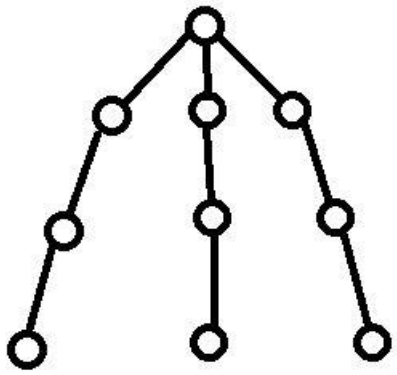
CPT_S 580 HW3

Yang Zhang

11529139 (graduate)

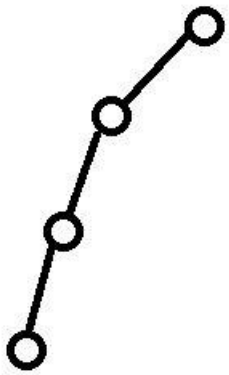
3.2.10

A ternary tree of height 3 with 10 vertices.



3.2.12

A ternary tree of height 3 with 4 vertices.



3.3.1

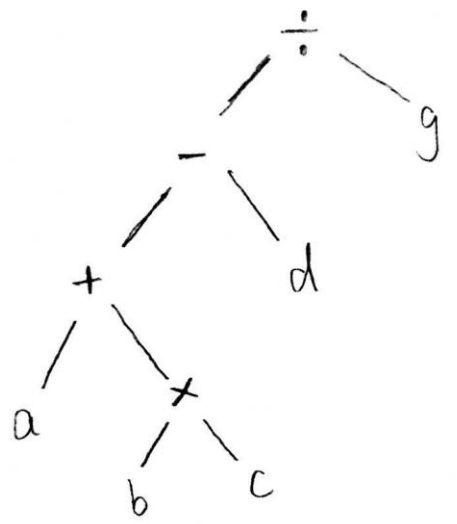
Level-order: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Pre-order: {1, 2, 4, 7, 8, 5, 3, 6, 9, 10}

In-order: {7, 4, 8, 2, 5, 1, 9, 6, 10, 3}

Post-order: {7, 8, 4, 5, 2, 9, 10, 6, 3, 1}

3.3.6



Prefix: {/, -, +, a, *, b, c, d, g}

Postfix: {a, b, c, *, +, d, -, g, /}

3.5.2

letter	a	b	c	d	e	f	g
Code word	00	0100	0101	011	10	110	111

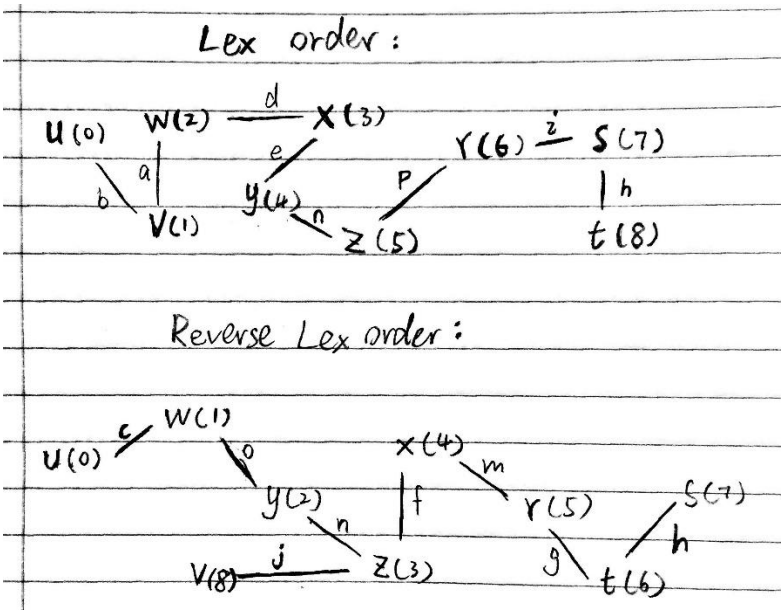
0100|00|111|111|00|111|10 = baggage

4.1.2

Frontier edges (one endpoint in the tree): {l, e, m, g, h}

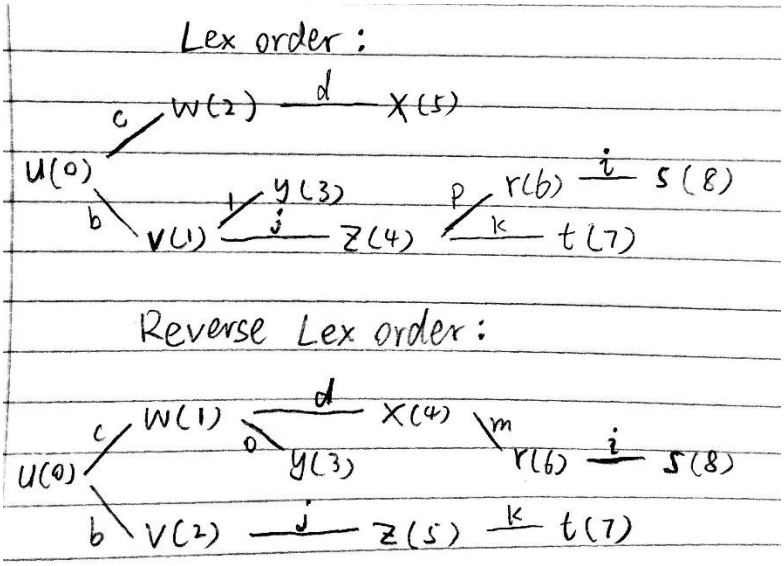
4.2.2

(Based on the vertex label)



4.2.9

(Based on the vertex label)



3.4.16

According to the definition of BST, any value from the left subtree must be smaller than current root value. Start from the root r, the value of left child c1 of the root is smaller than the value of root, and then we move the cursor from root r to c1. The value of c1's left child c2 is smaller than the value of c1, so we continue moving the cursor to c2. Repeating the steps until the cursor reaches the leaf node which is the left most leaf node, which means no other value could be smaller than it. Therefore, the left leaf node has the smallest value in BST.

4.1.17

According to the process of tree-growing algorithm, it initializes the starting vertex **v** as the root of spanning tree **T**. At each iteration, the algorithm grows the tree **T** by adding the specific frontier edge and a new vertex (the non-tree vertex of the frontier edge) is also discovered**.** Since new added vertex is part of the tree **T**, it can be reached from the starting vertex. Therefore, the starting vertex can reach any vertex discovered by the algorithm.

Coding Problems (in Java)

3.4.17

The problem is asking to implement the binary search tree search algorithm. By implementing the algorithm, we need firstly need to construct a BST from given sorted array (at beginning we pass start = 0, end = length of the array - 1):

If start > end return null, else:

1. Create node **v** with value of mid index of the array (mid = (start + end)/2)
2. **v** -> left sub tree = recursive call (arry, start, mid-1)
3. **v** -> left sub tree = recursive call (arry, mid+1, end)
4. return **v**

By implementing the search algorithm:

we test the root of the tree at each recursive call, return true if the value of current node is the answer, otherwise:

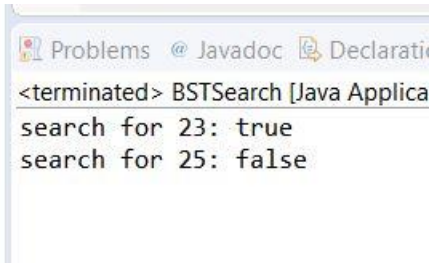
1. If the target > root.data we recursively exam the right sub tree.
2. If the target $\leq$ root.data we recursively exam the left sub tree.
3. If we reach the leaf and its data not equal to the target, then there is no such value in this tree, return false.

Input: sorted array

Output: Boolean value

Test case: {3,8,9,12,14,22,23,28,35,40,46}

The result:



From the result, the algorithm could determine if the target is in the tree correctly.

Functions:

BSTreeNode(int data): constructor that constructs a node with given data as its value

BSTreeNode(int data, BSTreeNode left, BSTreeNode right): constructor that constructs a node with given data and sub trees

constructTree(int[] ary, int start, int end): constructs a BST from a sorted array recursively

search(BSTreeNode root, int key): search a target value in the given BST

3.5.10

The problem is asking to implement the Huffman algorithm. By implementing the algorithm, we need firstly construct the Huffman tree:

1. Convert the data with its frequency into a Huffman tree node.
2. Put all those nodes into a priority queue (sorting based on its the frequency)
3. While the queue size is bigger than one, each time pop out two nodes from the priority queue, and use them to form a new node with frequency equals to the sum of the two nodes.
4. When queue size is 1, pop out the node as the tree root

To get the Huffman code:

Traverse the Huffman tree, record 0 when checking the left sub tree (record 1 for right sub tree), until reach the leaf node.

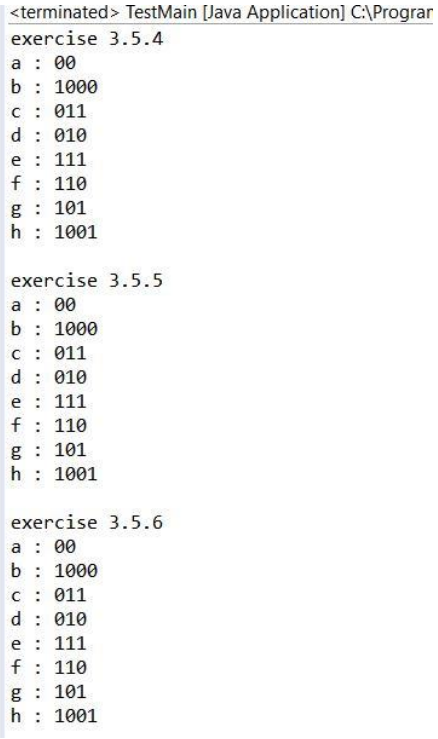
Input: map of (char, frequency) pairs

Output: list of Huffman code

Test case:

chars	a	b	c	d	e	f	g	h
Freq 1 (3.5.4)	.2	.05	.1	.1	.18	.15	.15	.07
Freq 2 (3.5.5)	.1	.15	.2	.17	.13	.15	.05	.05
Freq 3 (3.5.6)	.15	.1	.15	.12	.08	.25	.05	.1

The result:



According to the result, we can see that the Huffman algorithm always put the node with higher frequency at the front.

Functions:

HuffmanTree(Map<Character, Double> counts): constructor that build a Huffman tree from given map

Map<Character, String> createEncodings(): function that creates Huffman encoding

Map<Character, String> mapHelper (HuffmanNode cur, Map<Character, String> map,String temp): helper function to actually create Huffman encoding