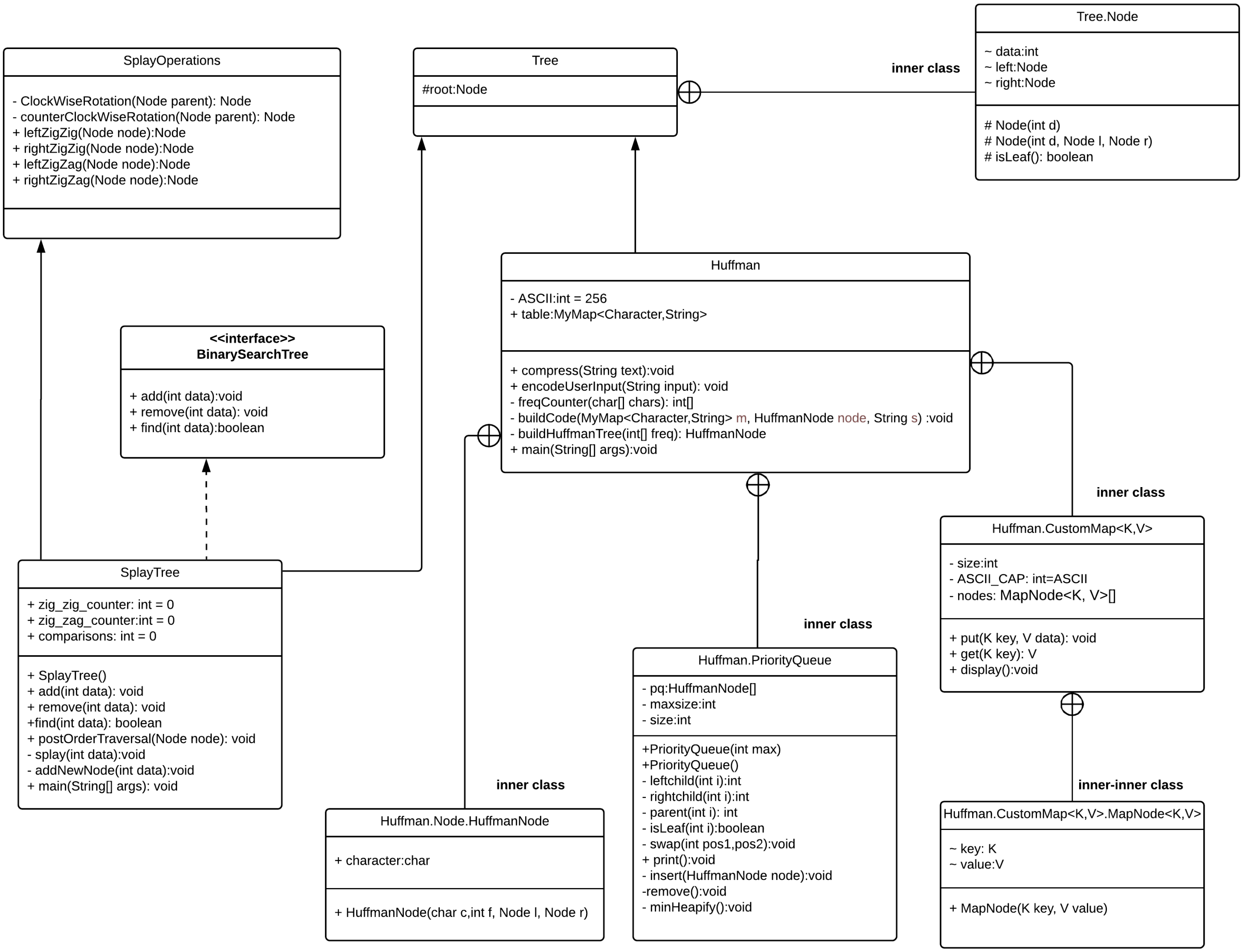
SplayOperations
- ClockWiseRotation(Node parent): Node
- counterClockWiseRotation(Node parent): Node
+ leftZigZig(Node node):Node
+ rightZigZig(Node node):Node
+ leftZigZag(Node node):Node
+ rightZigZag(Node node):Node
Tree
#root:Node
inner class
Tree.Node
~ data:int
~ left:Node
~ right:Node
# Node(int d)
# Node(int d, Node l, Node r)
# isLeaf(): boolean
<<interface>>
BinarySearchTree
+ add(int data):void
+ remove(int data): void
+ find(int data):boolean
Huffman
- ASCII:int = 256
+ table:MyMap<Character,String>
+ compress(String text):void
+ encodeUserInput(String input): void
- freqCounter(char[] chars): int[]
- buildCode(MyMap<Character,String> m, HuffmanNode node, String s) :void
- buildHuffmanTree(int[] freq): HuffmanNode
+ main(String[] args):void
inner class
Huffman.CustomMap<K,V>
- size:int
- ASCII_CAP: int=ASCII
- nodes: MapNode<K, V>[]
+ put(K key, V data): void
+ get(K key): V
+ display():void
SplayTree
+ zig_zig_counter: int = 0
+ zig_zag_counter:int = 0
+ comparisons: int = 0
+ SplayTree()
+ add(int data): void
+ remove(int data): void
+find(int data): boolean
+ postOrderTraversal(Node node): void
- splay(int data):void
- addNewNode(int data):void
+ main(String[] args): void
inner class
Huffman.PriorityQueue
- pq:HuffmanNode[]
- maxsize:int
- size:int
+PriorityQueue(int max)
+PriorityQueue()
- leftchild(int i):int
- rightchild(int i):int
- parent(int i): int
- isLeaf(int i):boolean
- swap(int pos1,pos2):void
+ print():void
- insert(HuffmanNode node):void
-remove():void
- minHeapify():void
inner class
Huffman.Node.HuffmanNode
+ character:char
+ HuffmanNode(char c,int f, Node l, Node r)
inner-inner class
Huffman.CustomMap<K,V>.MapNode<K,V>
~ key: K
~ value:V
+ MapNode(K key, V value)

(c)

I have created a Tree class that has node variable **root** and an inner Node class that contains the variables: **int data, Node left, and Node right**. The SplayTree class inherits and uses the node variable and inner Node class as is, while the HuffmanTree creates another inner class, “HuffmanNode”, that extends the inner class Node. I realized that a simple Tree structure as such would suffice given that an implementation of both Trees would work efficiently without using identical/similar methods and variables.

(e)

As for how the HuffmanTree code implementation extended the general structure:
The HuffmanTree class inherits the inner Node class from the Tree class. An inner HuffmanNode class that extends the inner Node class in Tree is then created within the HuffmanTree class. The new inner HuffmanNode class will hold an extra character variable.
I have also implemented 2 other inner classes, a custom priority queue and map.
The reason I added them as inner classes is because they won’t be used outside the HuffmanTree class. The hash map inner class adds another inner class called MapNode, which is used to hold the variables stored in the bins of the map array.

(g)

As for how the SplayTree code implementation extended the general structure:

The SplayTree class inherits from Tree class, SplayOperations class, and a BinarySearchTree interface. The Node Inner class in Tree won't be extended/altered and will be used as is. The SplayOperations class contains the implementations of methods such a **Zig-Zig and Zig-Zag** that are needed in the splay method. I decided to create a class for these operations to ensure code readability. Lastly, SplayTree implements a BinarySearchTree interface with the three main operations: **add, remove, and find.**

(h)

I wrote a program that counts the frequency of lookups on the elements in the Operations.txt file. Below are the top counts found in the text file.

```
Element:841 ->15 operations.
Element:964 ->15 operations.
Element:330 ->15 operations.
Element:679 ->16 operations.
Element:322 ->16 operations.
Element:785 ->16 operations.
Element:33 ->16 operations.
Element:983 ->16 operations.
Element:169 ->15 operations.
Element:493 ->15 operations.
Element:96 ->15 operations.
Element:736 ->15 operations.
Element:876 ->15 operations.
Element:741 ->16 operations.
Element:176 ->15 operations.
Element:146 ->15 operations.
Element:237 ->16 operations.
Element:821 ->15 operations.
Element:806 ->15 operations.
Element:937 ->15 operations.
Element:536 ->16 operations.
Element:252 ->15 operations.
```

Explain the advantages of using a Splay Tree over an AVL Tree to solve question 3:

The results above prove that certain elements in this program are looked up much more frequently than others. Since our application deals with a lot of data in the tree and will need access to a subset of the data frequently (certain elements more so than others), the Splay Tree would be a better choice.
In other words, the Splay method will make frequently accessed nodes stay near the top of the tree, resulting in reduced access cost that could be better than that of an AVL tree (When looking up frequently accessed nodes).