COMP 352: ASSIGNMENT 3

TATUM ALENKO (40055122)

1. *Designing a flexible tree structure.*

(a, b, c)

- *Design a tree structure that you will implement and subclass to solve both the Huffman Coding Problem and the Splay Tree problem. Include a class diagram showing all the supporting classes of this tree structure. Explain your decisions.*

- See last page of this document for the UML diagram.
- The goal was to design both the `Tree` and `TreeNode` super classes as abstract and extend them into their appropriate Huffman and Splay implementations (`Huffman, SplayTree`, and `HuffmanNode`, `SplayNode`, respectively).
- This tree super class contains a `size` attribute and a post-order traversal method (`postOrderTraverse`) common to all trees.
- Both extended trees compose their node counterparts as a `protected root` attribute.
- In some cases, `private` access visibility was employed as to encourage encapsulation-based design strategies (especially if the attributes or methods only served in the context of its own class, i.e. helper attributes or methods). Many methods were declared as `public` as they were designed for use potential use in the scope outside its package (especially auxiliary classes like `ArrayList`, `HashMap`, and `PriorityQueue`). However, in some situations, such as the nodes classes, the attributes were declared `protected` since they would need access within the tree subclasses (the nodes were designed as separate classes, not internal ones like in some traditional implementations). This was chosen since it was deemed sufficient for the application without more specification on how these classes would be used outside the context of this assignment. The use of getters and setters for the node attributes would have created unnecessary method overhead for situations where importing these classes as a package library. Regardless, the node attributes would be invisible to any class trying to access them out of package.

(d, e)

- *Include a class diagram showing how you Huffman Coding implementation will extend your general tree structure. Explain your decisions textually.*

- See last page of this document for the UML diagram.
- Use of a priority queue (`PriorityQueue`) instead of a simple array and re-sorting on each add/remove/find operation was used since it is more efficient (log$n$ in the worst case) as opposed to sorting an entire array (n, n^2, or $n$log$n$) in the worst case. This class is used as a helper data structure to aid in building the Huffman tree starting from the highest priority (lowest weight) nodes and reinserting the newly generated internal nodes as needed and re-polling them upon joining the nodes up until the root. The priority queue is able to assess the priority of its element through the overridden `compare` method of the `Comparator` interface implemented within the generic objects that it is composed of (in this case, the `HuffmanNode`).

- The `HashMap` employed was designed using a trivial ASCII character key map/table-based data structure. This class assumes a `Character` type based key, of which only 128 total possible values supply its entire size. These character keys then map to a specified index represented by its ASCII integer value (which is implemented in the `hash` method). The `HashMap` value entries are declared as a generic type for maximum flexibility.
- In both `PriorityQueue` and `HashMap` classes, the `ArrayList` helper class was used to store the generic objects (i.e. both compose `ArrayList` as an attribute of their class). The class, like the built-in Java class, takes care of automatically resizing the array of generic objects once its capacity is full.

(f, g)

- *Include a class diagram showing how your Splay Tree implementation will extend your general tree structure. Explain your decisions textually.*

- See last page of this document for the UML diagram.
- Unlike traditional splay tree algorithms, this one does not perform and extra 'zig' operation (if needed) to push the node up the root if it is just below it (i.e. a child of the root) when it ends up there. This has no practical use of course other than being cosmetically different from traditional implementations.
- The `splay` method is the core of the algorithm, it first detects the node parent and grandparent relationship to establish which double rotation is required to move the child node closer to the root level. Then, two consecutive single rotation methods (either `zigLeft` and/or `zigRight`) are called in order to perform said double rotation. A special conditional is provided within the `while` loop that performs the double rotations iteratively to ensure that if the node being splayed is either the root or a parent of the root (as indicated in the assignment handout) to break out of the loop. After this, the proper reference links that correspond to the rotation operation that are properly set and attached to the root (or as the root if it ends up there) appropriately.
- Special class counter variables are included to keep track of the number of comparisons between nodes (reference address comparison, not data values, or null checks), zig-zig, and zig-zag operations involved in inserting, searching, and removing a node; they are named `comparisonCount`, `zigzigCount`, and `zigzagCount`, respectively.

- **(h)** *Explain the advantages of using a Splay Tree over an AVL Tree to solve question 3, referencing your diagrams and decisions as is appropriate.*

- See 4. (b).

4. *Textual responses.*

(a) *Huffman.*

- *Encode the string associated with your student id and record the result here.*

```
she advised him to come back at once.
1110010001111001001101010101001110111110011111010100100011011010110010110110001010000110010111111
000111110011010000101011001001101100011011010101000111111001011
```

- *Describe what percentage fewer bits were used than the fixed-length ASCII encoding would have taken.*

- It would take $\log_2(34)$ or 6 bits to provide the 34 unique codes needed to represent the 34 characters in that string. This amounts to 6×37 or 222 total bits for fixed-length encoding. In contrast, using Jabberwock.txt text to create the Huffman encoding, the total number of bits required for the string given above would require only 160 total bits. This is a space savings of ~28%, i.e. Huffman encoding requires ~72% of the total number of bits compared to fixed-length encoding.

- *Indicate whether you feel that the encoded string matched the source text frequencies, and provide your reasoning.*

- Since there is a space reduction (less required total bits) to encode the string, this implies that the string does match (to some extent) the source text; otherwise, there would have been hardly any reduction compared to using fixed-length encoding, if any at all.

(b) *Advanced trees.*

- *Provide the output you recorded based on the Operations file provided in question 3.*

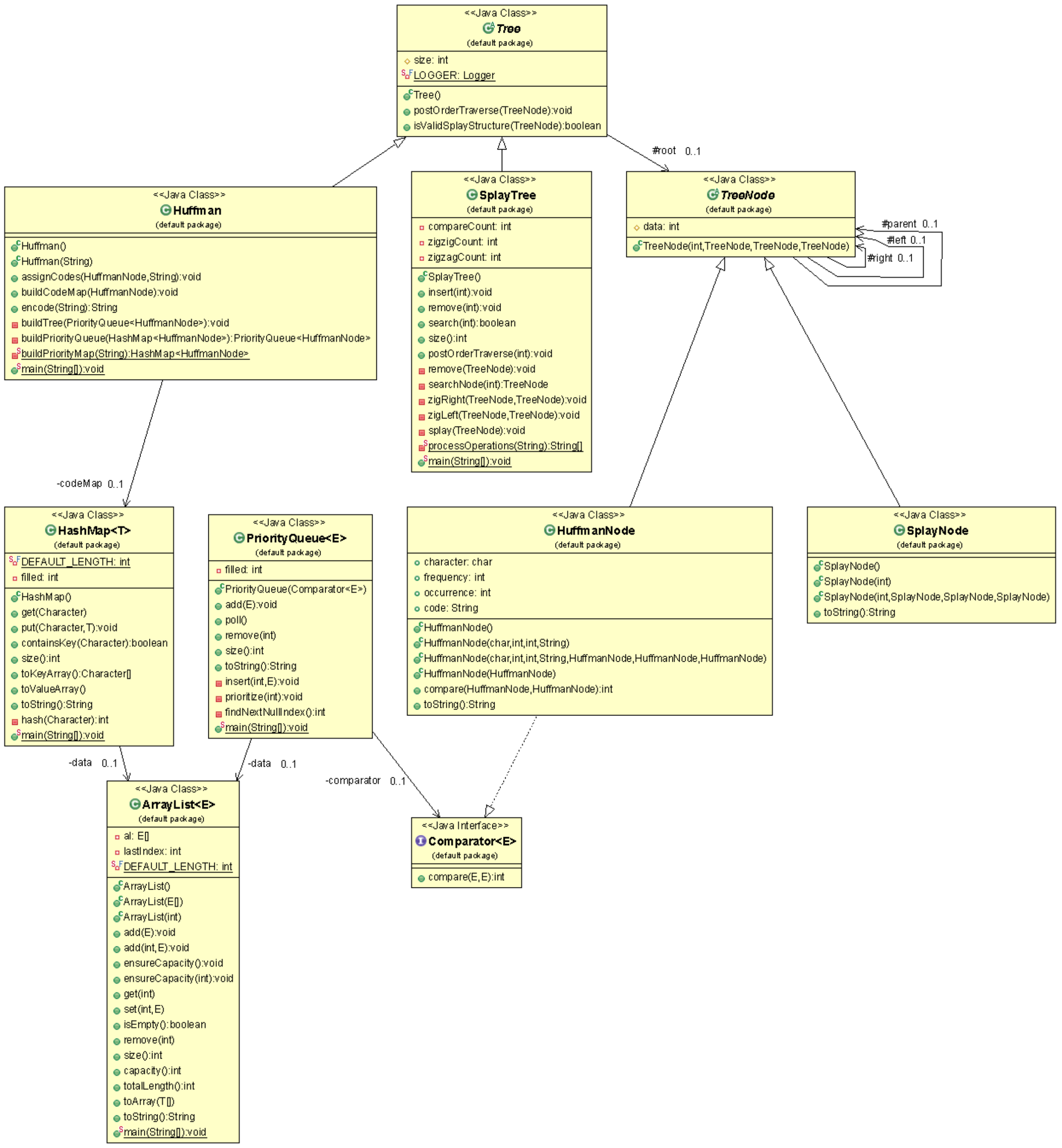
```
33780 compares
4487 Zig-Zigs
4289 Zig-Zags
```

- *Explain whether you feel that the Splay Tree was the correct choice, given that you have now implemented and tested it against the provided representative data. Explain anything you have learned through implementing and testing that would affect any future designs (for example, might you use an AVL tree).*

- As long as the same BST is being acted upon over a series of many operations (search, add, remove) are numerous enough (which in our case, the series of operations is), the guaranteed average cost of $k\log n$ using a splay implementation, where k is the number of operations, will justify its use over say a AVL based implementation. Although AVL implementations have the advantage of a guaranteed cost of $\log n$ for each operation, unless very few operations are performed on the same tree, a splay implementation has an equivalent, if not better, performance in the average multi-operation case (which, in Operations.txt, represents over 2000 operations, and hence approaching this limit for acceptable use of a splay tree algorithm).
- It should be noted, however, there is no 'one size fits all solution' to any problem. More context regarding the purpose of the design (and not simply one text file to mimic a scenario of use) would be necessary. It is important to note that splaying operations 'bubble up' recently operated on nodes, hence for a cache-based or like use case, splaying could be an obvious choice for frequent accesses to recently visited nodes.

COMP 352: ASSIGNMENT 3

TATUM ALENKO (40055122)



<<Java Class>>
Tree
(default package)
size: int
LOGGER: Logger
Tree()
postOrderTraverse(TreeNode):void
isValidSplayStructure(TreeNode):boolean
#root 0..1
<<Java Class>>
Huffman
(default package)
Huffman()
Huffman(String)
assignCodes(HuffmanNode,String):void
buildCodeMap(HuffmanNode):void
encode(String):String
buildTree(PriorityQueue<HuffmanNode>):void
buildPriorityQueue(HashMap<HuffmanNode>):PriorityQueue<HuffmanNode>
buildPriorityMap(String):HashMap<HuffmanNode>
main(String[]):void
<<Java Class>>
SplayTree
(default package)
compareCount: int
zigzigCount: int
zigzagCount: int
SplayTree()
insert(int):void
remove(int):void
search(int):boolean
size():int
postOrderTraverse(int):void
remove(TreeNode):void
searchNode(int):TreeNode
zigRight(TreeNode,TreeNode):void
zigLeft(TreeNode,TreeNode):void
splay(TreeNode):void
processOperations(String):String[]
main(String[]):void
<<Java Class>>
TreeNode
(default package)
data: int
TreeNode(int,TreeNode,TreeNode,TreeNode)
#parent 0..1
#left 0..1
#right 0..1
-codeMap 0..1
<<Java Class>>
HashMap<T>
(default package)
DEFAULT_LENGTH: int
filled: int
HashMap()
get(Character)
put(Character,T):void
containsKey(Character):boolean
size():int
toKeyArray():Character[]
toValueArray()
toString():String
hash(Character):int
main(String[]):void
<<Java Class>>
PriorityQueue<E>
(default package)
filled: int
PriorityQueue(Comparator<E>)
add(E):void
poll()
remove(int)
size():int
toString():String
insert(int,E):void
prioritize(int):void
findNextNullIndex():int
main(String[]):void
<<Java Class>>
HuffmanNode
(default package)
character: char
frequency: int
occurrence: int
code: String
HuffmanNode()
HuffmanNode(char,int,int,String)
HuffmanNode(char,int,int,String,HuffmanNode,HuffmanNode,HuffmanNode)
HuffmanNode(HuffmanNode)
compare(HuffmanNode,HuffmanNode):int
toString():String
<<Java Class>>
SplayNode
(default package)
SplayNode()
SplayNode(int)
SplayNode(int,SplayNode,SplayNode,SplayNode)
toString():String
-data 0..1
-data 0..1
-comparator 0..1
<<Java Class>>
ArrayList<E>
(default package)
al: E[]
lastIndex: int
DEFAULT_LENGTH: int
ArrayList()
ArrayList(E[])
ArrayList(int)
add(E):void
add(int,E):void
ensureCapacity():void
ensureCapacity(int):void
get(int)
set(int,E)
isEmpty():boolean
remove(int)
size():int
capacity():int
totalLength():int
toArray(T[])
toString():String
main(String[]):void
<<Java Interface>>
Comparator<E>
(default package)
compare(E,E):int