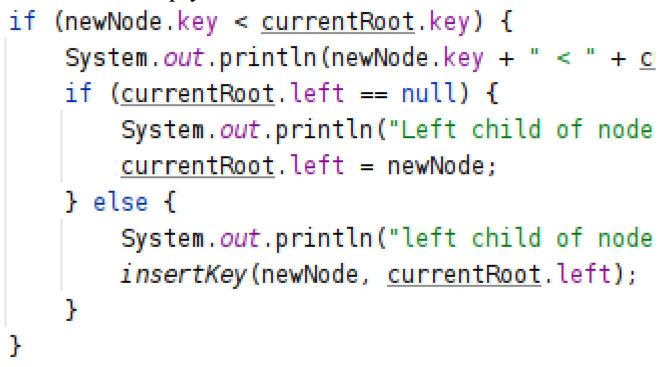
2. The main function that operates and recurs on the tree itself.

The function that does not contain the logic shall be referred to as the 'operation handler'. By splitting the functions into two different parts, the code becomes more modularised and brings some benefits. For example, by letting the *insert()* operation handler call the *insertKey()* function internally, only the desired *key* to be inserted is required as an argument, unlike in *insertKey()* where both the key and currently accessed node are required. Furthermore, it is paramount that the main operation function is separated since it is a recursion function, meaning the currently accessed Node constantly changes on every recursion. In the insert and delete operation functions, the return value of the recursion functions is assigned to the Node *root* so that it is updated with the modified tree. Another benefit is that it helps with the formatting of terminal output since it only executes once. In essence, by separating the two functions, code is more readable and the main recursion function is isolated.

Additionally, in all operations, appropriate print statements are included to inform the user on what is happening within the tree as it traverses and modifies it.

1) Insert Operation: insertKey() is the main recursion function. There are two outcomes to this operation, either the desired key does get inserted, or a duplicate is found and the tree is not modified.

Once the arguments have been passed through, the function will navigate through the given *root* node. If the new Node's key is smaller than the currently accessed Node's key, two more options are given. If the currently accessed Node's left child is null, meaning there is no child node, then it will assign the new Node to that position. However, if there is a node in the left child, then the function recurs by calling the *insertKey()* function with the same desired Node to be inserted but with the currently accessed Node's left child as the currently accessed Node. The same concept is applied if the new Node's key is greater than the currently accessed Node's key, but is flipped to navigate through the right children. In other words, the function continuously recurs until it finds a Node that is empty to insert the new Node.



```
if (newNode.key < currentRoot.key) {
    System.out.println(newNode.key + " < " + c
    if (currentRoot.left == null) {
        System.out.println("Left child of node
        currentRoot.left = newNode;
    } else {
        System.out.println("left child of node
        insertKey(newNode, currentRoot.left);
    }
}
```

Fig. 2. Lesser than logic for insertion operation.

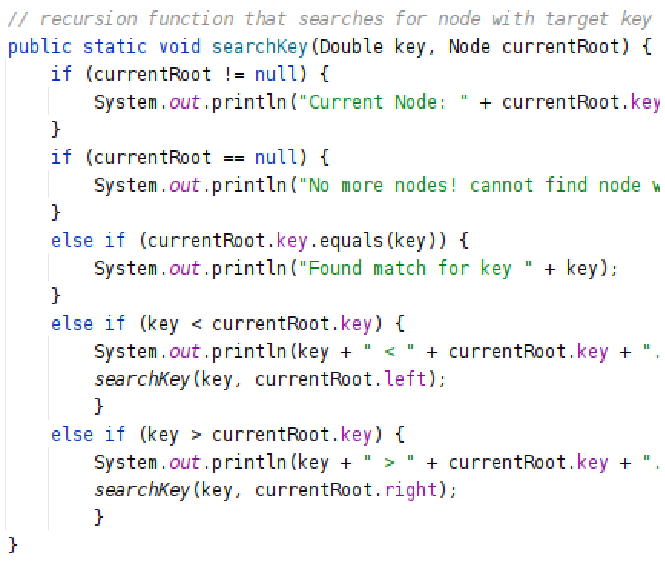
As it traverses through the BST, if there happens to be a duplicate of the Node's key to be inserted, then no changes are made to the tree in this scenario.

Furthermore, a control structure was also added in the event that it is a new tree and no Nodes have been inserted.

The operation handler *insert()*, calls the *insertKey()* function. The two arguments passed to the *insertKey()* function are a new initialisation of a Node object with the desired Double *key*, and the Node *root*.

2) Search Operation: The main recursion function is named *searchKey()*. There are two outcomes to this operation, either the desired key is found, or it is not.

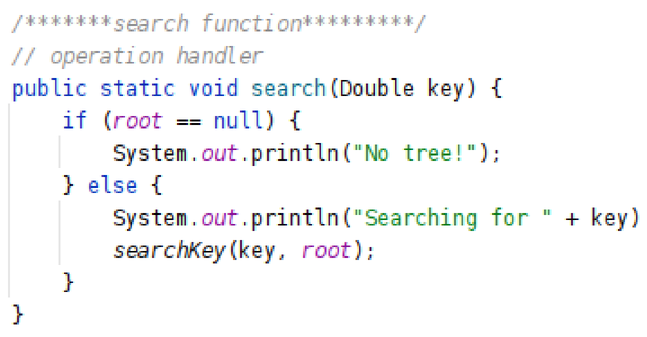
If the desired key is lesser than the currently accessed Node's key, then the operation function simply recurs with the same desired key but with the left child Node as the currently accessed node for the arguments. The same process is applied if the desired key is greater than the currently accessed Node's key, but is flipped to recur on the currently accessed Node's right child. This process continues until the currently accessed Node's key has the same value as the desired key, and a simple terminal message will be printed out. On the other hand, an appropriate terminal message will be printed if the desired key is not found.



```
// recursion function that searches for node with target key
public static void searchKey(Double key, Node currentRoot) {
    if (currentRoot != null) {
        System.out.println("Current Node: " + currentRoot.key
    }
    if (currentRoot == null) {
        System.out.println("No more nodes! cannot find node w
    }
    else if (currentRoot.key.equals(key)) {
        System.out.println("Found match for key " + key);
    }
    else if (key < currentRoot.key) {
        System.out.println(key + " < " + currentRoot.key + ".
        searchKey(key, currentRoot.left);
        }
    else if (key > currentRoot.key) {
        System.out.println(key + " > " + currentRoot.key + ".
        searchKey(key, currentRoot.right);
        }
}
```

Fig. 3. *searchKey()* recursion function.

As for the *search()* operation handler, an extra condition was added that checks if the *root* of the tree is null. If it is, the operation handler will not proceed with the *searchKey()* recursion function, thus saving time. This is an example of the benefit of splitting the operation function into two functions. However, if the tree is not empty, then it will call the *searchKey()* recursion function passing the arguments Double *key* and Node *root*.



```
/*******search function*********/
// operation handler
public static void search(Double key) {
    if (root == null) {
        System.out.println("No tree!");
    } else {
        System.out.println("Searching for " + key)
        searchKey(key, root);
    }
}
```

Fig. 4. *search()* operation handler.

3) Delete Operation: The delete operation is arguably the most complicated part when implementing a BST since there are many scenarios that can occur and must be properly handled. Before the main recursion function can be discussed, there are two prerequisite functions that must be made: one that checks if the currently accessed node is a leaf node, and one that looks for the Node with the minimum key value in the currently accessed Node's right subtree. To understand why these two extra functions must be made, the three possible cases when deleting a Node must first be understood.

In the first scenario, the target Node has no children, making it a leaf node. Here, no extra changes are made to the tree and the target node can simply be deleted, or assigned a null value. This scenario is why an *isLeaf()* function is required.

In the second scenario, the target Node only has one child. Here, when the target Node is deleted, the child, and by extension its own children, takes its place. In other words, the target node is replaced with its child.

In the third scenario, the target node has two children. Here, it must be decided which child takes the target Node's place in the tree. There are two options:

1. Replace the target Node with the Node having the minimum value in the right subtree.
2. Replace the target Node with the Node having the maximum value in the left subtree.

In both options, only the Node is taken to replace the target Node - its children are not taken. Both approaches are valid, but for this implementation, option 1 was chosen. As a result, this scenario is why a *minimumRight()* function is required. Scenario 3's logic is demonstrated through Fig. 5 and Fig. 6.

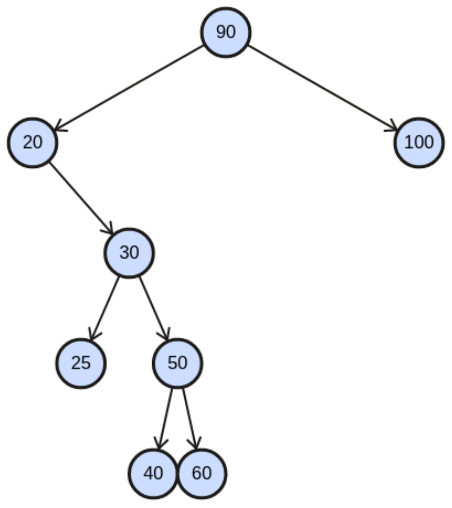


Fig. 5. Before deletion of key 30.

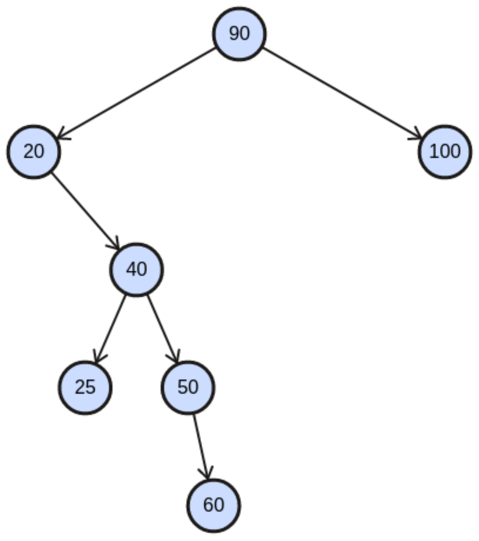
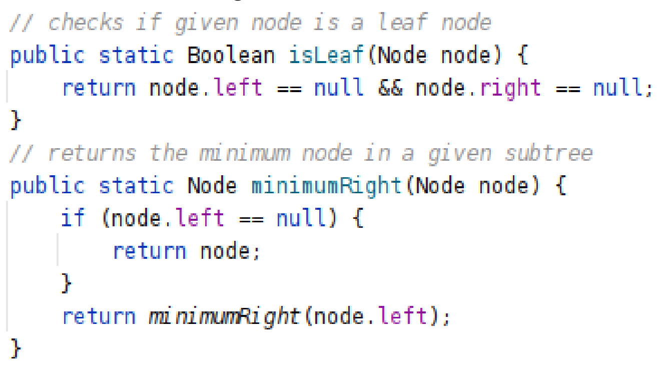


Fig. 6. After deletion of key 30.

A function of name *isLeaf()* with return type Boolean takes in a single argument *node* of the Node class. It simply checks if the passed in Node's left and right Node have null values using AND gate logic.

Additionally, a recursion function named *minimumRight()* was also created. To determine the minimum value of a currently accessed Node, the properties of a BST can be exploited. Since lesser values are assigned to the left subtree and all subtrees are also BSTs, the minimum value of a tree can be found by continuously recurring through the left subtrees. This is because the values will only get smaller as the tree is recurred on the left side. Thus, a simple control structure can be made that recurs until the minimum value has been reached. That Node's memory location can be returned for the main recursion function to use. If option 2 was chosen, the same concept can be applied to find the maximum value in the left subtree, except it needs to check if the right node is null and recur on the right nodes.

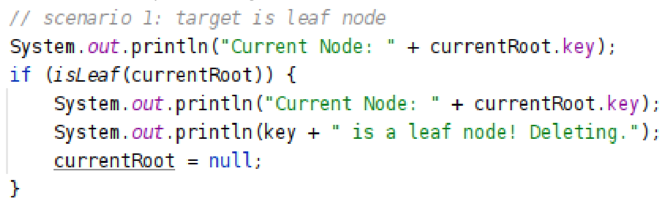


```
// checks if given node is a leaf node
public static Boolean isLeaf(Node node) {
    return node.left == null && node.right == null;
}
// returns the minimum node in a given subtree
public static Node minimumRight(Node node) {
    if (node.left == null) {
        return node;
    }
    return minimumRight(node.left);
}
```

Fig. 7. *isLeaf()* and *minimumRight()* functions.

Like the search operation, the *deleteNode()* recursion function must first recur down the tree until the target key is found. To recur, two control structures were made. If the target key is lesser than the currently accessed node's key value, then the function recurs using the same target key, but with the left node as the currently accessed node. However, it is important to note that if scenario 2 or 3 occurs, then that left node may change after deletion. Hence, the return of *deleteNode()* is assigned to the currently accessed node's left node. This is unlike the insert and search operations where the recursion functions are just called normally. Once the target node has been reached, one of the three aforementioned scenarios are determined using an if-else control structure.

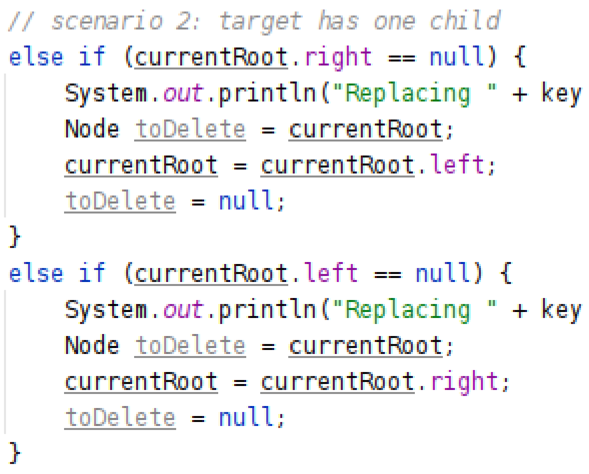
For the first scenario, it is determined using the *isLeaf()* function. If it is a leaf, the currently accessed node is assigned null, effectively deleting it.



```
// scenario 1: target is leaf node
System.out.println("Current Node: " + currentRoot.key);
if (isLeaf(currentRoot)) {
    System.out.println("Current Node: " + currentRoot.key);
    System.out.println(key + " is a leaf node! Deleting.");
    currentRoot = null;
}
```

Fig. 8. Scenario 1 code block..

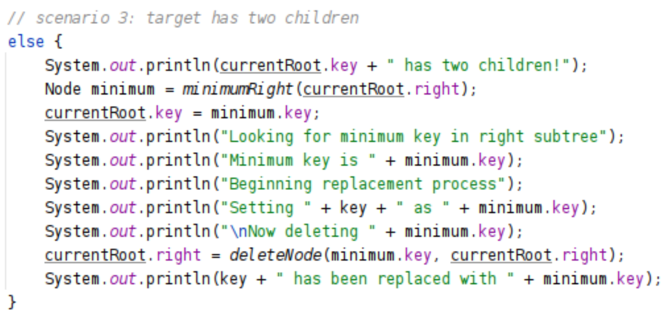
For the second scenario where there is only one child for the target node, the target node gets assigned to a temporarily initialised node *toDelete.* Then, the right child is assigned to the currently accessed node's location. Finally, *toDelete* is given the value null, thus deleting it. Should the situation be where the right child is null, the same concept can be applied but the swap is done with the left child instead.



```
// scenario 2: target has one child
else if (currentRoot.right == null) {
    System.out.println("Replacing " + key
    Node toDelete = currentRoot;
    currentRoot = currentRoot.left;
    toDelete = null;
}
else if (currentRoot.left == null) {
    System.out.println("Replacing " + key
    Node toDelete = currentRoot;
    currentRoot = currentRoot.right;
    toDelete = null;
}
```

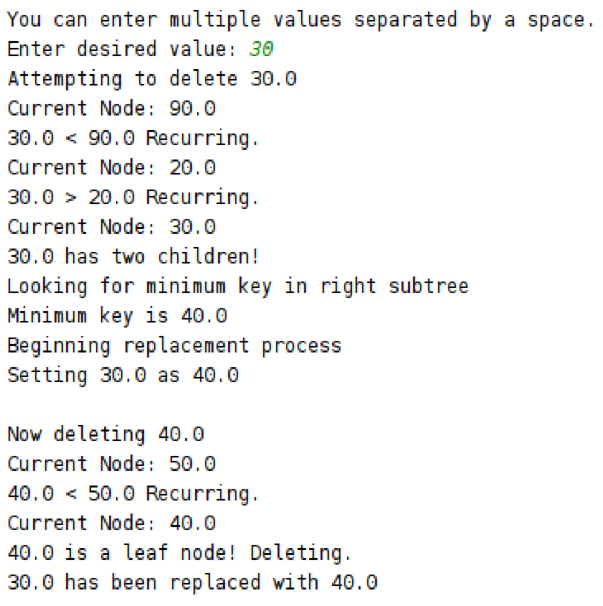
Fig. 9. Scenario 2 code block.

For the third scenario, the approach taken is to simplify it into either the first or second scenarios. The currently accessed node's key is assigned the key of the node that has the minimum value in the right subtree. Then *deleteNode()* is recurred using the arguments minimum key and right child node. This will then reduce the approach to one of the two other scenarios.



```
// scenario 3: target has two children
else {
    System.out.println(currentRoot.key + " has two children!");
    Node minimum = minimumRight(currentRoot.right);
    currentRoot.key = minimum.key;
    System.out.println("Looking for minimum key in right subtree");
    System.out.println("Minimum key is " + minimum.key);
    System.out.println("Beginning replacement process");
    System.out.println("Setting " + key + " as " + minimum.key);
    System.out.println("\nNow deleting " + minimum.key);
    currentRoot.right = deleteNode(minimum.key, currentRoot.right);
    System.out.println(key + " has been replaced with " + minimum.key);
}
```

Fig. 10. Scenario 3 code block.



```
You can enter multiple values separated by a space.
Enter desired value: 30
Attempting to delete 30.0
Current Node: 90.0
30.0 < 90.0 Recurring.
Current Node: 20.0
30.0 > 20.0 Recurring.
Current Node: 30.0
30.0 has two children!
Looking for minimum key in right subtree
Minimum key is 40.0
Beginning replacement process
Setting 30.0 as 40.0

Now deleting 40.0
Current Node: 50.0
40.0 < 50.0 Recurring.
Current Node: 40.0
40.0 is a leaf node! Deleting.
30.0 has been replaced with 40.0
```

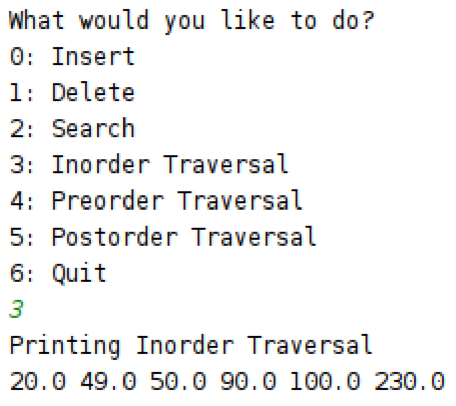
Fig. 11. Scenario 3 sample terminal output.

In the event that the target key is not found, the tree will not be modified.

As for the operation handler *delete(),* it is essentially identical to the *search()* function but employs the *deleteNode()* recursion function instead.

C. Traversal Functions

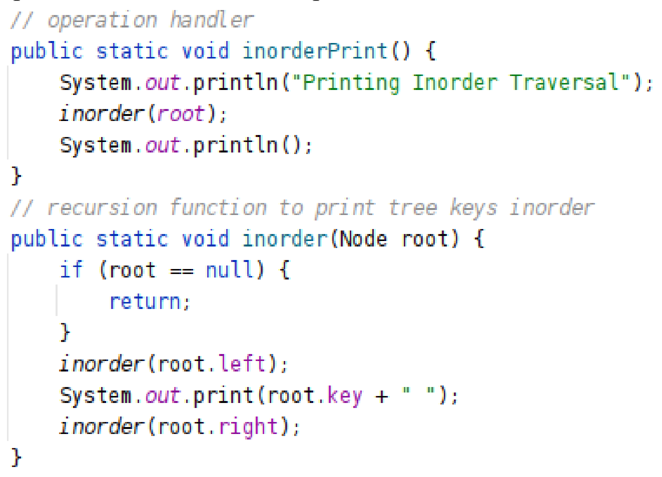
Three traversal functions were made to demonstrate the different ways the BST can be sorted. These three are inorder, postorder and preorder traversal. Similar to the tree operation functions, these traversal functions were split into two separate functions for the same reasons and benefits.



```
What would you like to do?
0: Insert
1: Delete
2: Search
3: Inorder Traversal
4: Preorder Traversal
5: Postorder Traversal
6: Quit
3
Printing Inorder Traversal
20.0 49.0 50.0 90.0 100.0 230.0
```

Fig. 12. Sample inorder traversal terminal output.

All three traversal functions were achieved through recursion by rearranging the sequence of calling the recurring function and printing the actual node's key, depending on the chosen traversal function. Once the recursion has reached a null node, the last node with a key is printed. This process repeats until the whole tree is printed.



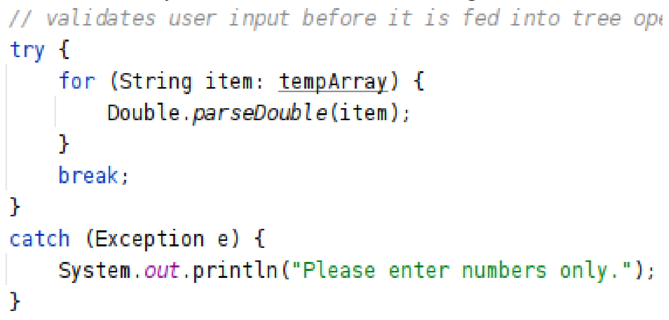
```java
// operation handler
public static void inorderPrint() {
    System.out.println("Printing Inorder Traversal");
    inorder(root);
    System.out.println();
}
// recursion function to print tree keys inorder
public static void inorder(Node root) {
    if (root == null) {
        return;
    }
    inorder(root.left);
    System.out.print(root.key + " ");
    inorder(root.right);
}
```

Fig. 13. Inorder operation code.

D. Terminal Functions

To allow the user to choose between the tree operation and traversal functions, a switch case was made. 6 options were given - one for each function - and an extra option to close the program for a total of 7 choices.

Users can choose to input multiple Double values separated by a space when using one of the tree operation functions. Furthermore, a while loop together with a try-catch block was implemented to only accept valid Double inputs to prevent crashing the program. Inputs are first processed to check for validity before they are passed to the actual operation functions. This is seen through the *keyEntry()* function that is used in the switch case for all the operation functions. The try-catch block is shown in Fig. 12.



```java
// validates user input before it is fed into tree ope
try {
    for (String item: tempArray) {
        Double.parseDouble(item);
    }
    break;
}
catch (Exception e) {
    System.out.println("Please enter numbers only.");
}
```

Fig. 14. *keyEntry()* try-catch block to validate user input.

As for the traversal functions, the appropriate functions are called depending on the desired selection. There is no need for *keyEntry()* in this scenario.

The entire switch case block is encapsulated in a while loop to let the user continuously utilise the functions. Should the user wish to quit, one of the options will cease the while loop and close the program.

E. Limitations

As for limitations, the user is not able to quit their current activity. The user must first complete their desired action before they can revisit the menu. For example, if the user decides to insert a key, but later changes their mind and wishes to delete a key, they are unable to do so until they insert a key.

III. Huffman Code Implementation

A program was written to accept an input string and display the Huffman table, the size comparison with ASCII encoding, the input message in binary form, and the input message in the Huffman encoding.

A. Node Data Structures

To create the Huffman tree, two different node classes were made for slightly better space optimisation and easier identification.

The first node class is the *Node* class which will store general information. The class will contain four keys which are the huffmanCode, value, leftChild, and rightChild. Aside from the basic constructor, setter and getter method, the class will contain an additional method *generateHuffmanCode()* which will take in the character code from the parent *Node* then assign the character code to itself and pass new character codes to its left and right child *Node*s. The purpose of this class is to act as the non-leaf nodes in the Huffman tree.

The second node class is the *CharacterNode* class which is an extension of the regular *Node* class that stores an additional value which is the *character*. The purpose of this class is to act as the root node that contains the identity of which unique character it belongs to.

B. Generating Huffman Code

1) Creating Huffman Tree: The first step to creating the Huffman Tree is to process the input string to identify every unique character and their corresponding frequency in the input text. After processing the input text, the information is passed on and used in the second step.

The second step is to generate the leaf nodes which will be instances of the *CharacterNode* class. A *CharacterNode* object is made for each of the unique characters. The *value* attribute is set as the frequency of the character in the input text while the *character* attribute is set to the unique character it belongs to. The attribute *leftChild* and *rightChild* is set as *null* as the *CharacterNode*s are leaf nodes and will not have child nodes.

For the ease of accessing the leaf nodes later, instead of making just one list to store the leaf nodes, two lists were made.

1. The first list is an *ArrayList* of type *CharacterNode*. It is where the leaf nodes were created as *CharacterNode* instances and added in. This list will not be processed and will be used to hold the memory location of every leaf node. This list will be returned at the end of the Huffman tree creation process.
2. The second list is a *PriorityQueue* of type *Node*. A *PriorityQueue* is a default Java list that automatically sorts itself when new items are added in. This list will be used for processing to create the Huffman tree. Copies of the leaf nodes memory location are passed from the first list to create the duplicate list of leaf nodes.

Typically, after creating the list of leaf nodes (the second list, *PriorityQueue*), the third step is to sort the list of nodes by each node's *value*. Since the *PriorityQueue* already does this automatically when new nodes are added in, the code implementation for this process can be skipped.

After sorting, the third step is to extract the first two nodes (nodes with the smallest value) out of the *PriorityQueue* and combine them to make a new node. The process of this is as follows:

1. Extract the first two nodes out from the *PriorityQueue*
2. Create a new *Node* instance
    a. Attribute *value* is set to the sum of the two node's *value*
    b. Attribute *leftChild* is set as the first extracted node
    c. Attribute *rightChild* is set as the second extracted node
3. Add the new *Node* instances back to the *PriorityQueue*

The third step is repeated until the *PriorityQueue* is left with only one node. When only one node is left in the *PriorityQueue*, the Huffman tree is complete and could be used for traversal.

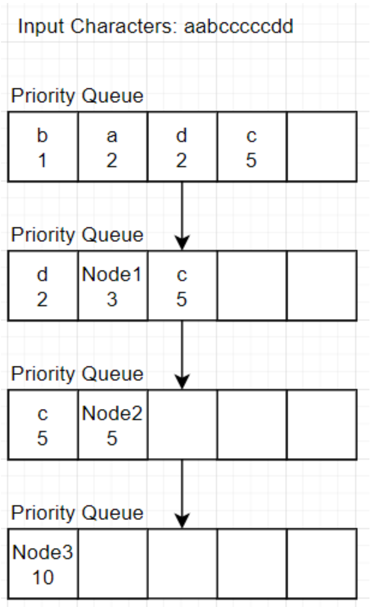


Fig. 15. Generating Huffman tree in *PriorityQueue.*

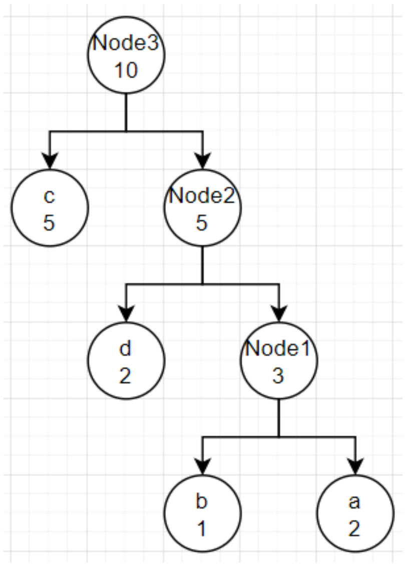


Fig. 16. Huffman tree visualisation.

2) Traversing Huffman Tree: After creating the Huffman tree, the tree will be used for traversal to generate the character code. To begin, an empty bit string is passed to the root of the Huffman tree. The root of the Huffman tree is also the only node left in the *PriorityQueue*. The bit string is recursively passed down to the child nodes until it reaches the leaf nodes. When a node receives a bit string, it will set its *huffmanCode* value as the bit string that was passed in. When passing the bit string down, a new bit character is added at the end of the bit string. The bit character added must be different for the left and right child and every node must follow the same pattern. Meaning, if the algorithm chooses to add a 1-bit for the left child and 0-bit for the right child, all nodes must also add a 1-bit for the left child and 0-bit for the right child. In this version of the Huffman code algorithm, a 0-bit was added to the end of the bit string for the left child while a 1-bit was added to the end of the bit string for the right child. The traversing ends when all nodes have obtained their corresponding bit string (character code). After generating the character codes for all unique characters, the Huffman table and encoded message can finally be created. The *ArrayList* of *CharacterNode* containing all leaf nodes can now be returned for further use.

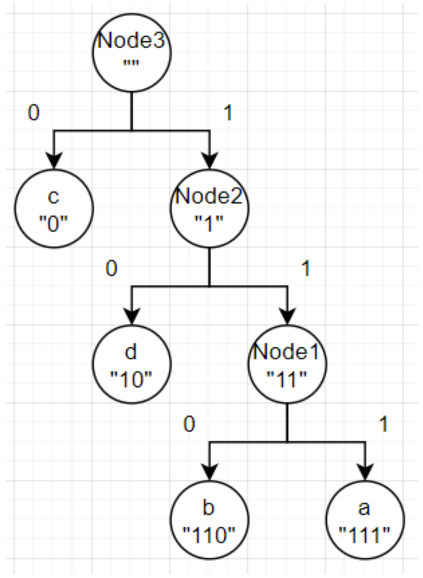
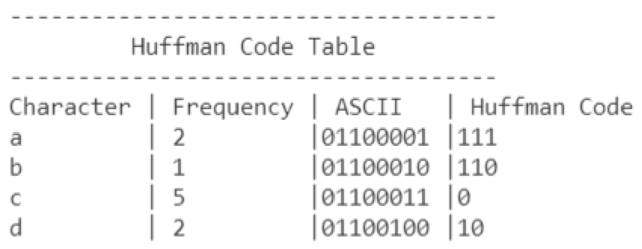


Fig. 17. Generating character code visualisation.

C. User Functions

1) Displaying Huffman Table: Using the *ArrayList* of *CharacterNode* generated, the unique character, character code, and frequency value of every character from the input text can be obtained. The character's ASCII binary character code can be obtained by first obtaining the byte character code using Java's default method *getBytes(Charset.forName("US-ASCII"))*, then converting it into binary.

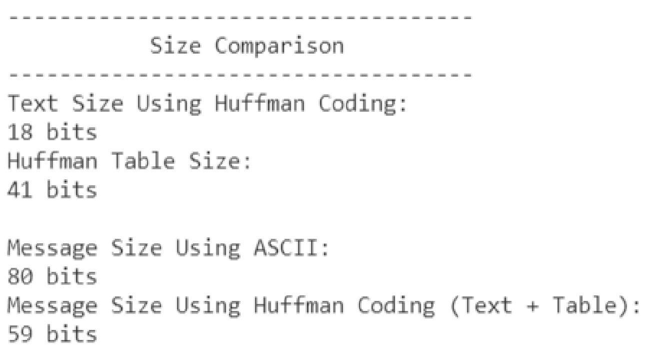


```
----------------------------------
        Huffman Code Table
----------------------------------
Character | Frequency | ASCII    | Huffman Code
a         | 2         |01100001 |111
b         | 1         |01100010 |110
c         | 5         |01100011 |0
d         | 2         |01100100 |10
```

Fig. 18. Result of Huffman table in terminal.

2) Displaying Size Comparison: In process, a few calculations need to be done with the following formula:

- ASCII text size = input text length * 8
  - o ASCII encoding codes every character in 8-bits character codes.
- Huffman encoded text size = Σ f_c * $code_c$
  - o Sum of every character's frequency value multiplied by its Huffman character code length.
  - o f_c = Frequency value of character
  - o $code_c$ = Character code length of character
- Huffman table size = Σ 8 + $code_c$
  - o Table usually only contains the ASCII and Huffman character code. Thus, the size of the table is the sum of every character's 8-bit ASCII character code and its Huffman character code length.
- Huffman encoded message size = Huffman table size + Huffman encoded text size
  - o Encoded message package passes the encoded message and the Huffman table along with it for decoding purposes. Thus, the message size is the sum of the encoded message size and Huffman table size.

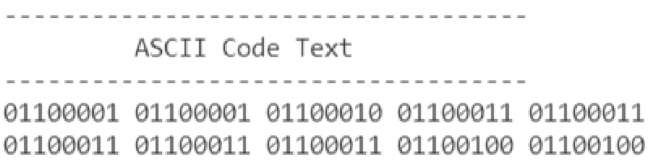


```
----------------------------------
          Size Comparison
----------------------------------
Text Size Using Huffman Coding:
18 bits
Huffman Table Size:
41 bits

Message Size Using ASCII:
80 bits
Message Size Using Huffman Coding (Text + Table):
59 bits
```

Fig. 19. Result of size comparison in terminal.

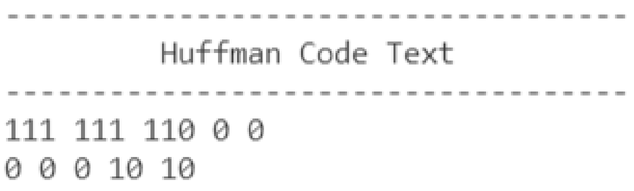
3) *Displaying Message in Binary Form*: The input string is first converted into a byte array by converting every character from the input string into bytes using Java's default method *getBytes(Charset.forName("US-ASCII"))*. After obtaining the byte array, iterate through every byte and convert the bytes into binary and print the binary result.



```
-----------------------------------
         ASCII Code Text
-----------------------------------
01100001 01100001 01100010 01100011 01100011
01100011 01100011 01100011 01100100 01100100
```

Fig. 20. Result of binary message in terminal.

4) Displaying Message in Huffman Encoding: Using the *ArrayList* of *CharacterNode* generated, the unique character, and the character code of every character from the input text can be obtained. These values are used to create a *Hashtable* (Java's equivalent of a dictionary). The program then iterates through every character of the input string and uses the hash table to map and print out the Huffman code characters.



```
-----------------------------------
         Huffman Code Text
-----------------------------------
111 111 110 0 0
0 0 0 10 10
```

Fig 21. Result of Huffman encoded message in terminal.

D. Limitations

In terms of limitations, the program can process and encode any user inputs except for an empty input and a reserve input text, "*exit*". The reserve input text is used as an identification key for users to quit the program if they wish to do so.

IV. Pierre's Contribution and Challenges

In terms of programming, Pierre was responsible for the implementation of BST. As for the report, he was tasked with writing the abstract, general introduction, Binary Search Tree explanation and implementation and conclusion sections. Furthermore, he provided guidelines on what to include in the report, ensured the report adhered to IEEE formatting and proofread content to ensure it satisfied the assignment's expectations. If any documentation issues were found, the team was actively updated so that it could be rectified quickly.

One challenge was the task of checking if the desired key value is the same as the currently accessed node, like in the *search()* and *insertKey()* functions. The approach taken was to use an '*==*' operator to compare keys. However, this approach was later found to be inaccurate since it would actually compare the memory locations of the two nodes, not their key values. The solution to this was to change the method of comparison into using the *.equals()* function. It is a more accurate approach since it will only compare the contents of two objects, not their memory locations.

Additionally, a challenge faced was in terms of optimising the code for user input. The same code block could be used for accepting user input for the tree operation functions, thus it was modularised into the *keyEntry()* function. However, this function needed to call a different tree operation function depending on the user's choice. The chosen solution was to add a control structure at the end of the *keyEntry()* function that changes which tree operation function is called depending on the argument given.

V. Michael's Contribution and Challenges

Michael contributed to the assignment by splitting and designating tasks to the members, created the Huffman encoding algorithm, and wrote the Huffman's implementation component for the introduction and implementation part of the report.

The first challenge faced when creating the Huffman coding program was how to access the leaf nodes without having to traverse down the root node every time they are needed. To solve this challenge, the idea of creating two lists and using the concept of subtype polymorphism was implemented (idea used in *1) Creating Huffman Tree*). In this idea, an *ArrayList* of type *CharacterNode* and a *PriorityQueue* of type *Node* were created to store the same leaf nodes. The leaf nodes were first created and stored in the *ArrayList* and then passed to the *PriorityQueue*. However, since the leaf nodes' data type differs, they need to be casted as an instance of *Node* using subtype polymorphism to suit the *ArrayList* type requirement. The reason why the second *ArrayList* needs to be in type *Node* is because new *Node* objects will be created and stored in the *ArrayList* during the creation of the Huffman tree. After the tree generation is complete, the leaf nodes can be easily accessed by utilising the *ArrayList*.

The second challenge faced when creating the Huffman encoding program was during the implementation of the *PriorityQueue*. Because the items inserted into the *PriorityQueue* are instances of the *Node* class, which have multiple attributes that can be used to compare. The *Node* class' *value* attribute needs to be specified to be used for comparison. To solve this challenge, a special property known as the comparator needs to be set for the *PriorityQueue* to compare the node's frequency value. This can be done by creating a new *Comparator* class, *NodeSorter*, where the values to be compared are set to the *Node's value* attribute. An instance of the *NodeSorter* can then be created and passed into the *PriorityQueue* constructor to create a *PriorityQueue* that compares a *Node's value* attribute.

VI. Conclusion

In conclusion, this assignment exposed the team to the concepts behind binary search tree and Huffman coding. Through hands-on experience with implementation, members were able to gather a more thorough and in-depth understanding on the chosen topics. For example, real world problems were encountered during implementation and had to be resolved, thus encouraging members to think critically about how to best utilise Java's features. Furthermore, members were able to learn about the other member's chosen algorithm and data structure. Finally, the assignment was a good opportunity for the team to work together not only in assisting each other for the implementation, but also for the creation of this report.

VII. References

[1] H. Garcia-Molina, J. Ullman and J. Widom, *Database system implementation*. Upper Saddle River, NJ: Prentice Hall, 2000.

[2] M. Pióro, Á. Szentesi, J. Harmatos, A. Jüttner, P. Gajowniczek and S. Kozdrowski, "On open shortest path first related network optimisation problems", *Performance Evaluation*, vol. 48, no. 1-4, pp. 201-223, 2002. Available: 10.1016/s0166-5316(02)00036-6.

[3] R. Garnier and J. Taylor, *Discrete Mathematics: Proofs, Structures and Applications*, 3rd ed. Boca Raton: CRC Press, 2010, p. 620.

[4] "Binary Search Tree Visualization", *cs.usfca.edu*, 2021. [Online]. Available: https://www.cs.usfca.edu/~galles/visualization/BST.html.

[5] D. A. Huffman, "A method for the construction of minimum-redundancy codes," Resonance, vol. 11, no. 2, pp. 91–99, 2006.

[6] "Huffman Coding: Greedy Algo-3," *GeeksforGeeks*, 23-Mar-2021. [Online]. Available: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. [Accessed: 06-Jun-2021].

[7] "Prefix Codes," *Encyclopedia.com*. [Online]. Available: https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/prefix-codes. [Accessed: 06-Jun-2021].

[8] S. Foote, *Learning to program*. Upper Saddle River, NJ: Addison-Wesley, 2015.