UNITED INTERNATRIONAL UNIVERSITY
Fall – 2023

CSE 2218 – Data Structure And Algorithms–II Laboratory
Section - J

Huffman coding

Presented To-

Presented By-

MD Sakhawat Hossain
011221240

Md Mostafizur Rahman 011221014

Huffman coding

- Formal Definition

> A technique that compress data to reduce its size without losing any of the details.

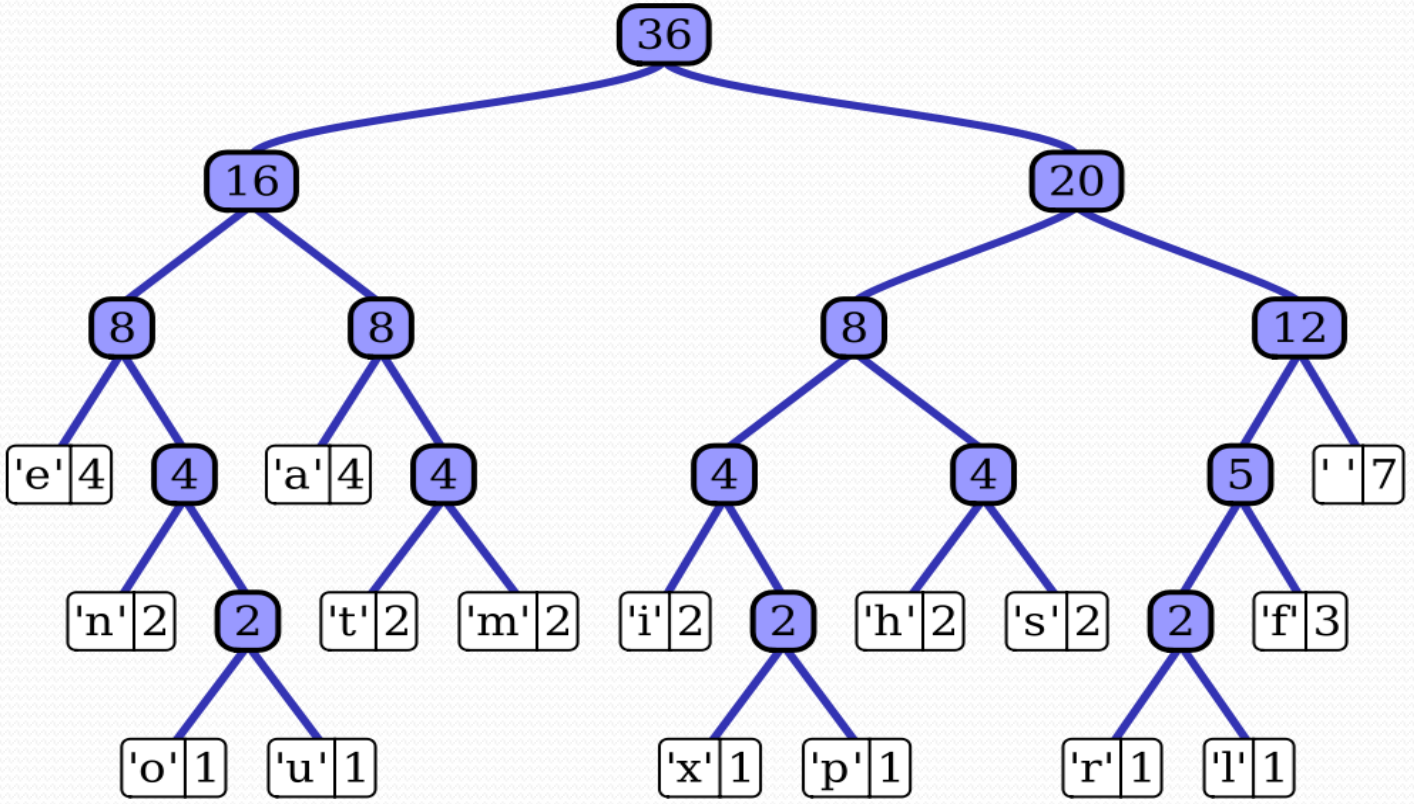
Huffman coding

- Formal Definition

A technique that compress data to reduce its size without losing any of the details.

Compress the data according to the frequency of occurring characters

Huffman coding



Let’s Consider

A String containing characters of-

AAABBBBCCCCCDDEEEF

Let’s Consider

A String containing characters of-

AAABBBBCCCCCDDEEEF

We know each character has its corresponding ASCII value with 8 bit length binary code

Let’s Consider

A String containing characters of-

AAABBBBCCCCCDDEEEF

We know each character has its corresponding ASCII value with 8 bit length binary code

So here there are 18 character

The length of the string by default

= **18*8= 144** bits

Let's Consider

A String containing characters of-

AAABBBBCCCCCDDEEEF

That's Huge! We can reduce it!

We know each character has its corresponding ASCII value with 8 bit length binary code

So here there are 18 character

The length of the string by default

= 18*8= 144 bits

HOW?

David Albert Huffman
introduced/developed a data compression method

David Albert Huffman (1925 – 1999)

- Introduced this method from the idea of using a frequency-sorted binary tree

Which one is Effective ?

Fixed length codes

or

Variable length codes

Which one is Effective ?

Fixed length codes?

Lets have a try!

Let's Call it again

A String containing characters of-

AAABBBBCCCCCDDEEEF

In fixed length code-

Character	frequency	Code [assume 3 bit]
A	3	000
B	4	001
C	5	010
D	2	001
E	3	100
F	1	101

length/char = 2^n here n=3, which represents minimum 2^3=8 different characters

Require = (6*8) + 18 + (18 * 3) = 120 bits

Let's Call it again

A String containing characters of-

AAABBBBCCCCCDDEEEF

We can reduce even more!

In fixed length code-

Character	frequency	Code [assume 3 bit]
A	3	000
B	4	001
C	5	010
D	2	001
E	3	100
F	1	101

length/char = 2^n here n=3, which represents minimum 2^3=8 different characters

Require = (6*8) + 18 + (18 * 3) = 120 bits

Which one is Effective ?

Fixed length codes

or

Variable length codes

Let' Try!

more effective??

Let's Call it again

A String containing characters of-

AAABBBBCCCCCDDEEEF

In Variable length code-

Character	frequency	Code [from tree]
A	3	111
B	4	01
C	5	10
D	2	1101
E	3	00
F	1	1100

Require=(3*3)+(2*4)+(5*2)+(2*4)+(3*2)+(1*4)+(6*8) +17
= 110 bits

Let's Call it again

A String containing characters of-

AAABBBBCCCCCDDEEEF

In Variable length code-

Character	frequency	Code [from tree]
A	3	111
B	4	01
C	5	10
D	2	1101
E	3	00
F	1	1100

Compressed!

Variable length codes Method (Huffman Coding) is more effective!

Require=(3*3)+(2*4)+(5*2)+(2*4)+(3*2)+(1*4)+(6*8) +17
= 110 bits

Basic Terminologies of **Huffman Coding**

Huffman Tree: Binary tree used to encode characters efficiently.

Codeword: Binary code assigned to each character in the input.

Frequency: Number of occurrences of a character in the input.

Prefix-free codes: Codes where no code is a prefix of another.
-the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol.

Bitwise representation: Binary representation of the Huffman-coded data.

Steps of Algorithm

1. create a priority queue Q consisting of each unique character.

2. sort them in ascending order of their frequencies.

3. for all the unique characters:
   - create a newNode
   - extract minimum value from Q and assign it to leftChild of newNode
   - extract minimum value from Q and assign it to rightChild of newNode
   - calculate the sum of these two minimum values and assign it to the value of newNode
   - insert this newNode into the tree

4. return rootNode

Let’s Call the string again

A String containing characters of-

AAABBBBCCCCCDDEEEF

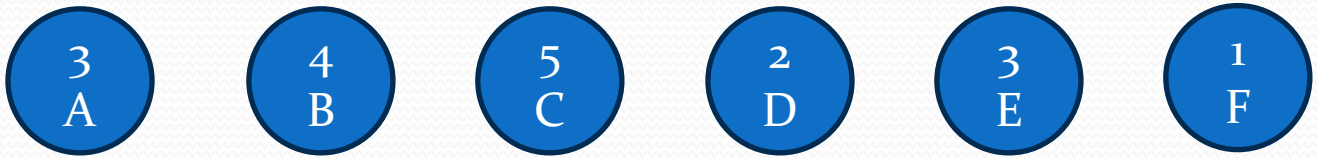
Step: 01

AAABBBBCCCCCDDEEEF

Make a Table:

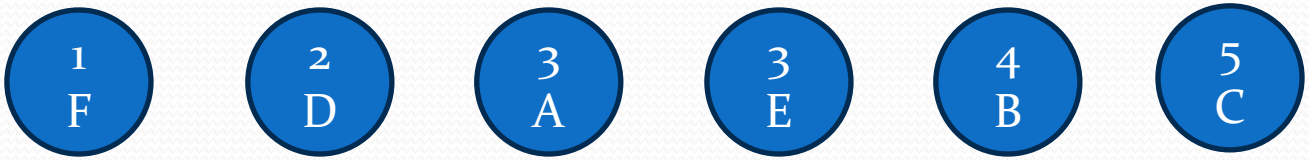
Character	*frequency*	*Binary Code*
A	3	
B	4	
C	5	
D	2	
E	3	
F	1	

Step: 02

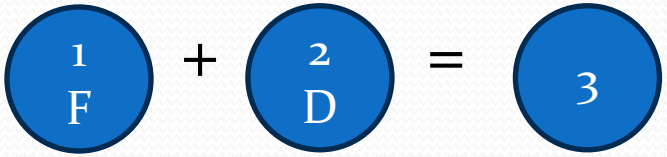


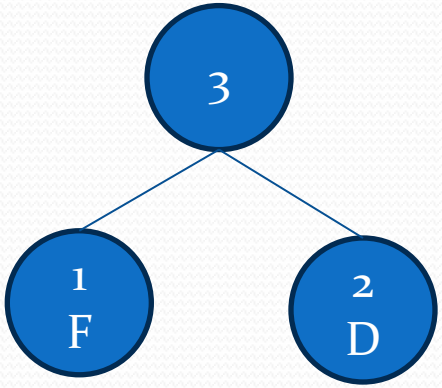
Step: 03

sort:



Merge:





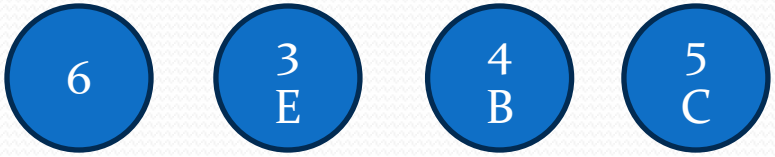
Now:

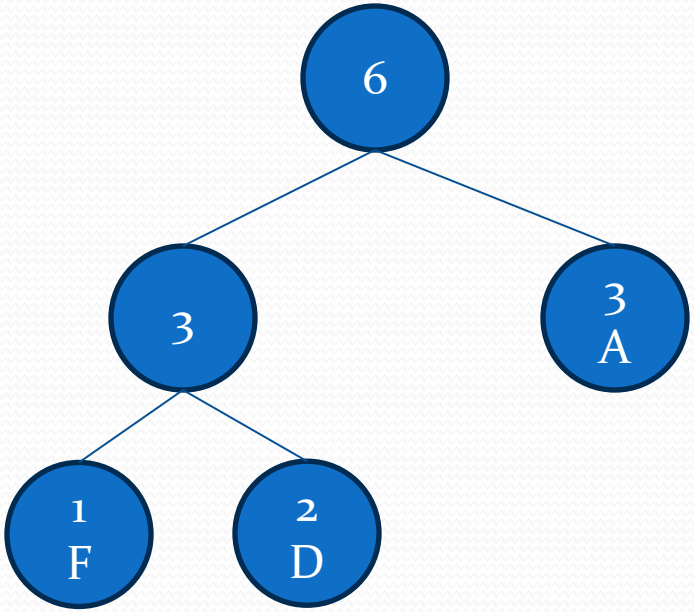
Step: 04

sort:

Merge:

Now:



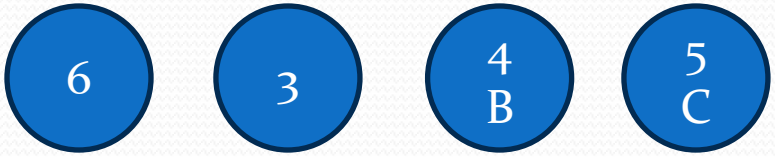


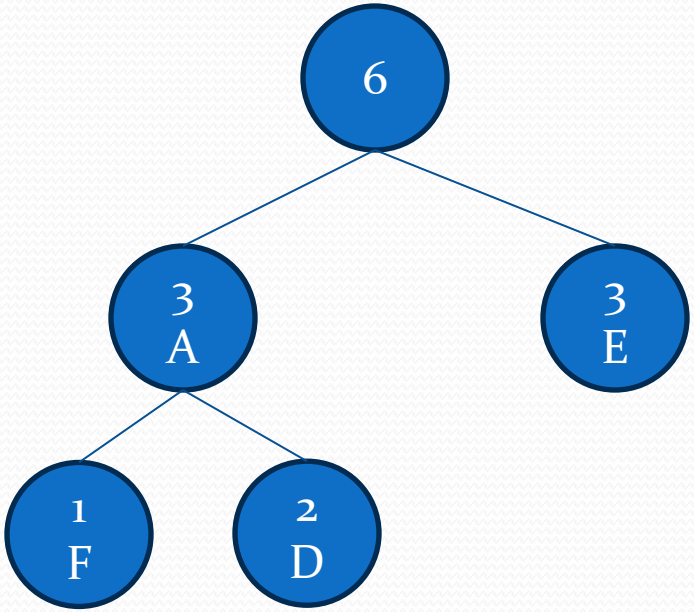
Step: 04 (Special Case)

sort:

Merge:

Now:



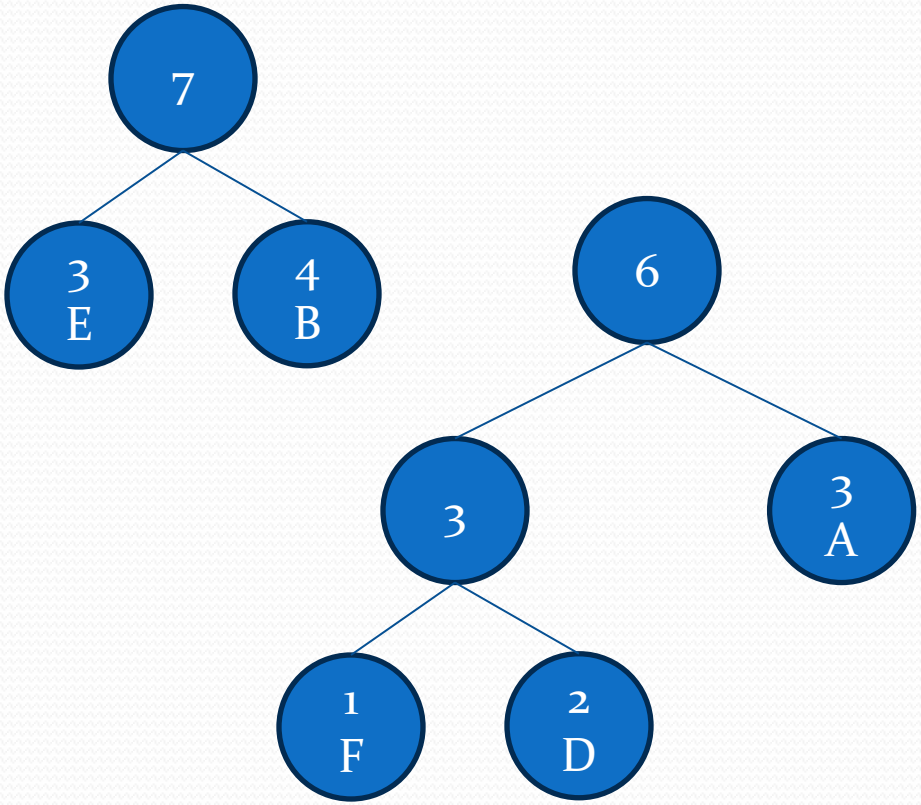


Step: 05

sort:

Merge:

Now:

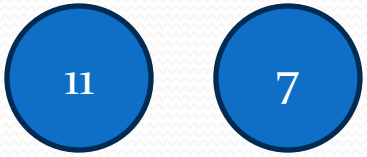


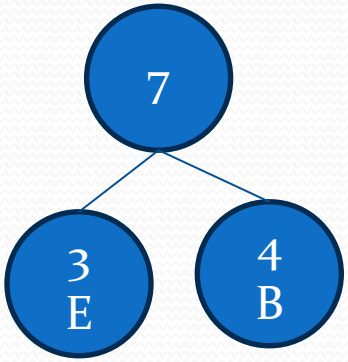
Step: 06

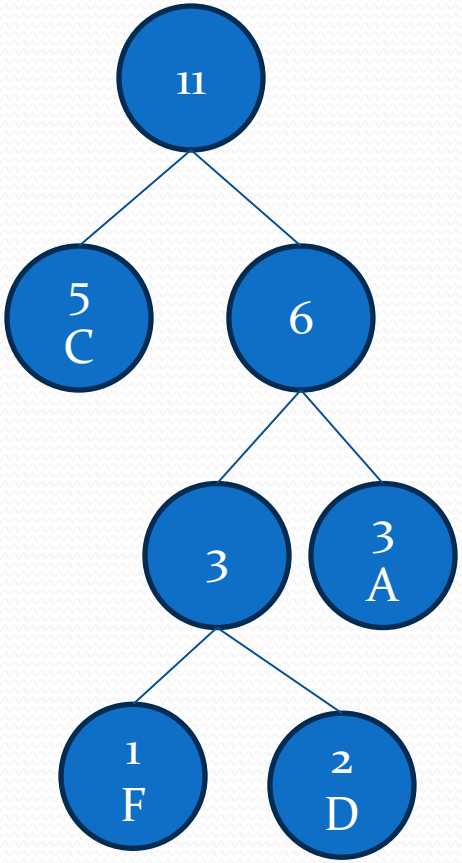
sort:

Merge:

Now:

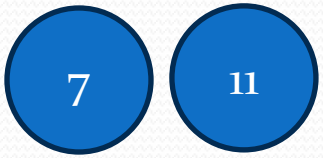






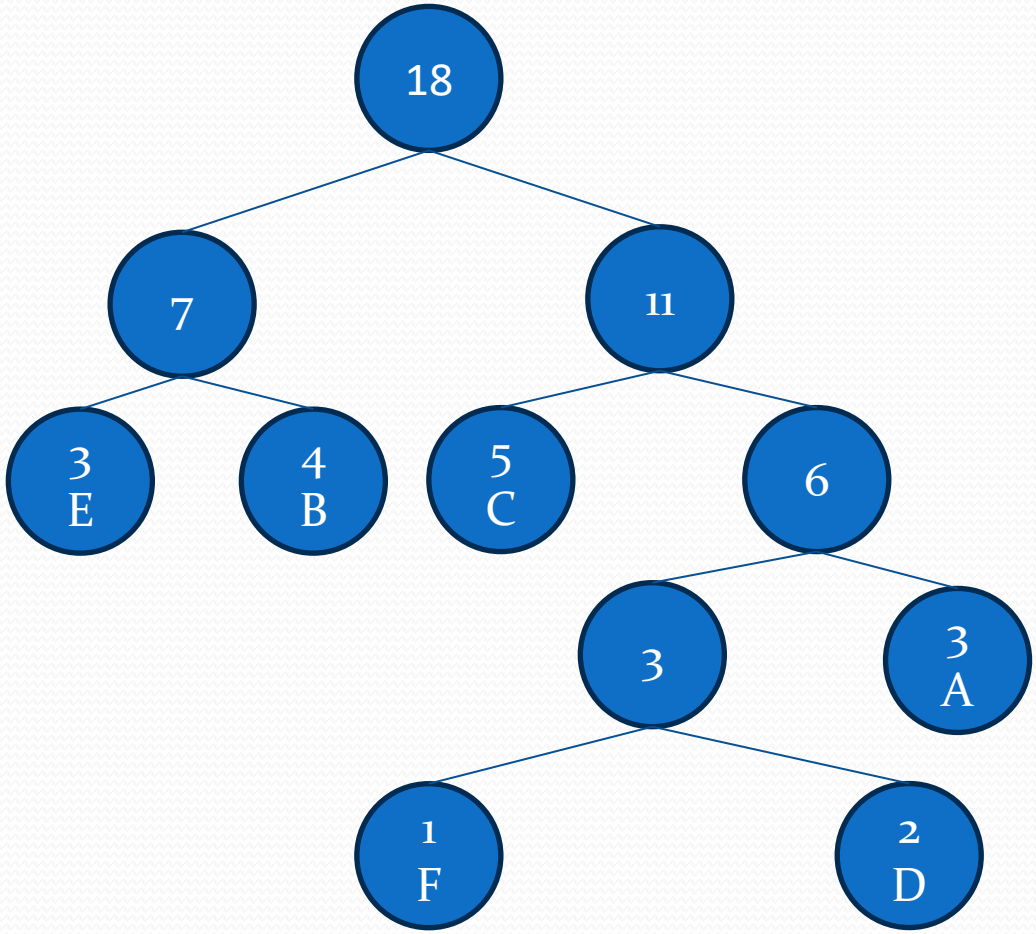
Step: 07

sort:

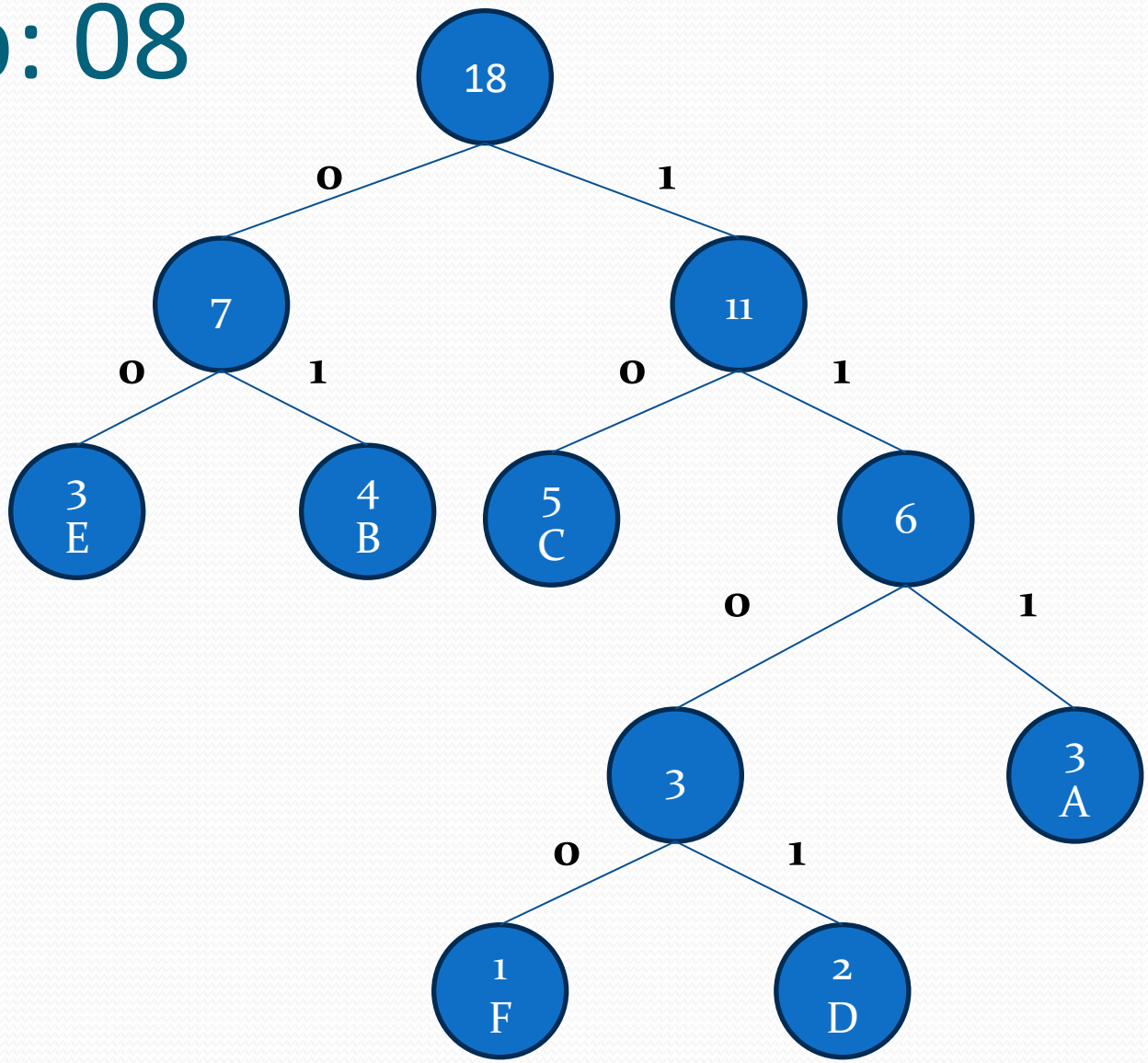


Merge:

Now:



Step: 08



Step: 09

Table:

Character	*frequency*	*Binary Code*
A	**3**	**111**
B	**4**	**01**
C	**5**	**10**
D	**2**	**1101**
E	**3**	**00**
F	**1**	**1100**

Calculation:

Character	*frequency*	*Code*	
A	3	111	3 * 3 = 9
B	4	01	2 * 4 = 8
C	5	10	2 * 5 = 10
D	2	1101	4 * 2 = 8
E	3	00	3 * 2 = 6
F	1	1100	4 * 1 = 4
8 * 6 = 48		Total bits = 17	Total bits = 45

Total bits = 48 + 17 + 45 = 110 bits

Difference:

String: AAABBBBCCCCCDDEEEF

Normal way:

Here,

Total Char = 18

Per Char bits = 8

Total bits = 18 * 8 = **144 bits**

By Huffman Coding:

Total bits = **110 bits**

Reduction: ((144 – 110) /144) 100 % = **23.61 %**

Example:

Word: FACE

Normal way:

01000110	01000001	01000011	01000101
F	**A**	**C**	**E**

Total bits = 4 * 8 = **32 bits**

By Huffman Coding:

1100	111	10	00
F	**A**	**C**	**E**

Total bits = **11 bits**

Reduction: ((32 – 11) /32) 100 % = **65.6 %**

Pseudocode:

```
Huffman_coding {
 n = |c|
 min_Heap = c
 for  i ← 1  to  n-1 {
        left = extractMin(min_Heap)
        right = extractMin(min_Heap)
        temp = newnode (left -> freq + right -> freq)
        temp –> left = left
        temp -> right = right
        insert_MinHeap (min_Heap, temp)
       }
   return exractMin(min_Heap)
}
```

Pseudocode:

```
binaryCode{
    If(root  - >  left){
       arr[top] = 0
       binaryCode(root -> left, arr, top+1)
   }
    If(root  - >  right){
       arr[top] = 1
        binaryCode(root -> right, arr, top+1)
   }
   If(isLeaf(root){
       print: root -> data
        PrintArr (arr,top)
   }
}
```

Time Complexity:

Extracting minimum frequency from the priority queue, **O(log n)**

Overall time complexity, **O(nlog n)**

Applications:

File Compression:

ZIP format

Image Compression:

JPEG format

Video Compression:

MPEG format

Network Communication:

Data Transmission

Database Compression:

Columnar Database Compression

Thank You!