# Comp151 Lab13

You will be working on **four** applications that use Tree ADT.

## UML Diagram

**A screenshot of a cell phone

Description automatically generated**

## Application #1

The Morse code is a common code that is used to encode messages. In our application, we will assume that the messages contain letters only. Each letter consists of a series of dots (.) and underscores (\_); for example, the code for the letter *a* is .\_ and the code for the letter *b* is \_....

See the table below:



Our program stores each letter of the alphabet in a binary tree. The root of the tree contains no letter. Its left node store the letter *e* (code is .), and its right node stores the letter *t* (code is \_). The 4 nodes at the next level store the letters with codes (.., .\_, -., \_\_) .

To build the tree we created a text file where each line consists of a letter and its code. The letters in the file are ordered by the tree level. The method buildMorseCodeTree() creates the Morse Code Tree:



Study the provided MorseCode.java and run the program. It has main method that prompts the user for a message encoded in Morse code and calls the decode method to decode the message and display the result. Your task is to implement the decode method. To find the position for a letter in the tree, scan the code and branch left for the dot and branch right for an underscore. Assume that the encoded letters are separated by spaces.

Implement decode method defined in the MorseCode.java file.

The sample run of the program is provided below:

The Morse Code:

===============

e .

t \_

i ..

a .\_

n \_.

m \_\_

s ...

u ..\_

r .\_.

w .\_\_

d \_..

k \_.\_

g \_\_.

o \_\_\_

h ....

v ...\_

f ..\_.

l .\_..

p .\_\_.

j .\_\_\_

b \_...

x \_..\_

c \_.\_.

y \_.\_\_

z \_\_..

q \_\_.\_

The Morse Code Tree in level order:

e t i a n m s u r w d k g o h v f l p j b x c y z q

The Morse Code Tree in pre-order:

e i s h v u f a r l w p j t n d b x k c y m g z q o

Please enter a message in Morse Code, use space as a separator. Press enter to stop.

... \_\_\_ ...

Decoding "... \_\_\_ ..."

The decoded string is "sos"

Please enter a message in Morse Code, use space as a separator. Press enter to stop.

\_.\_. ... ..\_ \_.\_. ..

Decoding "\_.\_. ... ..\_ \_.\_. .."

The decoded string is "csuci"

Please enter a message in Morse Code, use space as a separator. Press enter to stop.

.\_.

Decoding ".\_."

The decoded string is "r"

Please enter a message in Morse Code, use space as a separator. Press enter to stop.

.\_.\_

Decoding ".\_.\_"

Not a morse pattern.

Please enter a message in Morse Code, use space as a separator. Press enter to stop.

Done decoding.

## Process finished with exit code 0

## Application #2

Write a program that takes a postfix expression and produces a binary expression tree. You can assume that the postfix expression is a string that has only binary operators and one-letter operands.

Modify PostfixTree.java file.  Follow steps #1 - #7 included in the file.

The expected output of your finished program is as follow:

The first postfix expression is:

ab\*c+

The inorder traversal is:

a \* b + c

The postorder traversal is:

a b \* c +

The second postfix expression is:

ab-c\*def-+g/+

The inorder traversal is:

a - b \* c + d + e - f / g

The postorder traversal is:

a b - c \* d e f - + g / +

**Application #3**

Open WorkingWithBinaryTrees.java and implement the following methods that are defined there:

1. isBST – **iterative** method that accepts as its argument a **BinaryNode** object representing the root of the tree and returns true if the argument tree is a binary search tree.
   * Examine each node in the given tree only once.
   * HINT – a binary tree whose inorder traversal does not visit nodes in the natural order of the data is not a binary tree. As you follow the ionorder traversal check this property and stop the traversal as soon as the property is not met.
2. areTwoBSTsSame – **iterative** method that accepts two **BinaryNode** objects representing roots of two binary search trees. Utilizing InorderIterator objects check if the two trees have the same elements. Please note that the InorderIterator class is defined as an inner class but next method must be first implemented.
3. buildBSTfromSortedArray – **recursive** method that creates shortest possible **binary search tree** from all the elements included in the **sorted array** that is passed to the method.
4. printBSTinLevelOrder – **iterative** method that accepts as its argument a **BinaryNode** object representing the root of the tree and prints the elements of the tree in level order.
   * Utilize queue to store the nodes
   * Enqueue the root
   * While the queue is not empty dequeue all nodes of the current level (use the queue size to determine how many nodes to pull) and enqueue all nodes of the next level (left and right child of the pulled node)
5. getSmallest – **recursive** method that finds and returns the smallest data in a **binary search tree** or null if the tree is empty
6. getSecondLargest – **iterative** method that finds and returns the second largest data in a **binary search tree** containing at least two nodes. If the binary search tree contains less than two nodes, the method returns null.
   * HINT - if the largest value is in node L, the second largest value is in either the parent of node L or the largest node in L’s left subtree. The following algorithm is based on this idea:

***Algorithm to find the second largest value in a binary search tree T***

*parentNode = the root of T*

*largestNode = the right child of parentNode*

*// Find the largest node and its parent (if there is one)*

*if (largestNode is not null){ while (the right child of largestNode is not null)*

*{ parentNode = largestNode largestNode = the right child of largestNode*

*} }*

*else*

*{*

*largestNode = the root of T*

*}*

*if (the left child of largestNode is not null)*

*{*

*secondLargestNode = the left child of largestNode*

*while (the right child of secondLargestNode is not null)*

*{*

*secondLargestNode = the right child of secondLargestNode*

*}*

*}*

*else*

*{*

*secondLargestNode = parentNode*

*}*

*return the data in secondLargestNode*

### The expected output of your finished program:

Tree 1:

A

/ \

B C

/ \ / \

D E F G

Traversing the tree "InOrder" to check if it is BST: D B

tree1 is BST = false -> CORRECT

Tree 1a:

D

/ \

B F

/ \ / \

A C E G

Traversing the tree "InOrder" to check if it is BST: A B C D E F G

tree1a is BST = true -> CORRECT

The smallest element = A

The second largest element = F

Tree 2:

A

/ \

B C

\ / \

E F G

Traversing the tree "InOrder" to check if it is BST: B E A

tree2 is BST = false -> CORRECT

Tree 2a:

C

/ \

A F

\ / \

B E G

Traversing the tree "InOrder" to check if it is BST: A B C E F G

tree2a is BST = true -> CORRECT

The smallest element = A

The second largest element = F

Tree 3:

A

/ \

B C

/ \ /

D E F

\

G

Traversing the tree "InOrder" to check if it is BST: D B

tree3 is BST = false -> CORRECT

Tree 3a:

D

/ \

B G

/ \ /

A C E

\

F

Traversing the tree "InOrder" to check if it is BST: A B C D E F G

tree3a is BST = true -> CORRECT

The smallest element = A

The second largest element = F

Tree 4:

A

/ \

B C

/ \ / \

D E F G

\

H

Traversing the tree "InOrder" to check if it is BST: D H B

tree4 is BST = false -> CORRECT

Tree 4a:

E

/ \

C G

/ \ / \

A D F H

\

B

Traversing the tree "InOrder" to check if it is BST: A B C D E F G H

tree4a is BST = true -> CORRECT

The smallest element = A

The second largest element = G

Tree 5:

A

/ \

B C

/ \ / \

D E F G

\

H

Traversing the tree "InOrder" to check if it is BST: D B

tree5 is BST = false -> CORRECT

Tree 5a:

E

/ \

B G

/ \ / \

A C F H

\

D

Traversing the tree "InOrder" to check if it is BST: A B C D E F G H

tree5a is BST = true -> CORRECT

The smallest element = A

The second largest element = G

Tree 6:

A

/ \

B C

/ \ \

D E H

/ \

F G

Traversing the tree "InOrder" to check if it is BST: D B

tree6 is BST = false -> CORRECT

Tree 6a:

F

/ \

B G

/ \ \

A D H

/ \

C E

Traversing the tree "InOrder" to check if it is BST: A B C D E F G H

tree6a is BST = true -> CORRECT

The smallest element = A

The second largest element = G

Tree 7:

G

\

B

\

E

\

A

Traversing the tree "InOrder" to check if it is BST: G B

tree7 is BST = false -> CORRECT

Tree 7a:

A

\

B

\

E

\

G

Traversing the tree "InOrder" to check if it is BST: A B E G

tree7a is BST = true -> CORRECT

The smallest element = A

The second largest element = E

Tree 8a:

D

/ \

B F

Traversing the tree "InOrder" to check if it is BST: B D F

tree8a is BST = true -> CORRECT

The smallest element = B

The second largest element = D

Tree 9a:

D

/

B

Traversing the tree "InOrder" to check if it is BST: B D

tree9a is BST = true -> CORRECT

The smallest element = B

The second largest element = B

Tree 10a:

D

/

B

Traversing the tree "InOrder" to check if it is BST: B D

tree10a is BST = true -> CORRECT

The smallest element = B

The second largest element = B

Tree 11a:

D

Traversing the tree "InOrder" to check if it is BST: D

tree11a is BST = true -> CORRECT

The smallest element = D

The second largest element = null

Tree 12a:

B

/ \

A D

/ \

C E

Traversing the tree "InOrder" to check if it is BST: A B C D E

tree12a is BST = true -> CORRECT

The smallest element = A

The second largest element = D

Tree 12b:

B

/ \

A C

\

E

/

D

Traversing the tree "InOrder" to check if it is BST: A B C D E

tree12b is BST = true -> CORRECT

The smallest element = A

The second largest element = D

Are elements in tree12a and tree12b the same = true -> CORRECT

Tree 12c:

B

/ \

A C

\

F

/

D

Traversing the tree "InOrder" to check if it is BST: A B C D F

tree12c is BST = true -> CORRECT

The smallest element = A

The second largest element = D

Are elements in tree12a and tree12c the same = false -> CORRECT

Tree 12d:

A

\

B

\

C

\

D

\

E

Traversing the tree "InOrder" to check if it is BST: A B C D E

tree12d is BST = true -> CORRECT

The smallest element = A

The second largest element = D

Are elements in tree12a and tree12d the same = true -> CORRECT

Tree 12e:

E

/

D

/

C

/

B

/

A

Traversing the tree "InOrder" to check if it is BST: A B C D E

tree12d is BST = true -> CORRECT

The smallest element = A

The second largest element = D

Are elements in tree12a and tree12e the same = true -> CORRECT

Working with tree built from the array: [A, B, C, D, E, F, G, H, I, J, K]

Traversing the tree "InOrder" to check if it is BST: A B C D E F G H I J K

The tree is BST = true

The tree has 11 nodes and the height of 4

The tree in level order:

F <-- level 1

C I <-- level 2

A D G J <-- level 3

B E H K <-- level 4

Process finished with exit code 0

## Application #4

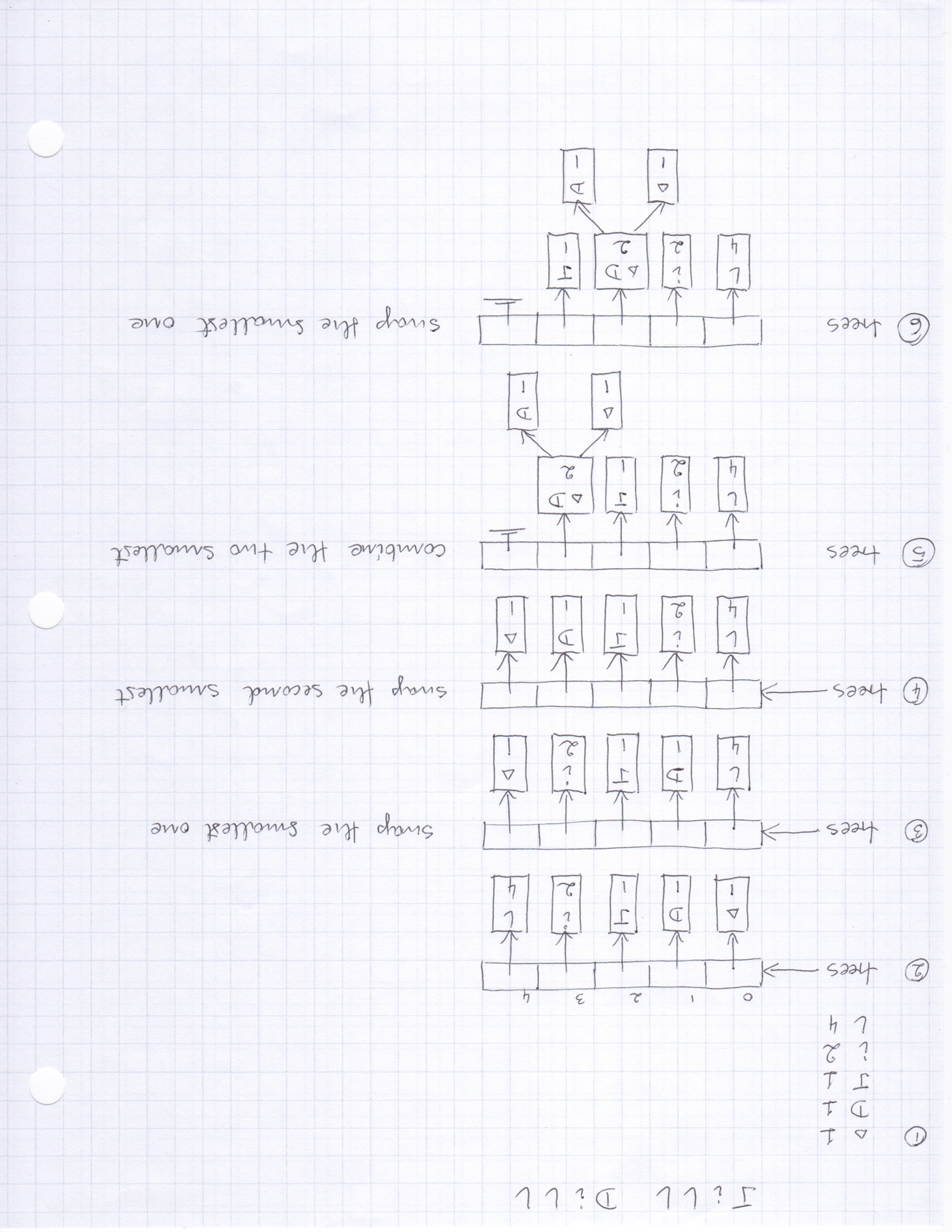
We will generalize above application by applying [Huffman Coding](http://en.wikipedia.org/wiki/Huffman_coding) algorithm. Please read the provided link first.

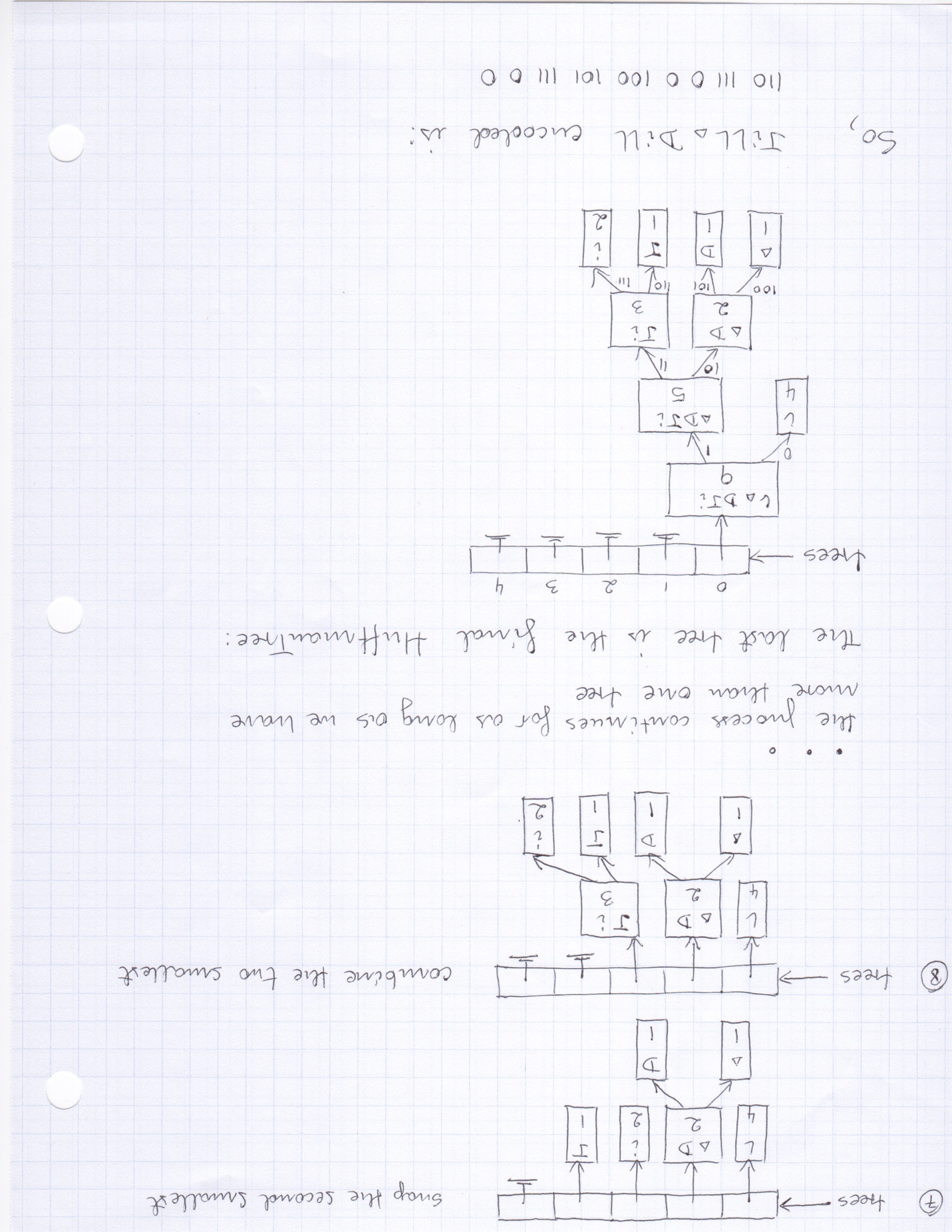
In general terms the algorithm is as follow:

1. Count frequency of each letter and save this information in an array of Huffman trees
2. Repeat the following steps until the complete Huffman tree is created (the number of trees at the end will be 1)
   1. Find the smallest tree and swap it to be the last
   2. Find the second smallest tree and swap it to be the second last
   3. Combine the two smallest trees into one and place it where the second smallest three was
   4. Decrement number of trees by 1
3. Encode the given message
   1. Branching to the left will encode as ‘0’, and branching to the right will encode as ‘1’

Based on the above algorithm “Jill Dill” will encode to “110 111 0 0 100 101 111 0 0”

See the steps depicted below:





Following the above example create the Huffman code for your first name last name (as part of the pre-lab assignment). See the sample runs below to better understand the algorithm.

### Sample run for message4.txt:

The message lines are:

Jill Dill

--->Count of character is 1

--->Count of character D is 1

--->Count of character J is 1

--->Count of character i is 2

--->Count of character l is 4

Creating 5 initial trees

Building Huffman Tree

--->Smallest tree moved to the position 4

--->Second smallest tree moved to the position 3

--->Combined tree created: [ , D] -> 2

--->Combined tree added at position 3

--->Smallest tree moved to the position 3

--->Second smallest tree moved to the position 2

--->Combined tree created: [J, i] -> 3

--->Combined tree added at position 2

--->Smallest tree moved to the position 2

--->Second smallest tree moved to the position 1

--->Combined tree created: [ , D, J, i] -> 5

--->Combined tree added at position 1

--->Smallest tree moved to the position 1

--->Second smallest tree moved to the position 0

--->Combined tree created: [l, , D, J, i] -> 9

--->Combined tree added at position 0

Huffman Tree:

[l, , D, J, i] -> 9

[l] -> 4

[ , D, J, i] -> 5

[ , D] -> 2

[ ] -> 1

[D] -> 1

[J, i] -> 3

[J] -> 1

[i] -> 2

The coded lines are (displaying 80 characters per line):

110 111 0 0 100 101 111 0 0

### Sample run for message1.txt:

The message lines are:

ABRACADABRA

ABBA

--->Count of character A is 7

--->Count of character B is 4

--->Count of character C is 1

--->Count of character D is 1

--->Count of character R is 2

Creating 5 initial trees

Building Huffman Tree

--->Smallest tree moved to the position 4

--->Second smallest tree moved to the position 3

--->Combined tree created: [C, D] -> 2

--->Combined tree added at position 3

--->Smallest tree moved to the position 3

--->Second smallest tree moved to the position 2

--->Combined tree created: [R, C, D] -> 4

--->Combined tree added at position 2

--->Smallest tree moved to the position 2

--->Second smallest tree moved to the position 1

--->Combined tree created: [B, R, C, D] -> 8

--->Combined tree added at position 1

--->Smallest tree moved to the position 1

--->Second smallest tree moved to the position 0

--->Combined tree created: [A, B, R, C, D] -> 15

--->Combined tree added at position 0

Huffman Tree:

[A, B, R, C, D] -> 15

[A] -> 7

[B, R, C, D] -> 8

[B] -> 4

[R, C, D] -> 4

[R] -> 2

[C, D] -> 2

[C] -> 1

[D] -> 1

The coded lines are (displaying 80 characters per line):

0 10 110 0 1110 0 1111 0 10 110 0 0 10 10 0

### Sample run for message3.txt:

The message lines are:

this is an example of a huffman tree

--->Count of character is 7

--->Count of character a is 4

--->Count of character e is 4

--->Count of character f is 3

--->Count of character h is 2

--->Count of character i is 2

--->Count of character l is 1

--->Count of character m is 2

--->Count of character n is 2

--->Count of character o is 1

--->Count of character p is 1

--->Count of character r is 1

--->Count of character s is 2

--->Count of character t is 2

--->Count of character u is 1

--->Count of character x is 1

Creating 16 initial trees

Building Huffman Tree

--->Smallest tree moved to the position 15

--->Second smallest tree moved to the position 14

--->Combined tree created: [l, x] -> 2

--->Combined tree added at position 14

--->Smallest tree moved to the position 14

--->Second smallest tree moved to the position 13

--->Combined tree created: [u, o] -> 2

--->Combined tree added at position 13

--->Smallest tree moved to the position 13

--->Second smallest tree moved to the position 12

--->Combined tree created: [p, r] -> 2

--->Combined tree added at position 12

--->Smallest tree moved to the position 12

--->Second smallest tree moved to the position 11

--->Combined tree created: [h, p, r] -> 4

--->Combined tree added at position 11

--->Smallest tree moved to the position 11

--->Second smallest tree moved to the position 10

--->Combined tree created: [s, i] -> 4

--->Combined tree added at position 10

--->Smallest tree moved to the position 10

--->Second smallest tree moved to the position 9

--->Combined tree created: [u, o, l, x] -> 4

--->Combined tree added at position 9

--->Smallest tree moved to the position 9

--->Second smallest tree moved to the position 8

--->Combined tree created: [t, m] -> 4

--->Combined tree added at position 8

--->Smallest tree moved to the position 8

--->Second smallest tree moved to the position 7

--->Combined tree created: [n, f] -> 5

--->Combined tree added at position 7

--->Smallest tree moved to the position 7

--->Second smallest tree moved to the position 6

--->Combined tree created: [a, e] -> 8

--->Combined tree added at position 6

--->Smallest tree moved to the position 6

--->Second smallest tree moved to the position 5

--->Combined tree created: [u, o, l, x, t, m] -> 8

--->Combined tree added at position 5

--->Smallest tree moved to the position 5

--->Second smallest tree moved to the position 4

--->Combined tree created: [s, i, h, p, r] -> 8

--->Combined tree added at position 4

--->Smallest tree moved to the position 4

--->Second smallest tree moved to the position 3

--->Combined tree created: [n, f, ] -> 12

--->Combined tree added at position 3

--->Smallest tree moved to the position 3

--->Second smallest tree moved to the position 2

--->Combined tree created: [u, o, l, x, t, m, s, i, h, p, r] -> 16

--->Combined tree added at position 2

--->Smallest tree moved to the position 2

--->Second smallest tree moved to the position 1

--->Combined tree created: [a, e, n, f, ] -> 20

--->Combined tree added at position 1

--->Smallest tree moved to the position 1

--->Second smallest tree moved to the position 0

--->Combined tree created: [u, o, l, x, t, m, s, i, h, p, r, a, e, n, f, ] -> 36

--->Combined tree added at position 0

Huffman Tree:

[u, o, l, x, t, m, s, i, h, p, r, a, e, n, f, ] -> 36

[u, o, l, x, t, m, s, i, h, p, r] -> 16

[u, o, l, x, t, m] -> 8

[u, o, l, x] -> 4

[u, o] -> 2

[u] -> 1

[o] -> 1

[l, x] -> 2

[l] -> 1

[x] -> 1

[t, m] -> 4

[t] -> 2

[m] -> 2

[s, i, h, p, r] -> 8

[s, i] -> 4

[s] -> 2

[i] -> 2

[h, p, r] -> 4

[h] -> 2

[p, r] -> 2

[p] -> 1

[r] -> 1

[a, e, n, f, ] -> 20

[a, e] -> 8

[a] -> 4

[e] -> 4

[n, f, ] -> 12

[n, f] -> 5

[n] -> 2

[f] -> 3

[ ] -> 7

The coded lines are (displaying 80 characters per line):

0010 0110 0101 0100 111 0101 0100 111 100 1100 111 101 00011 100 0011 01110 0001

0 101 111 00001 1101 111 100 111 0110 00000 1101 1101 0011 100 1100 111 0010 011

11 101 101