Huffman Coding

UNIVERSITY OF CALGARY

CPSC 319 - Data Structures

Benyamin Bashari

Java File I/O

- Extracting Frequency of Characters
- Building Huffman Tree
- Encoding a Text File
- Decoding a Binary File

Extracting Frequency of Characters

- This section focuses on extracting frequency of characters, when the input is a file.
- The first step is to use a proper data structure for maintaining the frequency of characters.
- One of the best data structures that can be used here is HashMap.

HashMap

- HashMap is a randomized data structure.
- Each entry in HashMap is a pair of <key, value>.
- Each entry in HashMap is accessible by the <key> value.
- Accessing element by <key>, adding pair of <key, value>, and removing a element by <key> takes O(1) in expected.

HashMap in Java

- **Class HashMap<K,V>**
  - **HashMap is Generic, K represent Key, and V represent Value**
  - `HashMap()`
    - Constructs an empty `HashMap`
  - `Void clear()`
    - Removes all of the mappings from this map.
  - `Boolean containsKey(Object key)`
    - Returns `true` if this map contains a mapping for the specified key.
  - `V get(Object key)`
    - Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
  - `V put(K key, V value)`
    - Associates the specified value with the specified key in this map.
    - Returns `the previous value associated with key, or null if there was no mapping for key`
  - `V remove(Object key)`
    - Removes the mapping for the specified key from this map if present.
    - `Returns the previous value associated with key, or null if there was no mapping for key`

Reading Input from a File

- Scanner class is used to read contents of the input file.

```
1.  public static String readFile(File inFile) throws FileNotFoundException {
2.      Scanner sc = new Scanner(inFile);
3.      StringBuilder inputText = new StringBuilder("");
4.      while(sc.hasNextLine()) { //Read the input line by line
5.          String line = sc.nextLine();
6.          inputText.append(line); //Add all of the lines together
7.          if(sc.hasNextLine()) //If this is not the last line
8.              inputText.append('\n'); //Between each two lines there is \n
9.      }
10.     return inputText.toString(); //Return the contents of the input file as a String
11. }
```

StringBuilder class works exactly like String class, with the difference that appending another string to the end is done efficiently.

Extracting Frequency of Characters

```
1.  /**
2.  * Increment the value of ch in the HashMap by 1
3.  */
4.  private static void incrementFreq(char ch, HashMap map) {
5.    if(!map.containsKey(ch)) //If ch is not in map
6.        map.put(ch, 0); //Make a new entry for ch with frequency 0
7.    Integer lastFreq = (Integer) map.get(ch);
8.    map.put(ch, lastFreq + 1);
9.  }
10. public static HashMap<Character, Integer> extractFrequnecy(String inputText)  {
11.   HashMap<Character, Integer> freqMap = new HashMap<>();
12.   for(int i = 0 ;  i < inputText.length() ; i++)
13.       incrementFreq(inputText.charAt((i)), freqMap);
14.   return freqMap;
15. }
```

Example

- Consider the following text as input

```
how can a clam cram in a clean cream can?
```

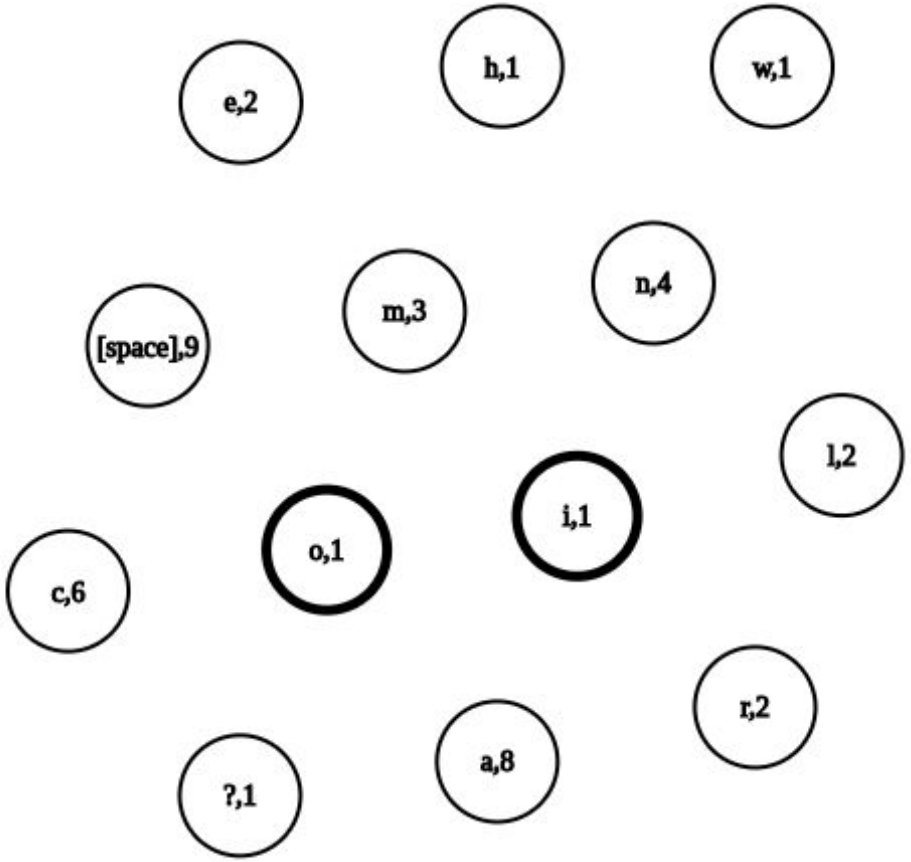
- Frequency of characters decreasing by their frequency (n = number of chars = 13)
  - [space] 9
  - a 8
  - c 6
  - n 4
  - m 3
  - r 2
  - l 2
  - e 2
  - ? 1
  - w 1
  - o 1
  - i 1
  - h 1

Building Huffman Tree

- In order to implement the Huffman Tree, it is required to consider each character with its frequency as a node (subtree) in a tree.
- At the beginning there are n subtrees without any edge.
- At each step, we require to find two nodes with the lowest frequency and merge them together to build a subtree.

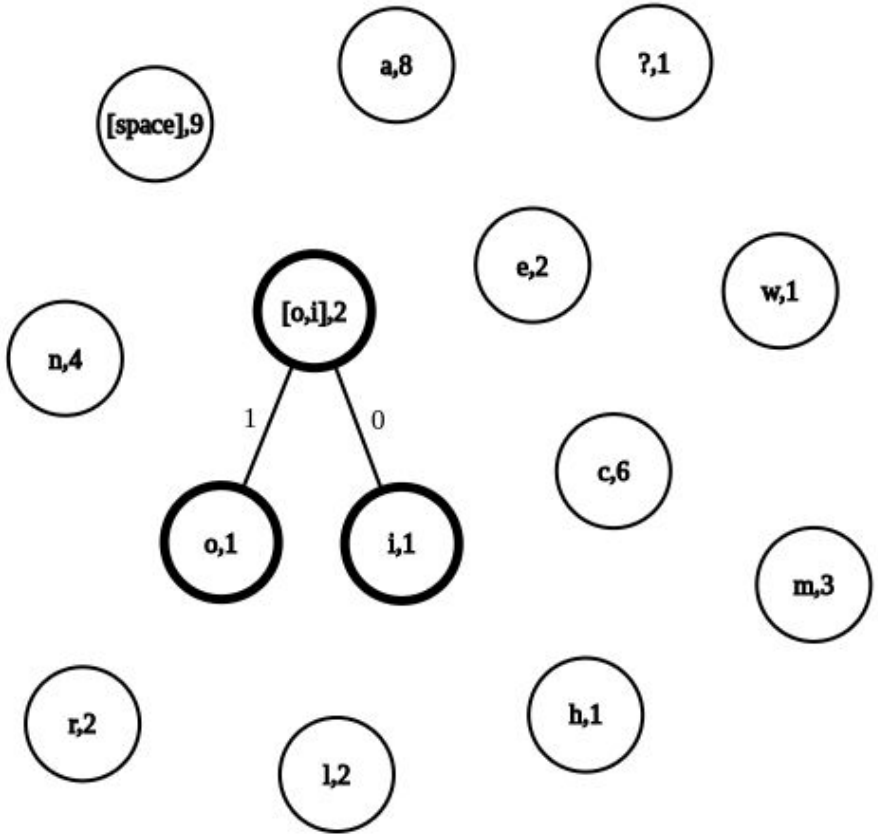
Example

- The following is the characters with their frequencies, two characters with the lowest frequency are determined.

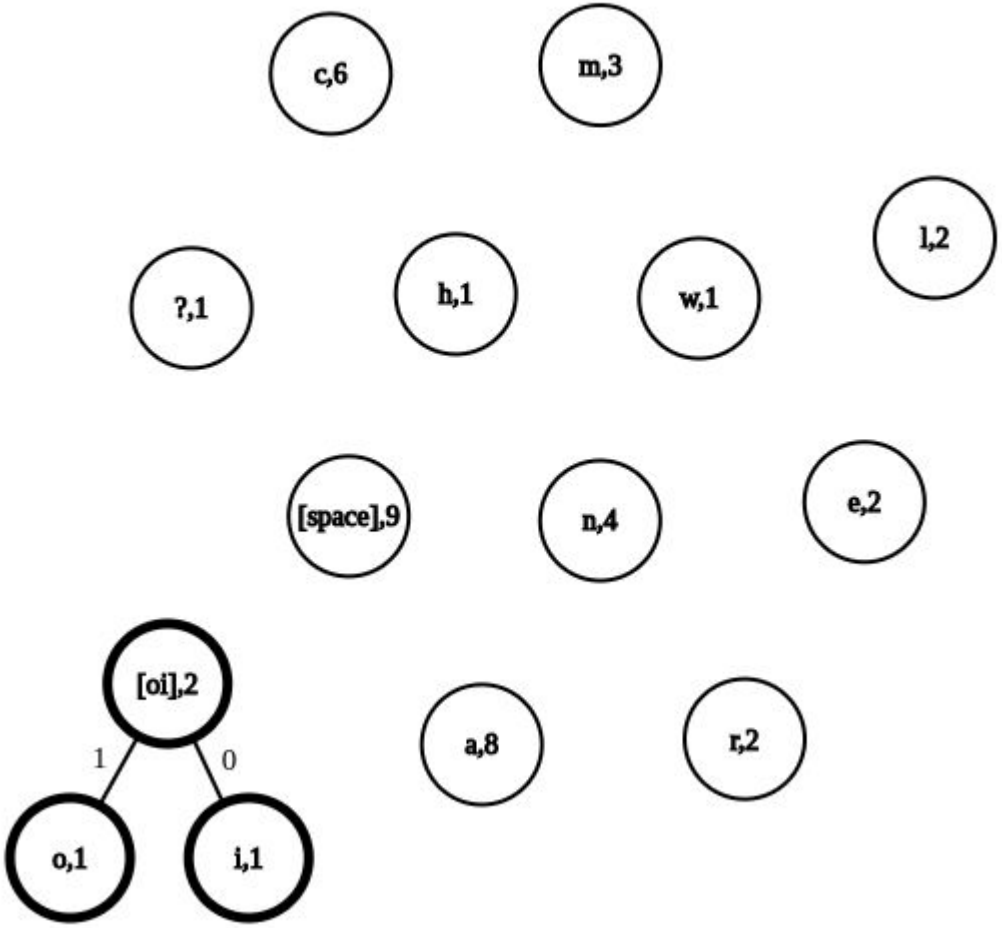


Example (Step 1)

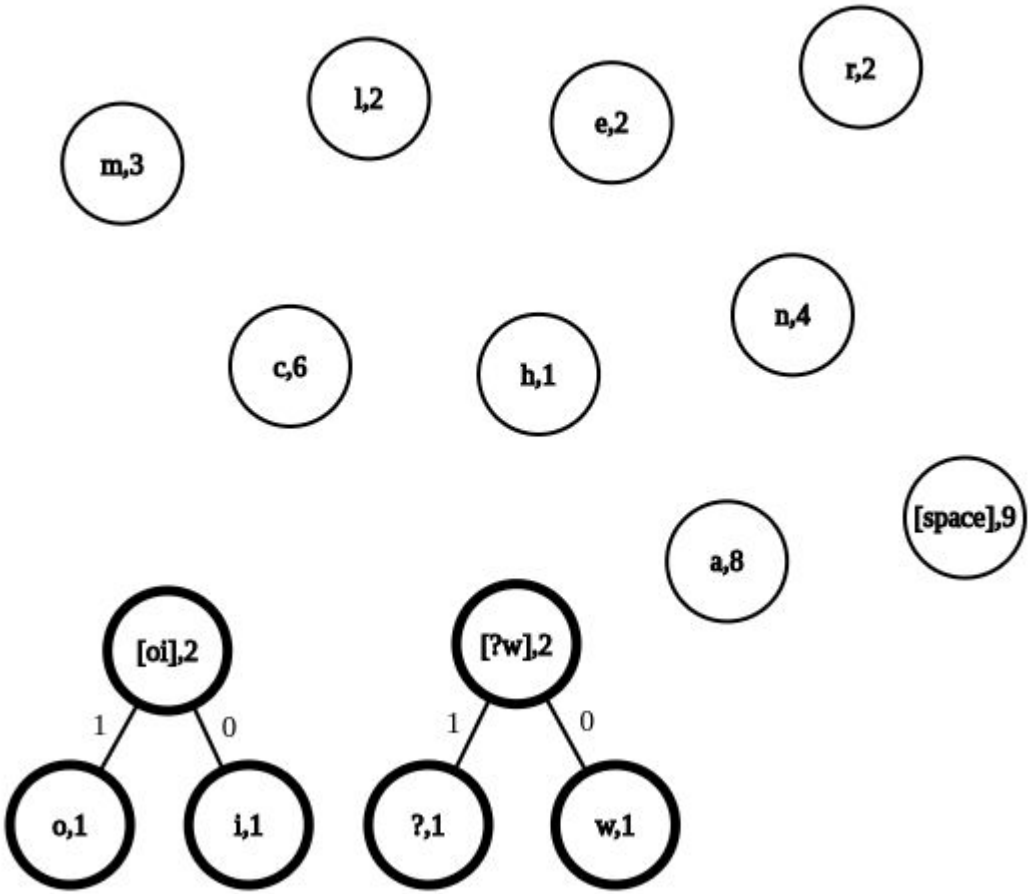
- Notice that the first step could be between any two nodes with the frequency of 1



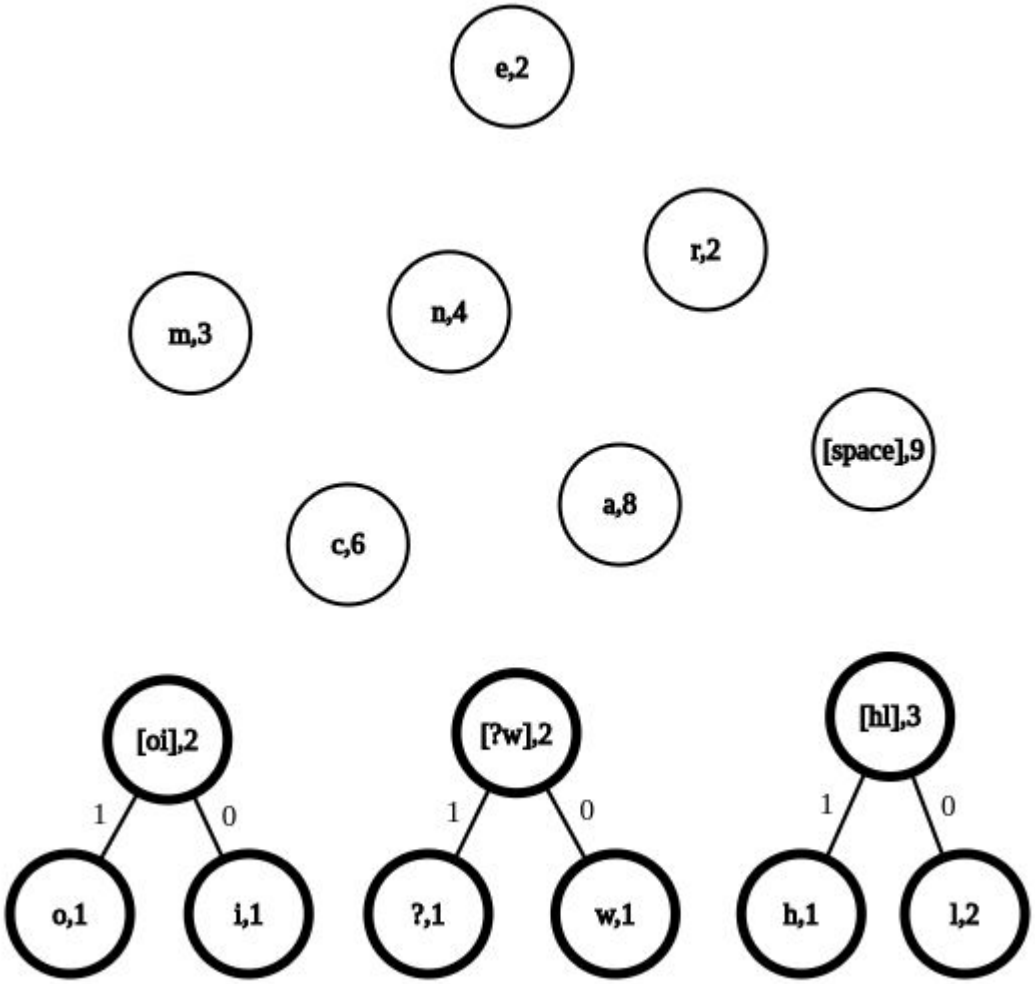
Example (Step 3)



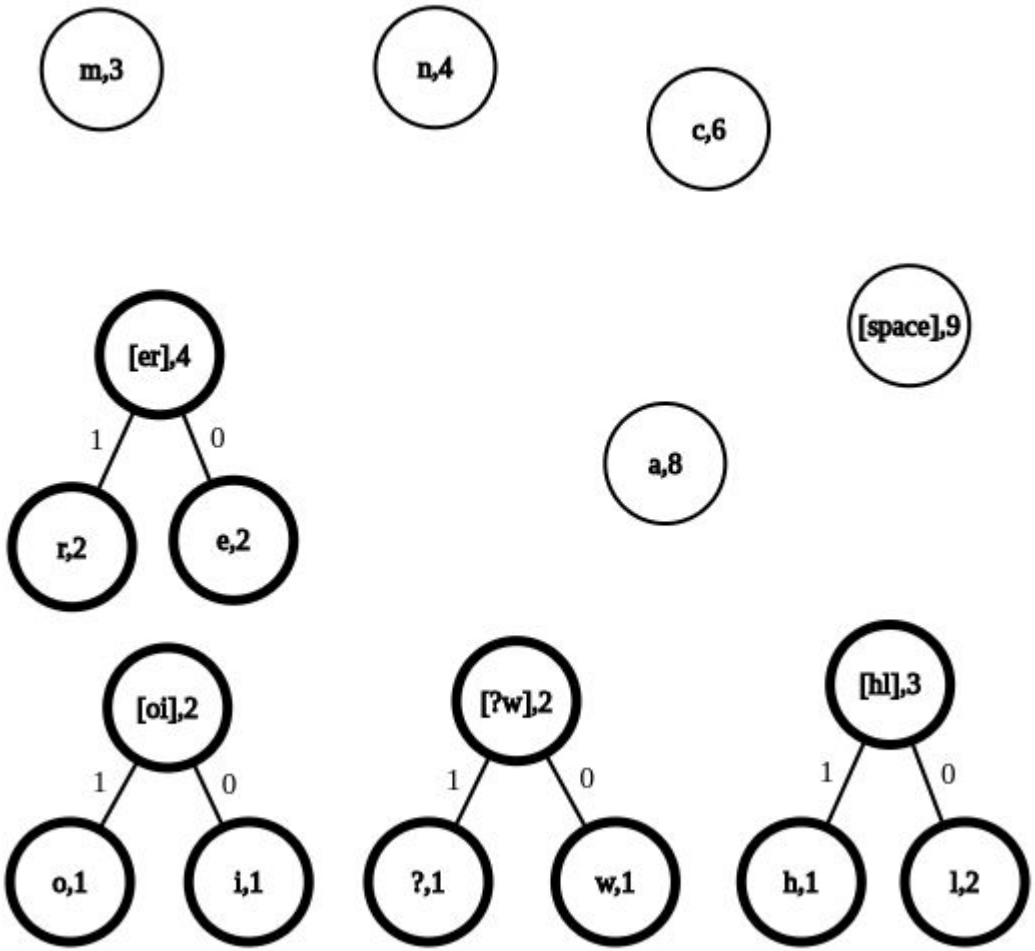
Example (Step 4)



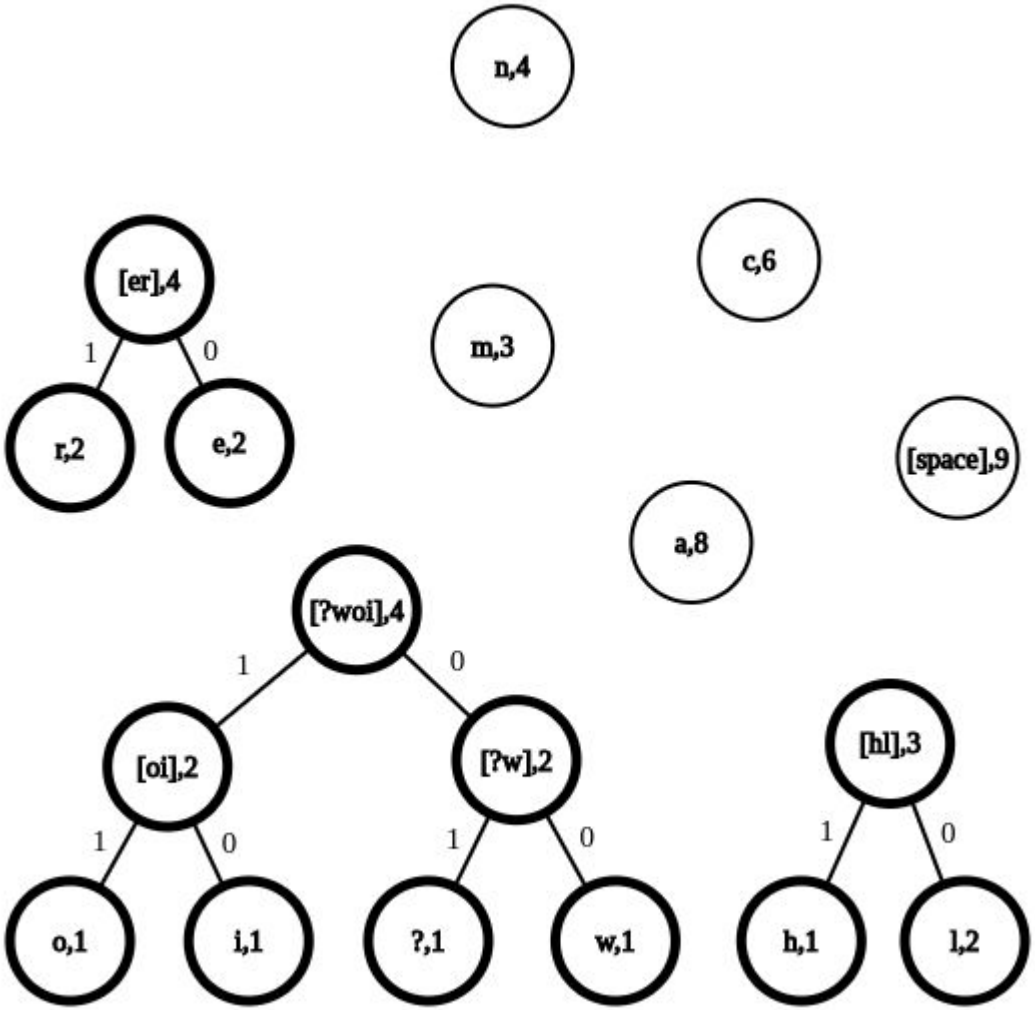
Example (Step 5)



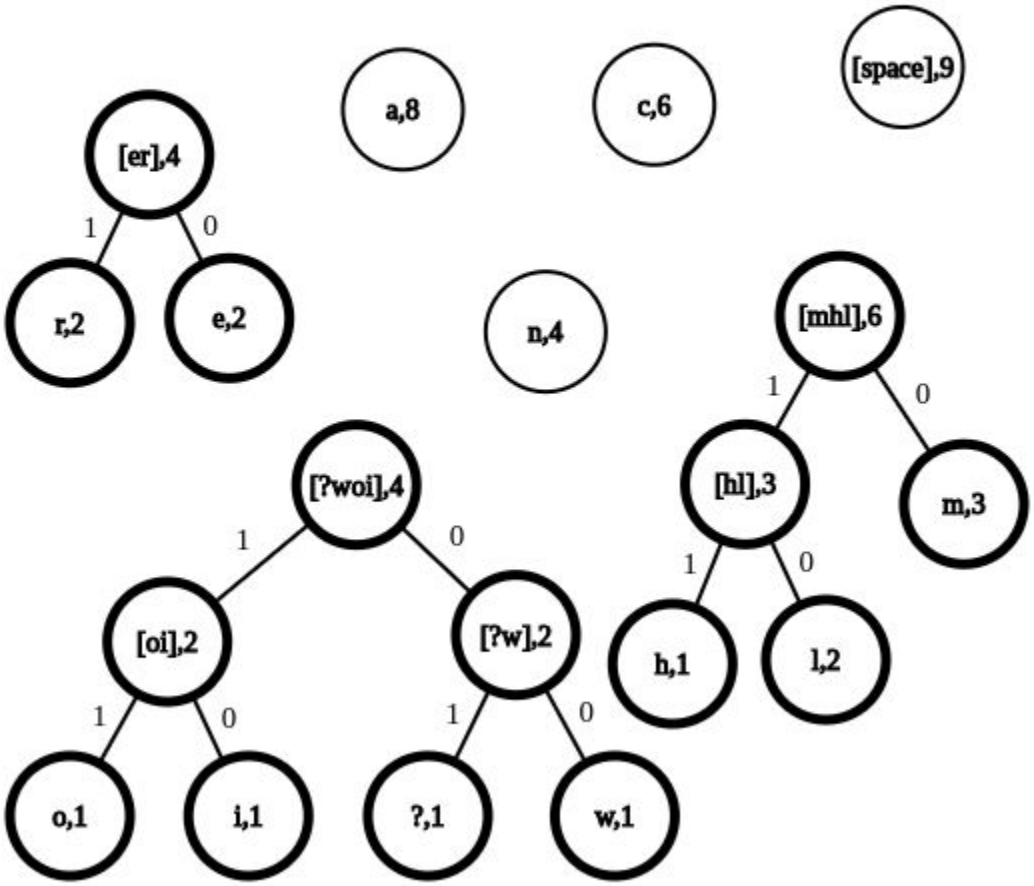
Example (Step 6)



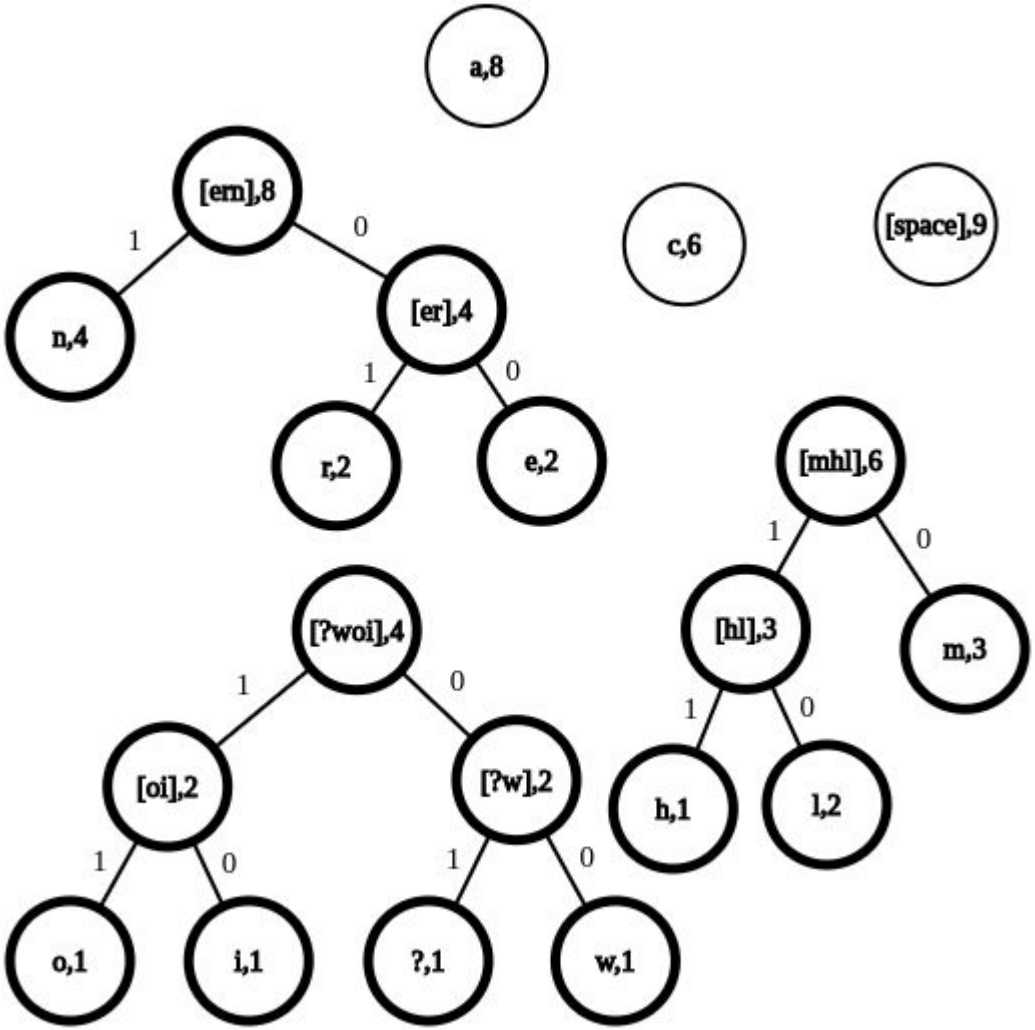
Example (Step 7)



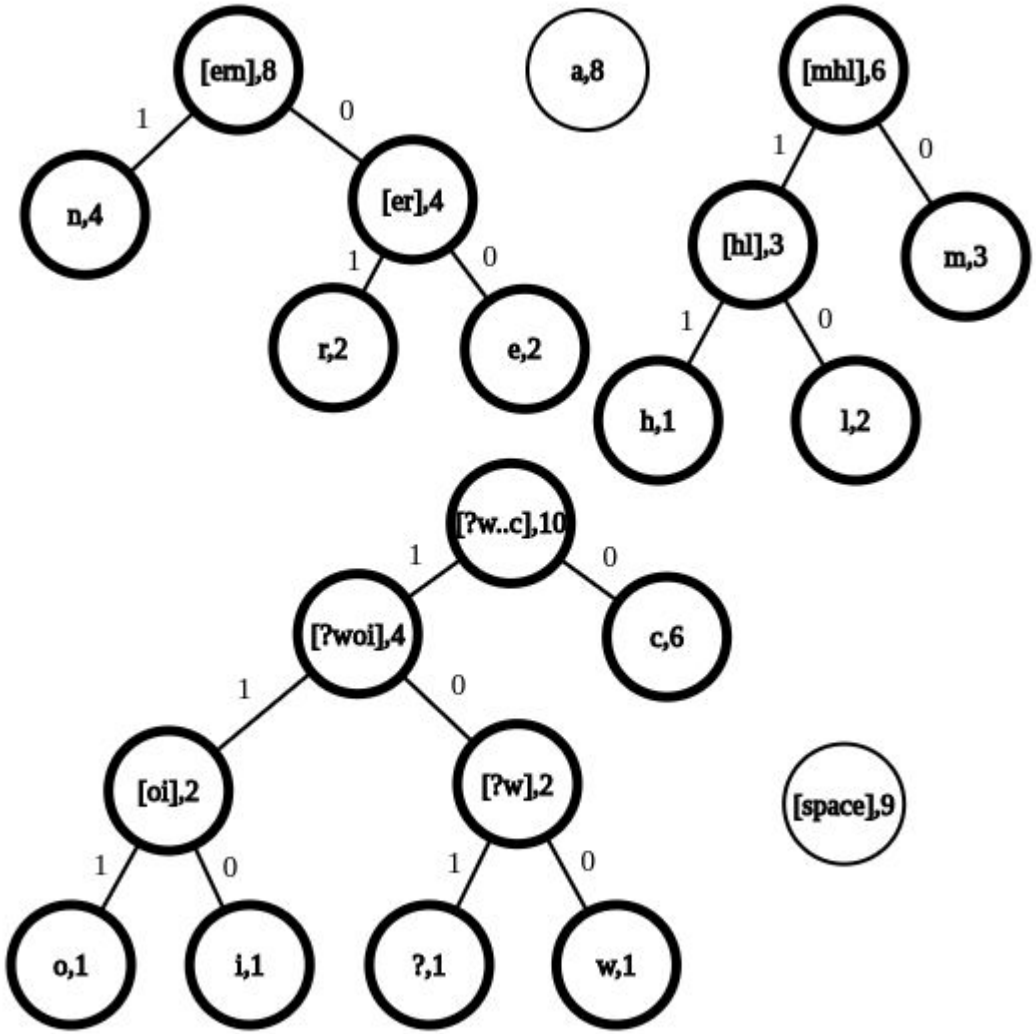
Example (Step 8)



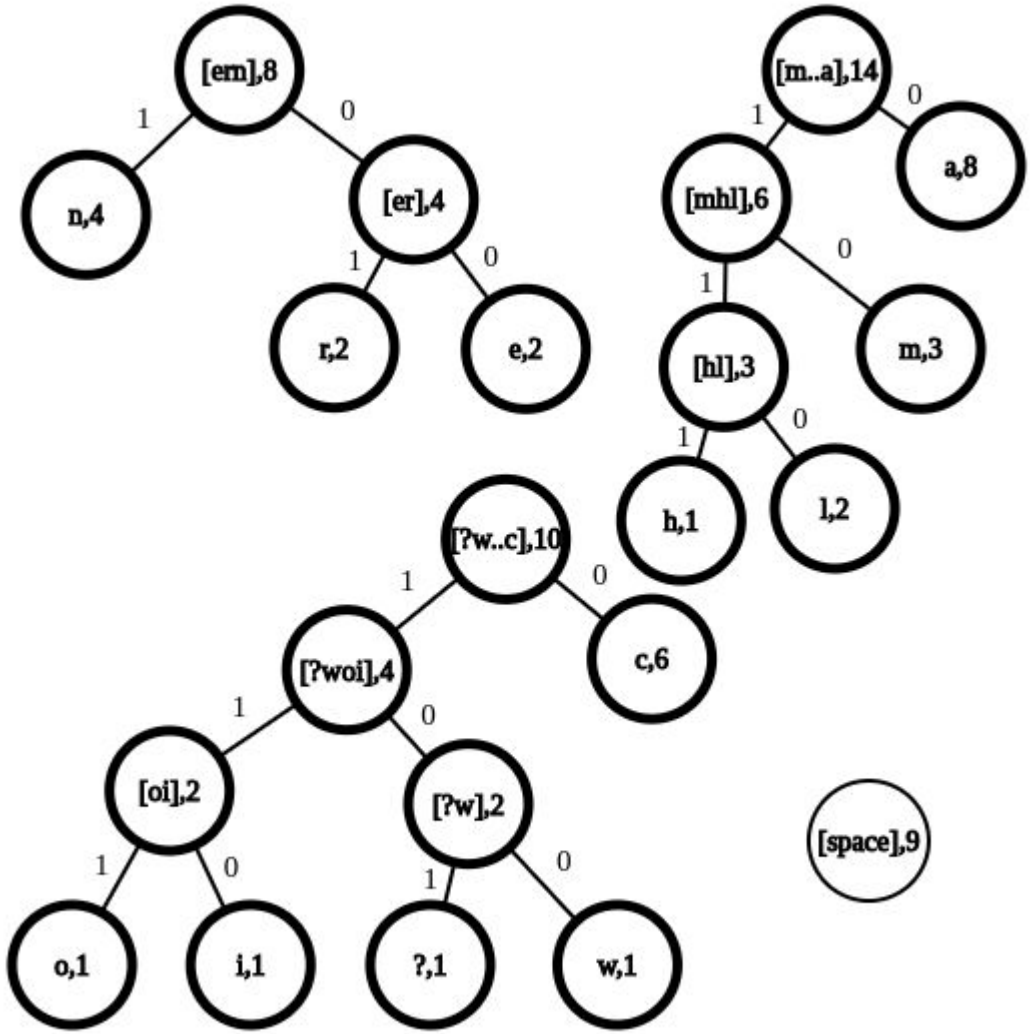
Example (Step 9)



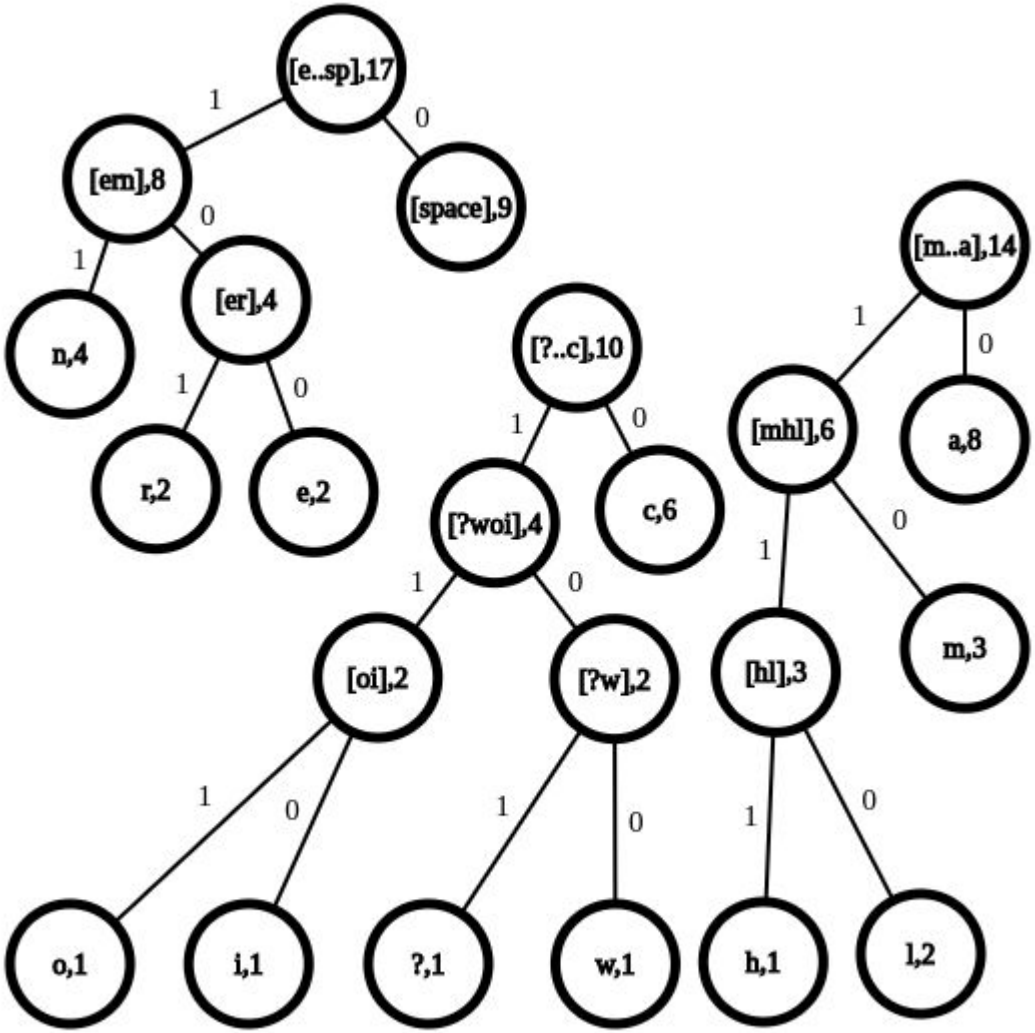
Example (Step 10)



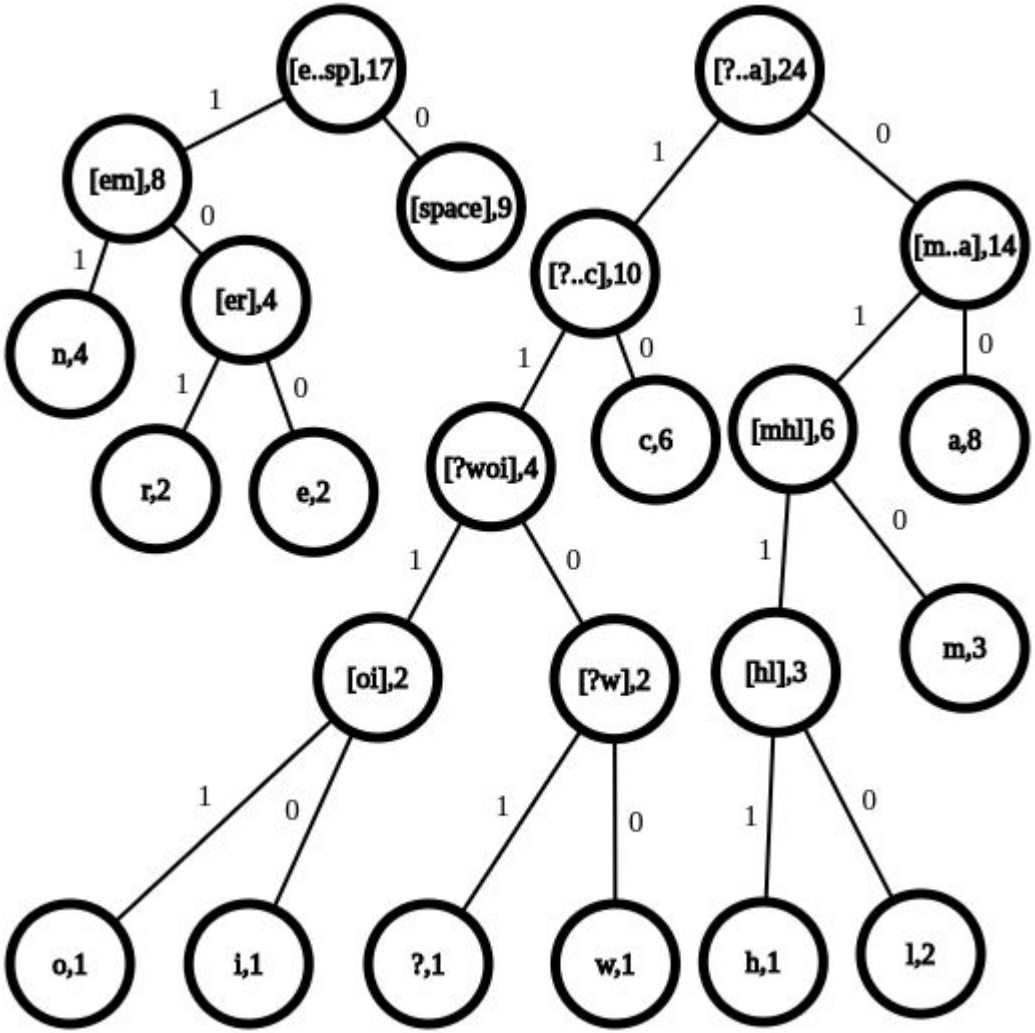
Example (Step 11)



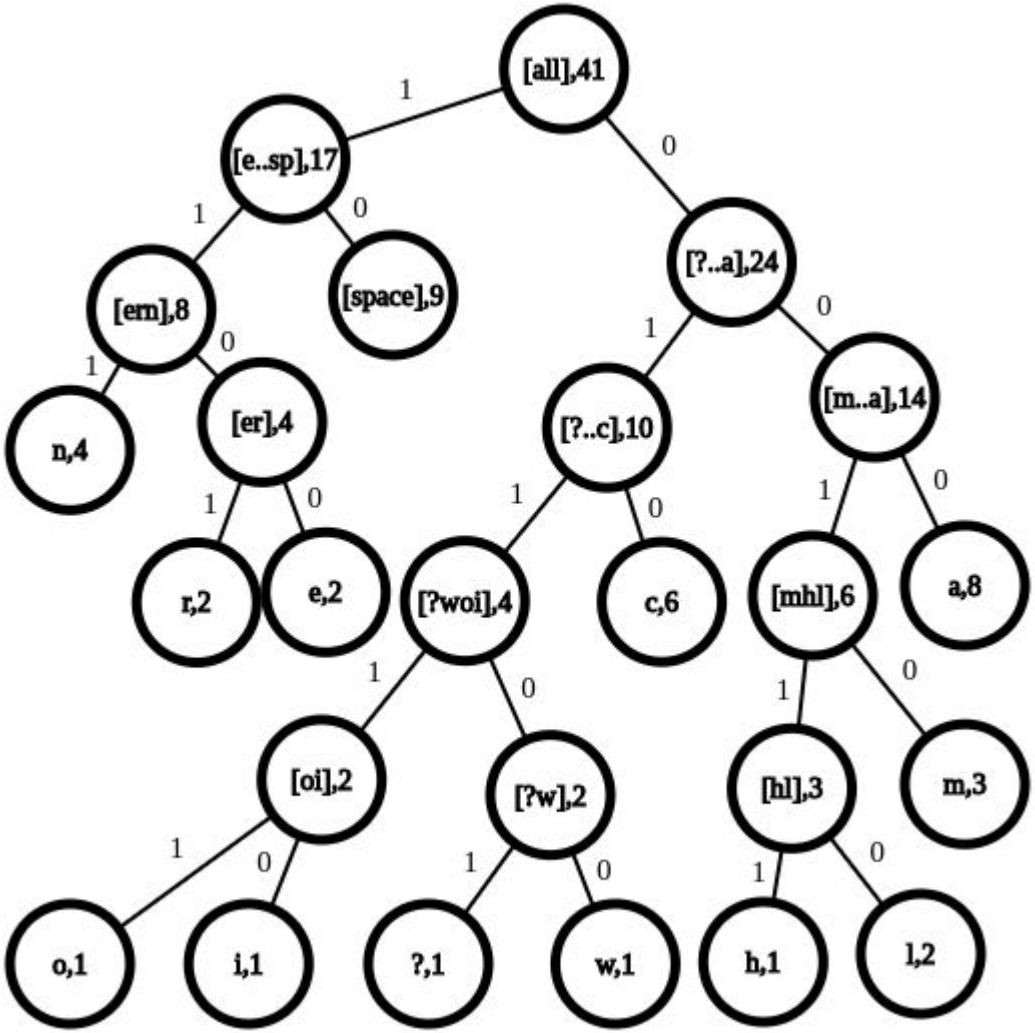
Example (Step 12)



Example (Step 13)



Example (Step 14)



Building Huffman Tree

- In order to keep the subtrees sorted by their frequency, we require another data structure.
- Data structure used for this purpose is PriorityQueue
- We covered Queue Interface in the tutorials, PriorityQueue is a class that implement Queue Interface.

Building Huffman Tree

- PriorityQueue use compareTo method of Comparable Interface in order to find out which elements should be at the beginning of the queue.
- Adding element to a PriorityQueue and removing element from the head takes O(1).

Tree Node

- We represent each node of the tree with Node class.

```
1.  class Node implements Comparable<Node> {
2.    private Node leftChild, rightChild;
3.    private int freq;
4.    private char ch;
5.    private boolean isLeaf;
6.  }
```

- Only the leaves of the tree have a character that represent them, other nodes only contains frequnecy, leftChild, and rightChild.

Tree Node

- There are two constructors for Node
  - Constructor for adding a character with its frequency.

```
public Node(int freq, char ch) {
  leftChild = rightChild = null;
  this.freq = freq;
  this.ch = ch;
  isLeaf = true;
}
```

  - Constructor that merges two Node together and build the parent of those nodes.

```
public Node(Node leftChild, Node rightChild) {
  freq = leftChild.freq + rightChild.freq;
  this.leftChild = leftChild;
  this.rightChild = rightChild;
  isLeaf = false;
}
```

Tree Node

- Since Node implement Comparable, it has to implement compareTo method.

```
@Override
public int compareTo(Node node) {
  return freq - node.getFreq();
}
```

- The Node with the lowest frequency comes first in the PriorityQueue.

Building Huffman Tree

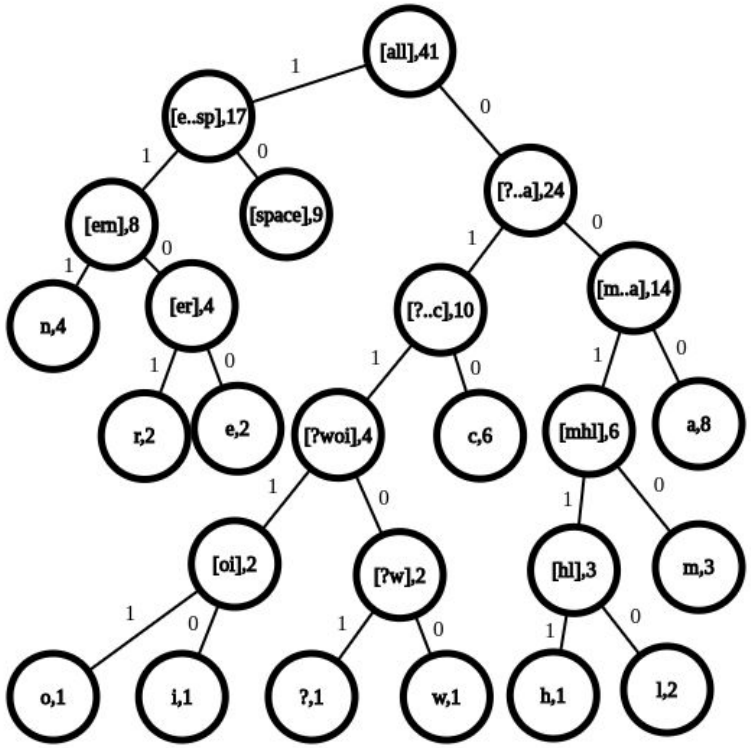
```
1.  ArrayList<Node> charsList = new ArrayList<>();
2.  for(Map.Entry elem: freqMap.entrySet()) {
3.     Node node = new Node((Integer) elem.getValue(), (Character) elem.getKey());
4.     charsList.add(node);
5.  }
6.  buildHuffmanTree(charsList)
7.
8.  public static Node buildHuffmanTree(ArrayList<Node> lst) {
9.     Queue<Node> q = new PriorityQueue<>(lst);//build PriorityQueue with elements of lst
10.    while(q.size() > 1) {
11.       Node a = q.remove();//Node with the lowest frequency
12.       Node b = q.remove();//Node with the second lowest frequency
13.       Node c = new Node(a, b);//Build Node c based on the merge of a and b
14.       q.add(c);
15.    }
16.    Node head = q.remove();
17.    return head;
18. }
19.
```

Encoding a Text File

- After building the tree, it only remains to traverse the tree, in order to find the binary code for each character.
- For example:

  n: 111

  m: 0010



Encoding a Text File

- The trick is to use a recursive algorithm for traversing the tree, and also save the edge labels in the tree that we encountered so far.
- We also use HashMap to save the binary code for each character.

Encoding a Text File

```
1.  HashMap<Character, String> codes = new HashMap<>();
2.  traverse(head, "", codes);
3.  private static void traverse(Node node, String code, HashMap<Character, String> codes) {
4.      if(node.isLeaf()) {
5.          codes.put(node.getCh(), code);
6.          return;
7.      }
8.      //add 0 to the end of binary value currently traversed
9.      traverse(node.getRightChild(), code.concat("0"), codes);
10.     traverse(node.getLeftChild(), code.concat("1"), codes);
11. }
```

Encoding a Text File

- Then we just change each character to its corresponding binary code.

```
1.  StringBuilder out = new StringBuilder("");
2.  for(int i = 0 ; i < inputText.length() ; i++)
3.     out.append(codes.get(inputText.charAt(i)));
```

- The following is the binary code of “how”:

  001110111101100

Decoding a Text File

- Huffman code has an important property, no two binary codes are prefix of each other.
- This property allows us to start from the beginning of the encoded file and consider the first prefix that matches with a binary code as the first character.
- Then apply the same operation for the rest of the encoded file.

Decoding a Text File

```
1.  String input = sc.next();
2.  String tmp = "";
3.  StringBuilder out = new StringBuilder("");
4.  for(int i = 0 ; i < input.length() ; i++) {
5.     tmp = tmp + input.charAt(i);
6.     if(codes.containsKey(tmp)) {
7.        out.append(codes.get(tmp));
8.        tmp = "";
9.     }
10. }
```

Decoding a Text File

- For example given 001110111101100, the first prefix that matches with any character is 00111**0**111101100, which is ‘h’.
- Then goes forward until it finds another match 00111**0**1111**0**1100, which is ‘o’.
- At last it finds the last match 00111**0**1111**01100**, which is ‘w’.