- Prim's and Kruskal's algorithms
- Shortest path in Weighted Graph [Dijkstra's]
- Coin change problem
- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height (or rank)
- Job scheduling algorithm
- Greedy techniques can be used as an approximation algorithm for complex problems

17.7 Understanding Greedy Technique

For better understanding let us go through an example.

Huffman Coding Algorithm

Definition

Given a set of *n* characters from the alphabet A [each character c ∈ A*]* and their associated frequency *freq*(c), find a binary code for each character c ∈ *A*, such that $\sum_{c \in A}$ freq(c)|binarycode(c)| is minimum, where /binarycode*(c)*/represents the length of binary code of character c. That means the sum of the lengths of all character codes should be minimum [the sum of each character's frequency multiplied by the number of bits in the representation].

The basic idea behind the Huffman coding algorithm is to use fewer bits for more frequently occurring characters. The Huffman coding algorithm compresses the storage of data using variable length codes. We know that each character takes 8 bits for representation. But in general, we do not use all of them. Also, we use some characters more frequently than others. When reading a file, the system generally reads 8 bits at a time to read a single character. But this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character ′e′ is used 10 times more frequently than the character ′*q*′. It would then be advantageous for us to instead use a 7 bit code for e and a 9 bit code for *q* because that could reduce our overall message length.

On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending on the character frequencies. The idea behind the character coding is to give longer binary codes for less frequent characters and groups of characters. Also, the character coding is constructed in such a way that no two character codes are prefixes of each other.

An Example

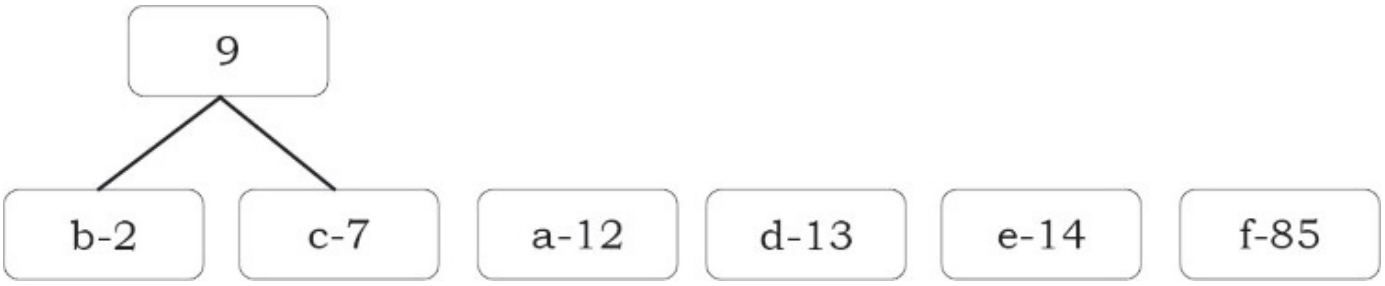
Let's assume that after scanning a file we find the following character frequencies:

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

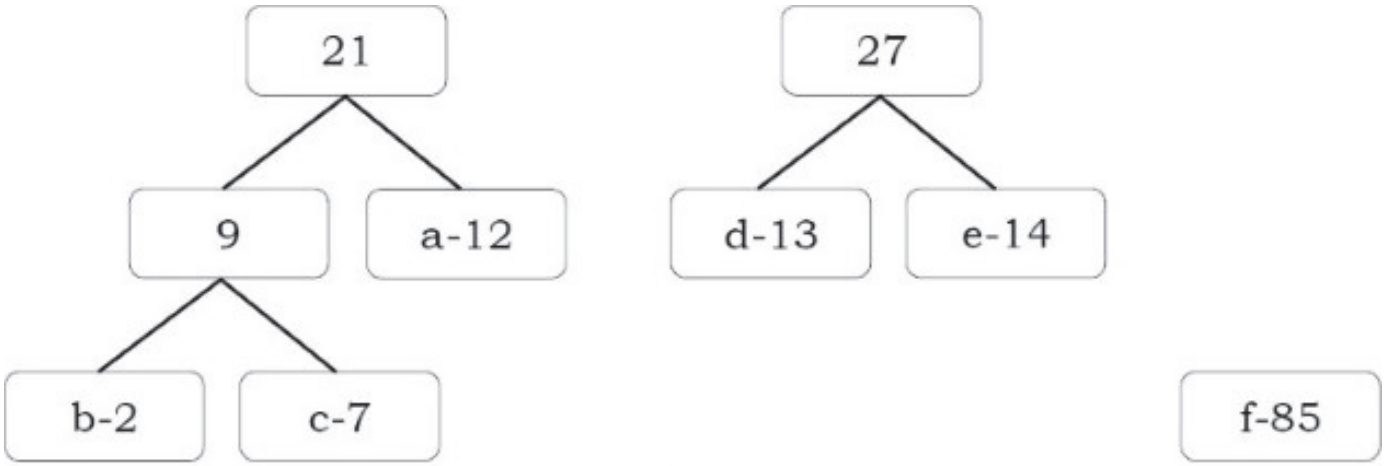
Given this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below).

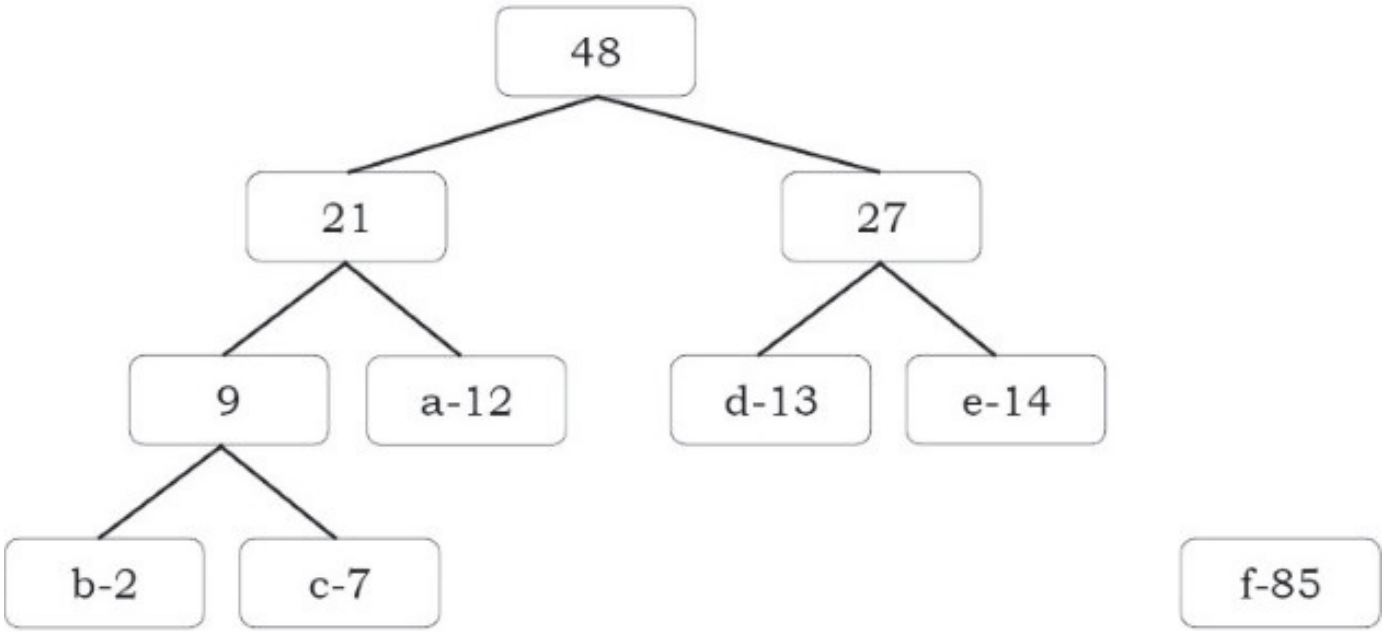
The algorithm works as follows: In the list, find the two binary trees that store minimum frequencies at their nodes.

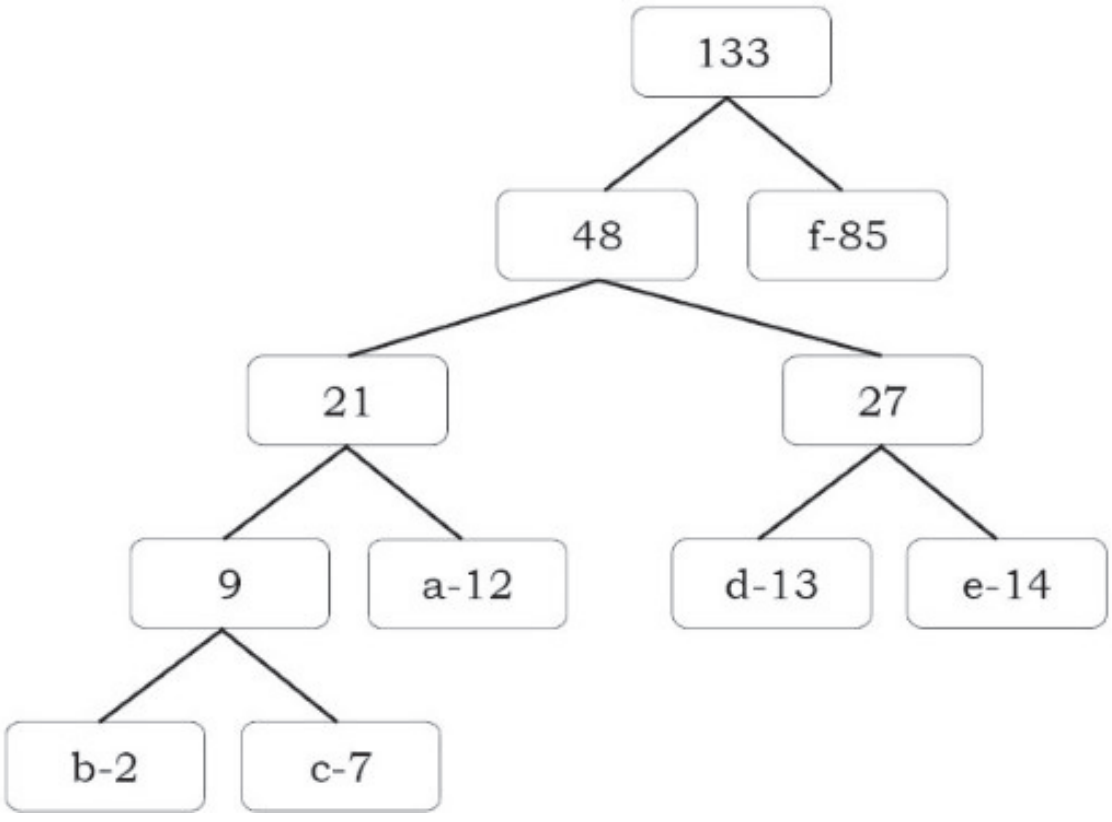
Connect these two nodes at a newly created common node that will store no character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like this:



Repeat this process until only one tree is left:


21
27
9
a-12
d-13
e-14
b-2
c-7
f-85


48
21
27
9
a-12
d-13
e-14
b-2
c-7
f-85


133
48
f-85
21
27
9
a-12
d-13
e-14
b-2
c-7

Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, traverse from the root to the leaf node. For each move to the left, append a 0 to the code, and for each move to the right, append a 1. As a result, for the above generated tree, we get the following codes:

Letter	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

Calculating Bits Saved

Now, let us see how many bits that Huffman coding algorithm is saving. All we need to do for this calculation is see how many bits are originally used to store the data and subtract from that the number of bits that are used to store the data using the Huffman code. In the above example, since we have six characters, let's assume each character is stored with a three bit code. Since there are 133 such characters (multiply total frequencies by 3), the total number of bits used is 3 * 133 = 399. Using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Thus, we saved 399 – 238 = 161 bits, or nearly 40% of the storage space.

```
HuffmanCodingAlgorithm(int A[], int n) {
    Initialize a priority queue, PQ, to contain the n elements in A;
    struct BinaryTreeNode *temp;
    for (i = 1; i<n; i++) {
        temp  = (struct *)malloc(sizeof(BinaryTreeNode));
        temp→left = Delete-Min(PQ);
        temp→right = Delete-Min(PQ);
        temp→data = temp→left→data + temp→right→data;
        Insert temp to PQ;
    }
    return PQ;
}
```

Time Complexity: O(*nlogn*), since there will be *one* build_heap, 2*n* – 2 delete_mins, and *n* – 2 inserts, on a priority queue that never has more than n elements. Refer to the *Priority Queues* chapter for details.

17.8 Greedy Algorithms: Problems & Solutions

Problem-1 Given an array *F* with size n. Assume the array content *F*[*i*] indicates the length of the i^{th} file and we want to merge all these files into one single file. Check whether the following algorithm gives the best solution for this problem or not?

Algorithm: Merge the files contiguously. That means select the first two files and merge them. Then select the output of the previous merge and merge with the third file, and keep going...

Note: Given two files *A* and *B* with sizes *m* and n, the complexity of merging is O(*m* + *n*).

Solution: This algorithm will not produce the optimal solution. For a counter example, let us consider the following file sizes array.

$$F = \{10,5,100,50,20,15\}$$

As per the above algorithm, we need to merge the first two files (10 and 5 size files), and as a result we get the following list of files. In the list below, 15 indicates the cost of merging two files with sizes 10 and 5.

$$\{15,100,50,20,15\}$$

Similarly, merging 15 with the next file 100 produces: {115,50,20,15}. For the subsequent steps