Implementation in C++ (Labs 2, 3 and 4):

The program will be implemented using four distinct modules

- A Weighted Tree ADT (C++ class WTree)
- A "List of Trees" ADT (C++ class TreeList)
- A Coder module (C++ class Coder)
- A main module (In this case, most of the administative work is done by the class Coder, main is quite simple)

For lab-2, the WeightedTree ADT and the List of Trees ADT will be provided to you in the form of object code (WTree.o TreeList.o). Refer to the files WTree.h TreeList.h for the specification of its interface. You should implement the Coder class except for the encode and decode methods and test it using the given driver program. The Coder class functions you need to implemnt includes:

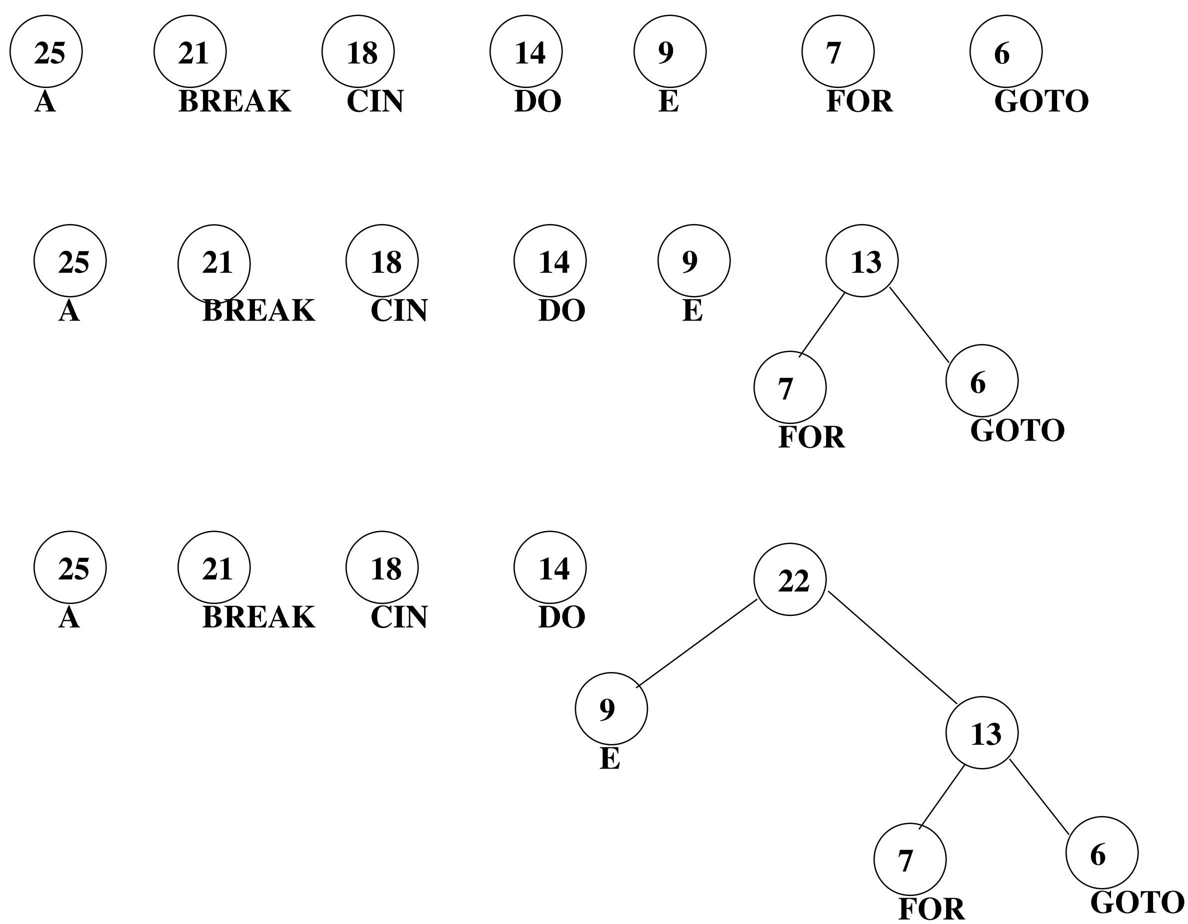
- Coder constructor, initialize its data structures
- read_symbols(), read in the symbols and their frequencies from the standard input stream. Refer to the given sample input files (Huffman*.inp) for the format of the input data.
- generate_huffman_code(), create the Huffman tree for the given set of symbols in this Coder:
  - o Declare a TreeList object.
  - o Loop: create a Tree (with a single leaf node) for each symbol and insert it into TreeList.
  - o Loop: find the two minimum weight trees in the TreeList and merge them.
  - o Store the Huffman code for each symbol in the appropriate data structure declared as part of the Coder class, i.e. for the final tree left in the TreeList, in a loop, use the Hcode method of class WTree to find the codes and write it in the huffman code data structure.
- print_huffman_code(), output the Huffman code for each symbol, i.e. Print out the huffman code stored in the Coder class with its original symbols. Refer to the expected output files (Huffman*.exp) to see the format in which to print out these codes.

For lab-3, you will implement a generic PriorityList ADT as a C++ template. Again, the specification will be provided in the TA account. The methods in the Coder class should then be modified so that it declares a TreeList class as an instance of this template, and uses it as before in the Huffman code computation. Also, the symbols and their frequencies will no longer be provided as input. Instead, the read_symbol method in the Coder class should read in a text file, extract all the symbols (including keywords and letters, the list of keywords will be given later) found in it and count their frequencies. Then the Huffman tree for these symbols should be built and used to encode the given text file for output.

For lab-4, you will implement the WeightedTree ADT, and test it independently using the provided driver program. You can then integrate it with the TreeList class and the Huffman program developed in lab-3, instead of using the original version provided to you.

The algorithm which assigns these codes constructs an explicit Huffman tree in bottom-up fashion. An array is constructed initially containing nothing but single-node subtrees, one for each of the symbols of the alphabet. Each of these subtrees has a corresponding frequency. When the algorithm is finished, there will be a single tree with *n* leaf nodes. Each leaf node will contain a symbol and its frequency.

In each iteration, the two trees with the lowest frequency are removed from the array and combined into a single tree which is added back into the array. The combined tree has the two original trees as left and right branches, and its frequency is the sum of the frequencies of those two branches. The first few steps of the process are shown below:



This process is continued until there is only one tree left in the array. This will be the Huffman tree representing the encoding for the alphabet we started out with.

CSci 3321: C++ and Data Structures (Algorithms and Data Structures I)

Huffman Encoding

If a great deal of text must be communicated, or if bandwidth is at a premium, it pays to compress the text in some manner before sending it. Huffman encoding does this by exploiting redundancy in the source text, for storage or communication.

Suppose we have an alphabet of symbols s_1, s_2, s_3 ..... s_n and the number of occurrences of the i^{th} symbol is f_i. Symbols can be both strings(keywords) or characters(letter). Usually, a computer represents a character by 8 bits using the ASCII code, for some keyword, such as 'int', it would have taken 24bits to represent. Instead, in Huffman code, each symbol s_i is represented by a binary string of length l_i. The total length L of an encoded string of length n will be the sum over all i, of (f_i * l_i). Assuming we have arranged the characters so that f_1 >= f_2 >= f_3 >= .... >= f_n, clearly L is minimized if and only if l_1 <= l_2 <= l_3 <= ..... <= l_n.

A Huffman code selects the binary codes for each symbol minimizing L. Suppose for example that the symbols A, BREAK, CIN, DO, E, FOR and GOTO appear with frequencies 25, 21, 18, 14, 9, 7 and 6 respectively. A Huffman tree for these symbols and their frequencies is a convenient visual representation of the binary strings which represent the symbols. Each node of the tree is labeled with the frequencies assigned to the nodes below it. Each symbol in the source alphabet lies at a terminal (leaf) node of the tree and the binary string which encodes it is the sequence of 0s and 1s we encounter in traversing the tree from the root to that node.

The Huffman code for this system is therefore:
[(A:00), (BREAK:10), (CIN:010), (DO:011), (E:111), (FOR:1100), (GOTO:1101)]