Greedy Algorithms
Huffman Encoding

(Class 24)

From Book's Page No 431

Greedy Algorithm: Huffman Encoding

- The Huffman codes provide a method of encoding data efficiently.
- Normally, when characters are coded using standard codes like ASCII.
- Each character is represented by a fixed-length codeword of bits, e.g., 8 bits per character.

- Fixed-length codes are popular because it is very easy to break up a string into its individual characters,
- And to access individual characters and substrings by direct indexing.
- However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

- Consider the string “ abacdaacac”.
- If the string is coded with ASCII codes, the message length would be:

$$10 \times 8 = 80\ bits$$

- We will see that the same string encoded with a variable length Huffman encoding scheme will produce a shorter message.

Priority Queues

- Huffman Encoding uses “Binary Tree” and “Minimum Priority Queue” in its implementation.
- Priority queue is a type of queue in which every element has a key associated to it.
- And the queue returns the element according to these keys.
- Unlike the traditional queue which works on first come first serve basis.

- A max-priority queue returns the element with maximum key first.
- A min-priority queue returns the element with the smallest key first.

- Heaps are great for implementing a priority queue because the largest/smallest element is at the root of the tree.
- We use a max-heap for a max-priority queue and a min-heap for a min-priority queue.

- There are mainly 4 operations we want from a priority queue:
  - **Insert:** To insert a new element in the queue.
  - **Maximum/Minimum:** To get the maximum and the minimum element from the max-priority queue and min-priority queue respectively.
  - **Extract Maximum/Minimum:** To remove and return the maximum and the minimum element from the max-priority queue and min-priority queue respectively.
  - **Increase/Decrease Key:** To increase or decrease the key of any element in the queue.

Huffman Encoding Algorithm

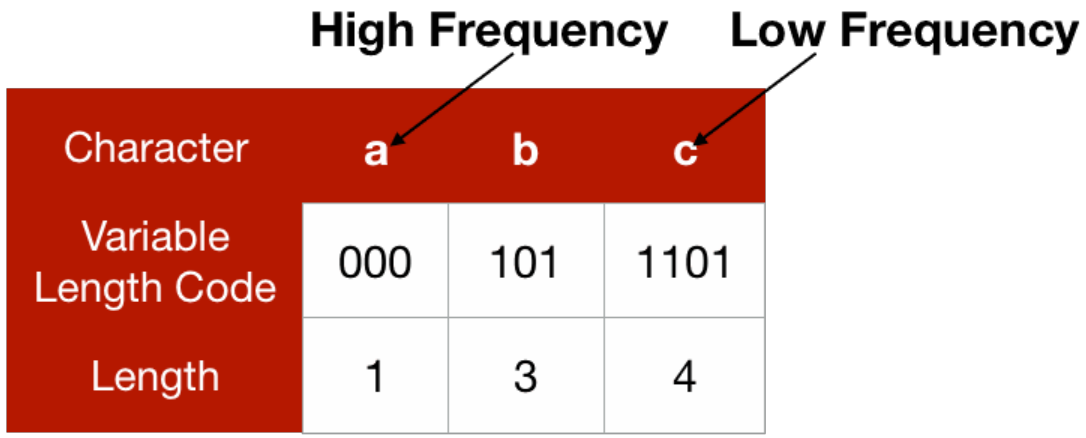
- Huffman code is a data compression algorithm which uses the greedy technique for its implementation.
- The algorithm is based on the frequency of the characters appearing in a file.
- Our files are stored as binary code in a computer and each character of the file is assigned a binary character code and normally, these character codes are of fixed length for different characters.

- For example, if we assign 'a' as 000 and 'b' as 001, the length of the codeword for both the characters are fixed i.e., both 'a' and 'b' are taking 3 bits.

Character	a	b	c
Code	000	001	010
Length	3	3	3

- Huffman code doesn't use fixed length codeword for each character and assigns codewords according to the frequency of the character appearing in the file.
- Huffman code assigns a shorter length codeword for a character which is used a greater number of time (or has a high frequency).

- And a longer length codeword for a character which is used a smaller number of times (or has a less frequency).



Character	a	b	c
Variable Length Code	000	101	1101
Length	1	3	4

- Since characters which have high frequency has lower length, they take less space and save the space required to store the file.

- Let's take an example.

Character	a	b	c	d	e	f
Fixed Length Code	000	001	010	011	100	101
Frequency	51	20	2	3	9	15
Variable Length Code	0	111	1100	1101	100	101

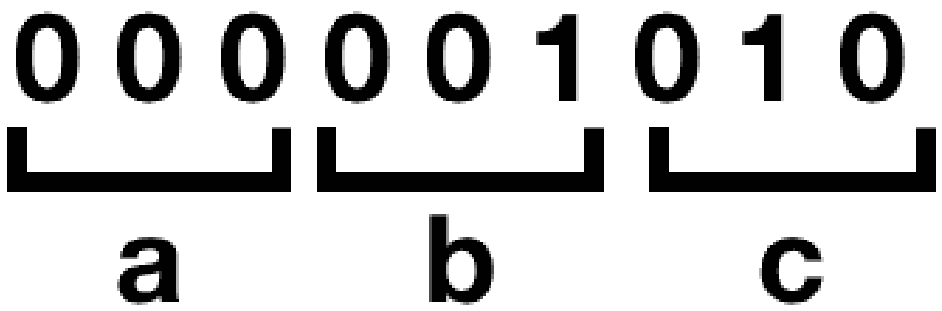
- In this example, 'a' appears 51 out of 100 times and has the highest frequency, 'c' is appearing only 2 out of 100 times and has the least frequency.
- Thus, we are assigning 'a' the codeword of the shortest length i.e., 0 and 'c' a longer one i.e., 1100.

- Now if we use characters of fixed length, we need 100*3 = 300 bits (each character is taking 3 bit) to represent 100 characters of the file.
- But to represent 100 characters with the variable length character, we need 51*1 + 20*3 + 2*4 + 3*4 + 9*3 + 15*3 = 203 bits.
- 51*1 as 'a' appears 51 out of 100 times and has length 1 and so on.
- Thus, we can save 32% of space by using the codeword for variable length in this case.

Storing, Encoding and Decoding

- We basically concatenate characters while storing them into a file.
- For example, to store 'abc', we would use 000.001.010 i.e., 000001010 using a fixed character codeword.
- Now for the decoding, we know that all of our characters are 3 bits long, so we would break the code for every 3 bits.

- And we can easily get 000, 001 and 010 which can be translated back to 'abc'.

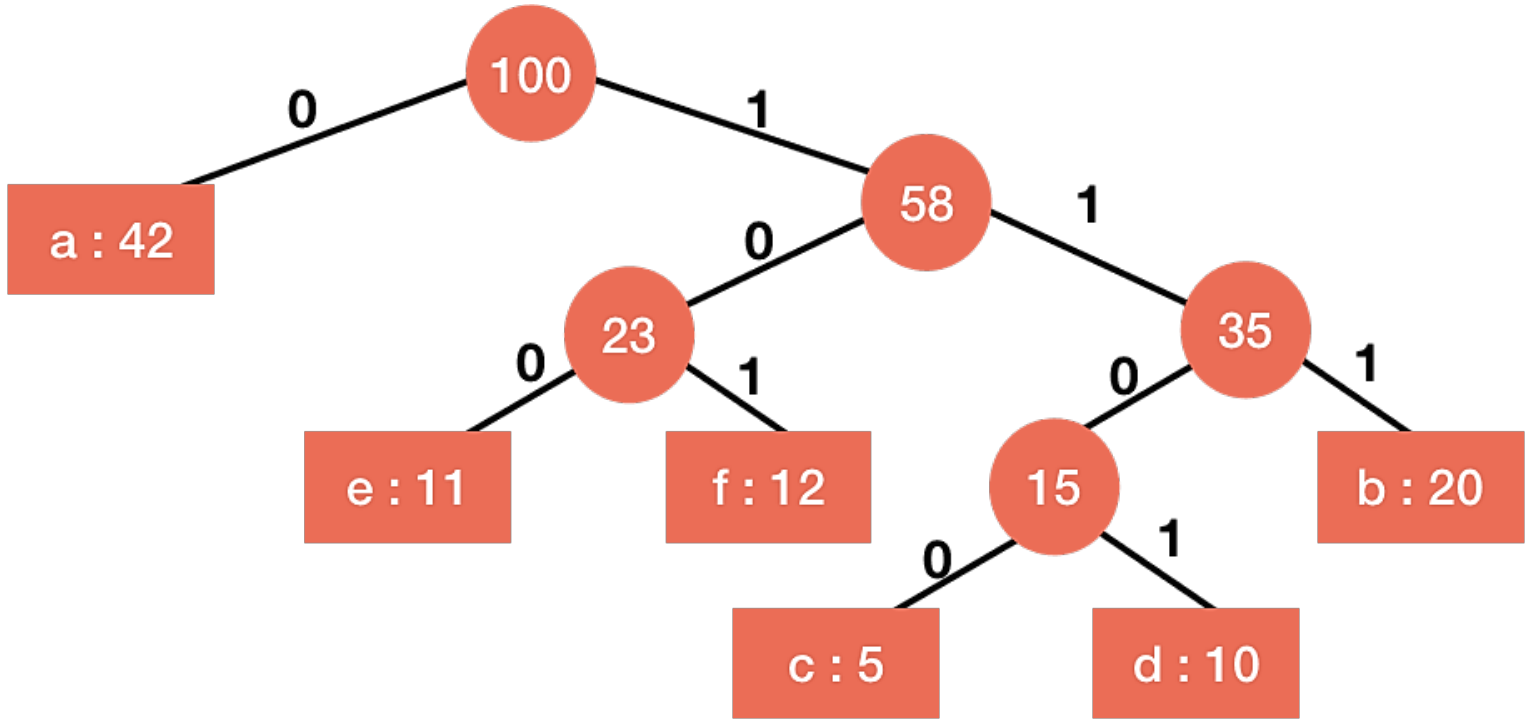


Prefix Codes

- In variable length codeword, we only use such code which are not the prefix of any other character, and these codes is known as *prefix codes*.
- For example, if we use 0, 1, 01, 10 to represent 'a', 'b', 'c' and 'd' respectively (0 is prefix of 01 and 1 is prefix of 10).
- Then the code 00101 can be translated into 'aabab', 'acc', 'aadb', 'acab' or 'aabc'.
- To avoid this kind of ambiguity, we use prefix codes.

- Using prefix codes made decoding unambiguous.
- In the above example, we have used 0, 111 and 1100 for 'a', 'b' and 'c' respectively.
- None of the code is the prefix of any other codes and thus any combination of these codes will decode into unique value.
- For example, if we write 01111100, it will uniquely decode into 'abc' only.

Implementing Huffman Code



- We construct this type of binary tree from the frequencies of the characters given to us.
- Let's focus on using this binary tree first and then we will learn about its construction.

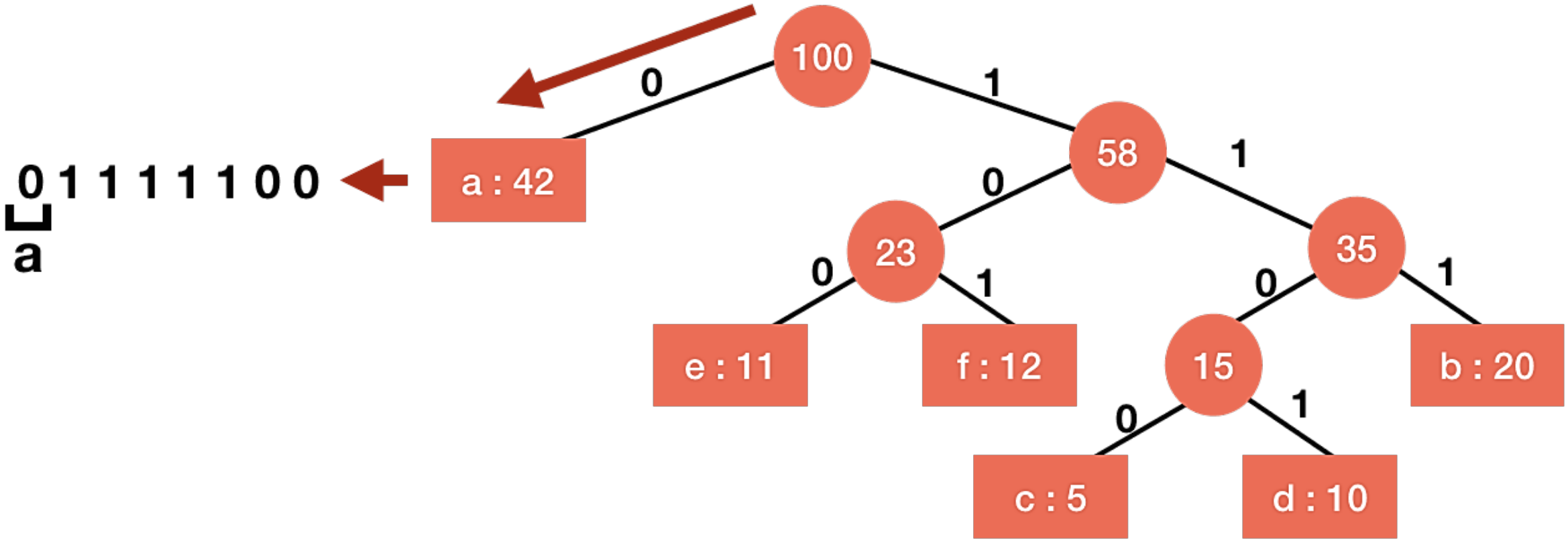
Decoding

- All the characters are on the leaves of the tree.
- To get the codeword for any character, we start from the root of the tree and proceed to that character.
- Now, if we move right from any node, we interpret that movement as 1 and if left, then 0.
- So, we move from root to the leaf containing that character and combining the 0s and 1s of each movement, we get the codeword of the character.

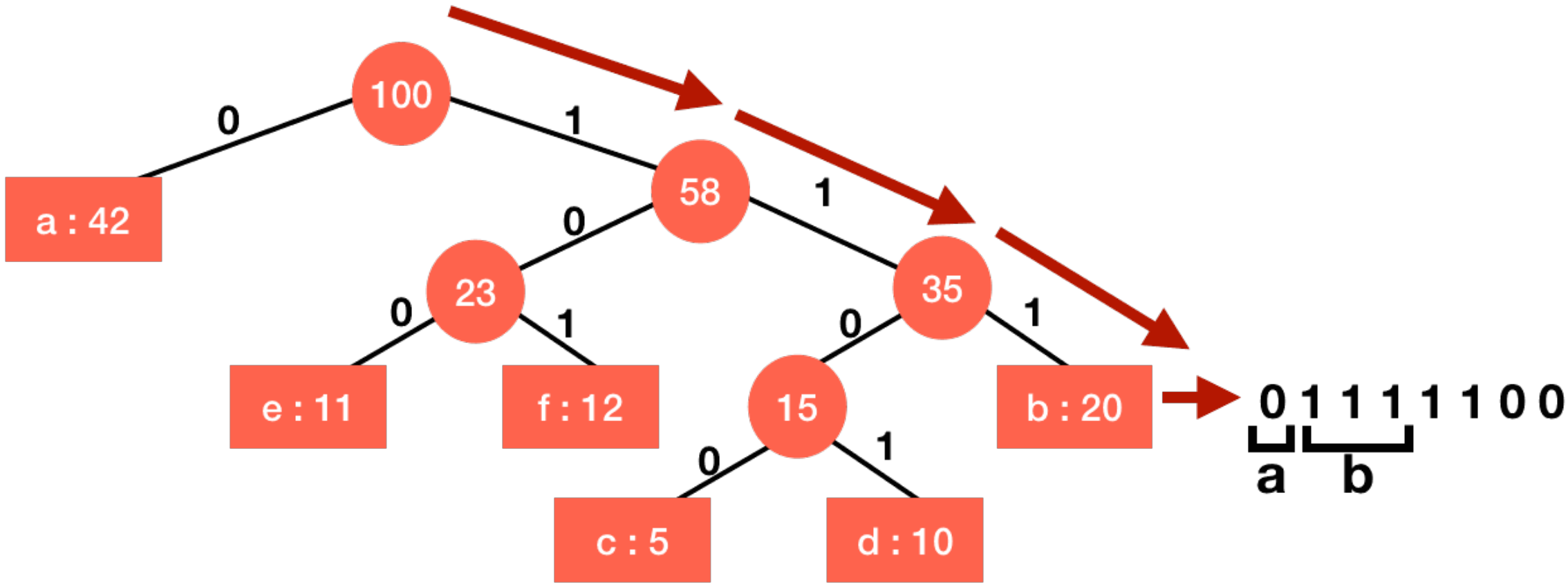
- In this way, we can get the code of each character.

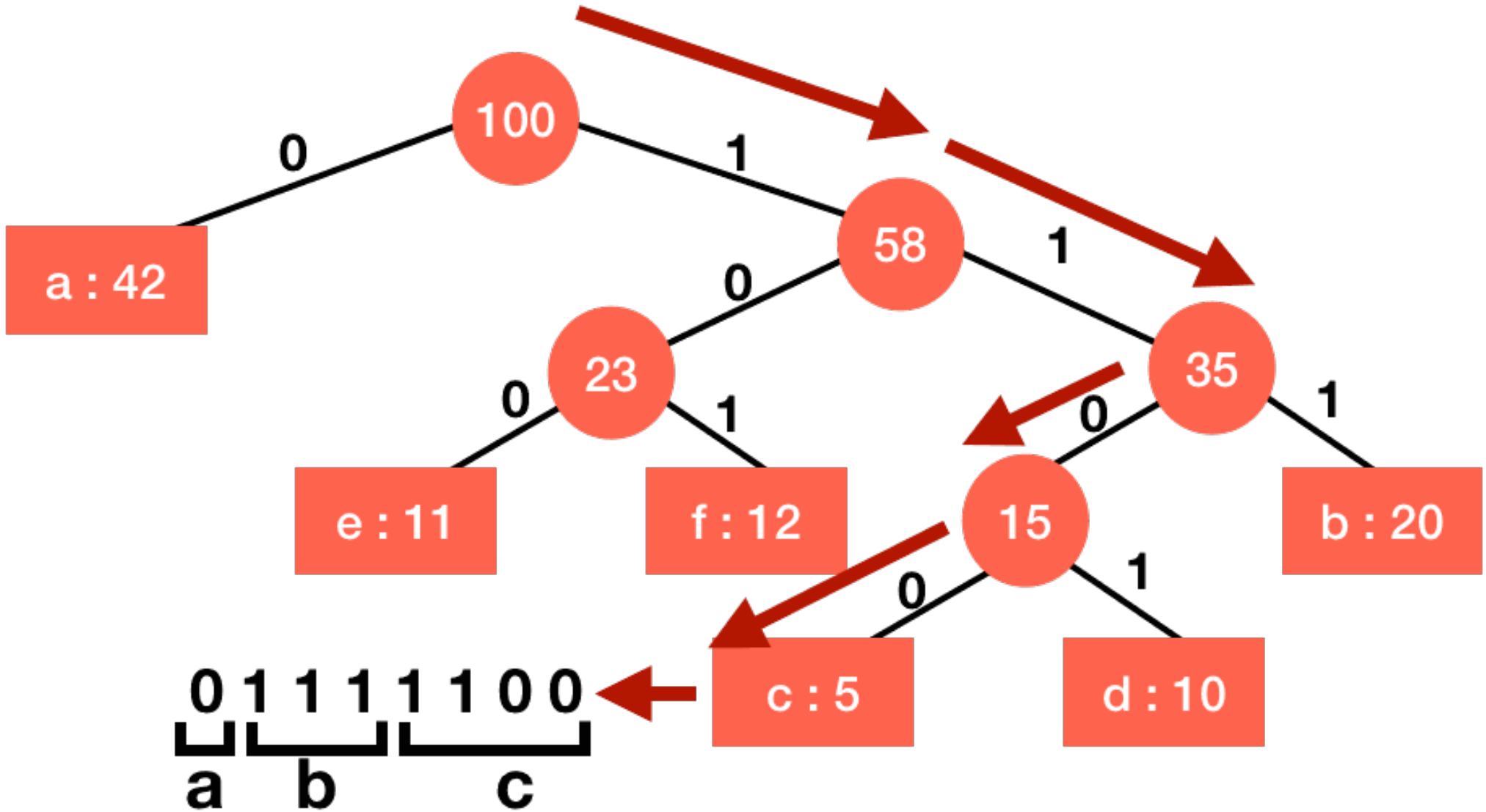
Character	Frequency	Code
a	12	0
b	20	111
c	5	1100
d	10	1101
e	11	100
f	12	101

- We can proceed in a similar way with any code given to us to decode it.
- For example, let's take a case of 01111100.
- We will start from the root and since the first number is 0, so we will move left.
- By moving left, we encountered a character 'a' and thus the first character is a.


0 1 1 1 1 1 0 0
a
100
0
1
a : 42
58
0
1
23
35
0
1
0
1
e : 11
f : 12
15
b : 20
0
1
c : 5
d : 10

- Now, we will again start from the root, we have 1 in the code, so we will move right.
- Since we have not reached any of the leaves, we will continue it.
- The next number is also 1 and so we will again move right.
- Still, we have not reached the leaf, so continuing, the next number is also 1.
- Moving right this time will give us the character 'b'.
- Thus, 'ab' is the string we have decoded till now.


100
0
1
a : 42
58
0
1
23
35
0
1
0
1
e : 11
f : 12
15
b : 20
0
1
c : 5
d : 10
0 1 1 1 1 1 0 0
a
b


100
0
1
a : 42
58
0
1
23
35
0
1
0
1
e : 11
f : 12
15
b : 20
0
1
c : 5
d : 10
0 1 1 1 1 1 0 0
a
b
c

- Thus, we have decoded 01111100 into 'abc'.
- We now know how to decode for Huffman code.
- Let's look at the encoding process now.

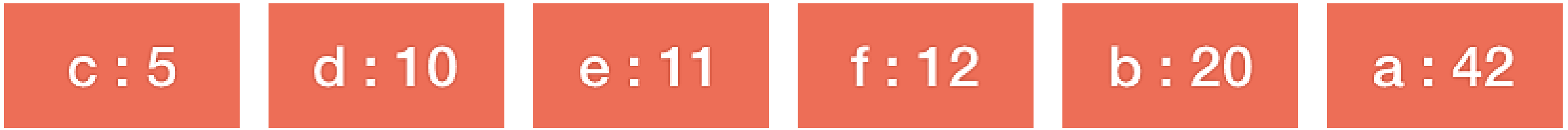
Encoding

- As stated above, encoding is simple.
- We just have to concatenate the code of the characters to encode them.
- For example, to encode 'abc', we will just concatenate 0, 111 and 1100 i.e., 01111100.
- But how to create this binary tree for the frequencies we have.

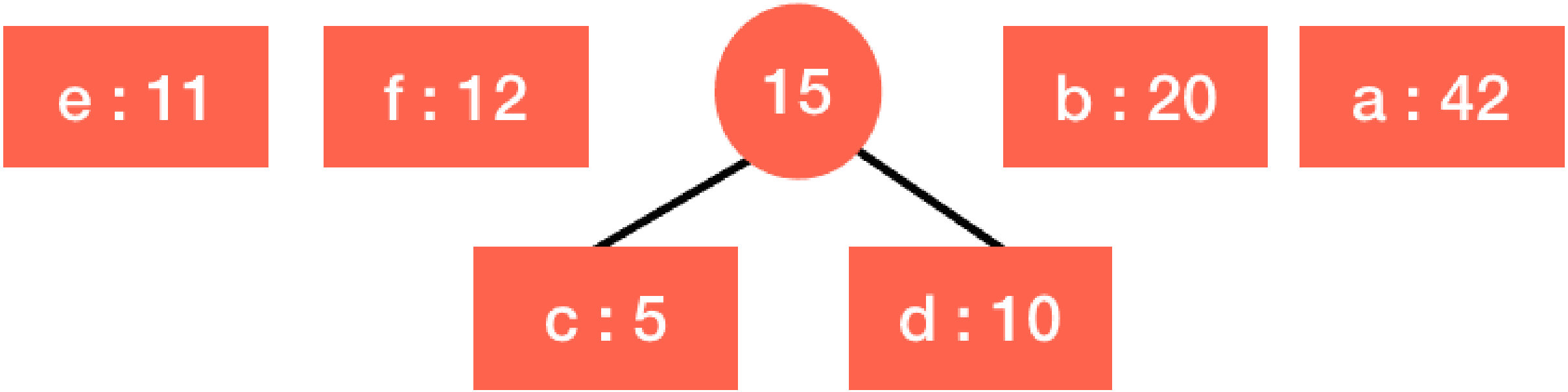
Construction of Binary Tree for Huffman Code

- This is the part where we use the greedy strategy.
- Basically, we have to assign shorter code to the character with higher frequency and vice-versa.
- We can do this in different ways and that will result in different trees, but a full binary tree gives us the optimal code i.e., using that code will save the maximum space in storing the file.

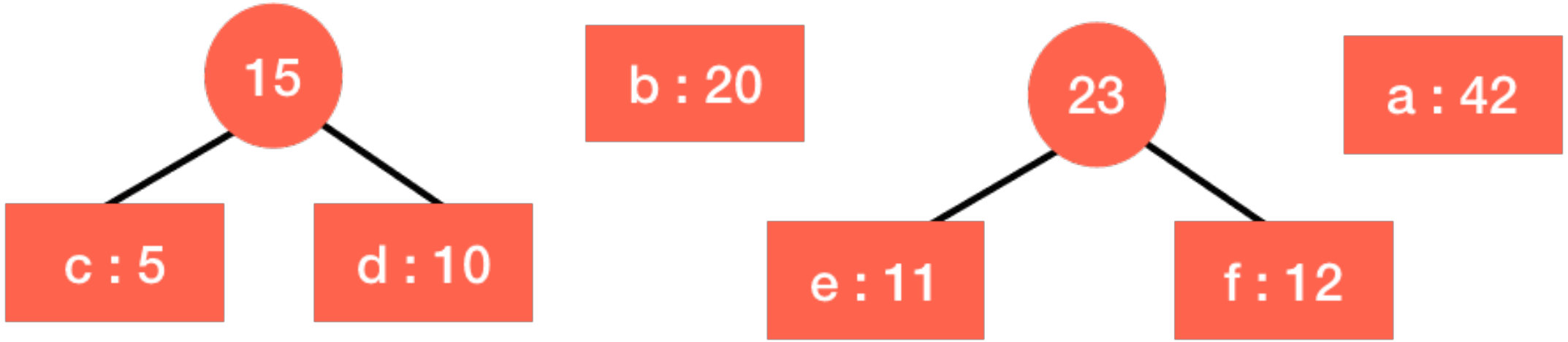
- To construct this tree for optimal prefix code, Huffman derived a greedy algorithm which we are going to use for the construction of the tree.
- We start by sorting the characters according to their frequencies.

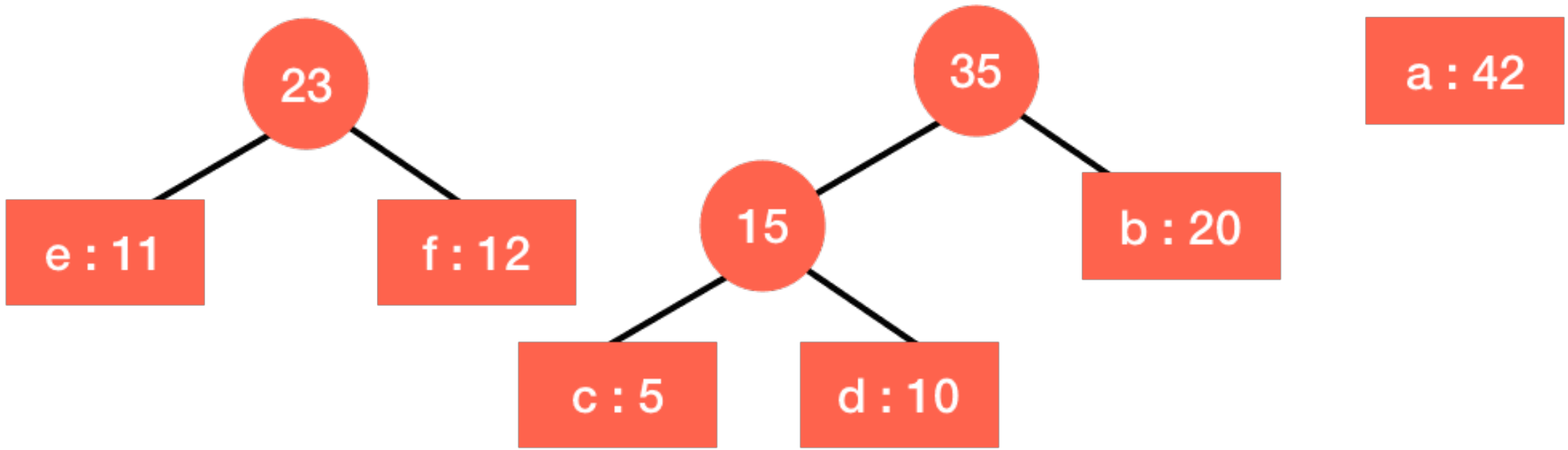


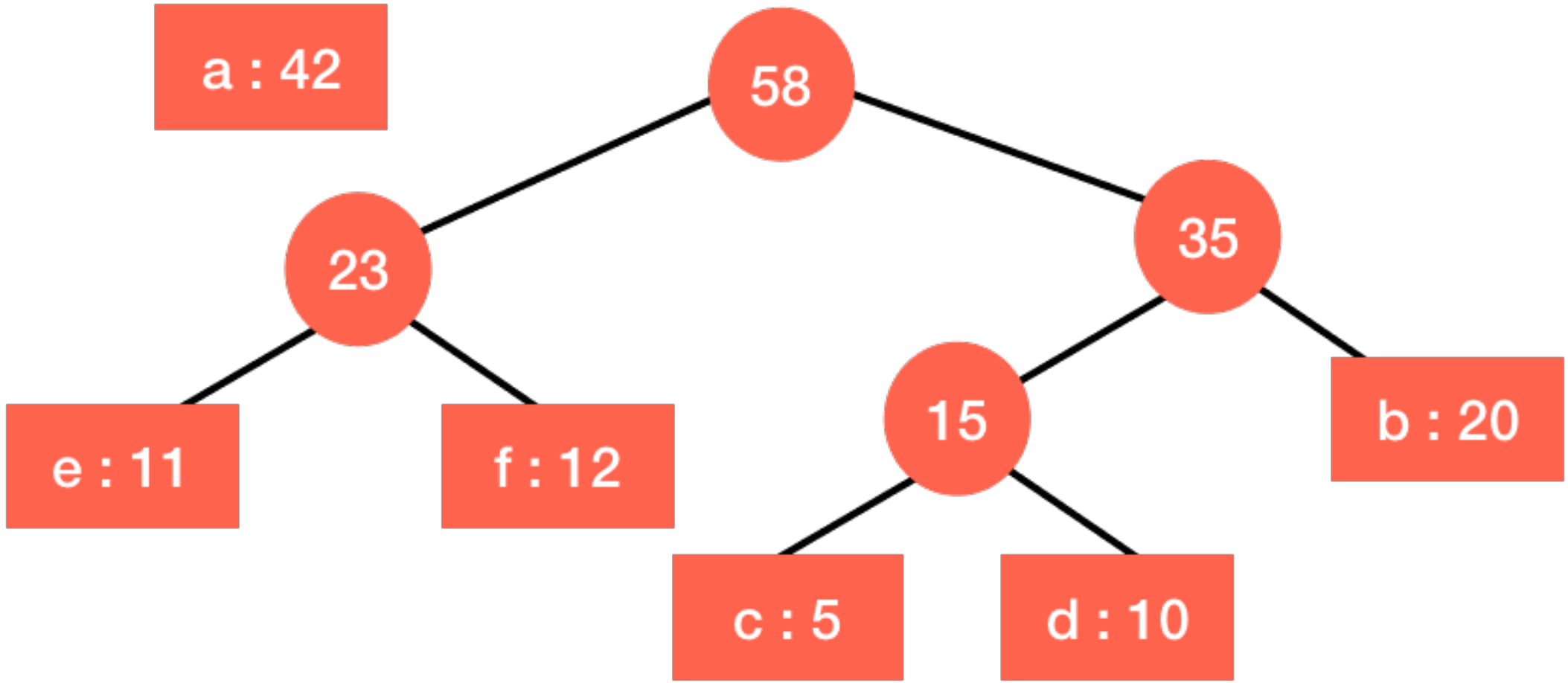
- Now, we make a new node and then greedily pick the first two nodes from the sorted character.
- And make the first node left child of the new node and second node as the right child of the new node.
- The value of the new node is the summation of the values of the children nodes.

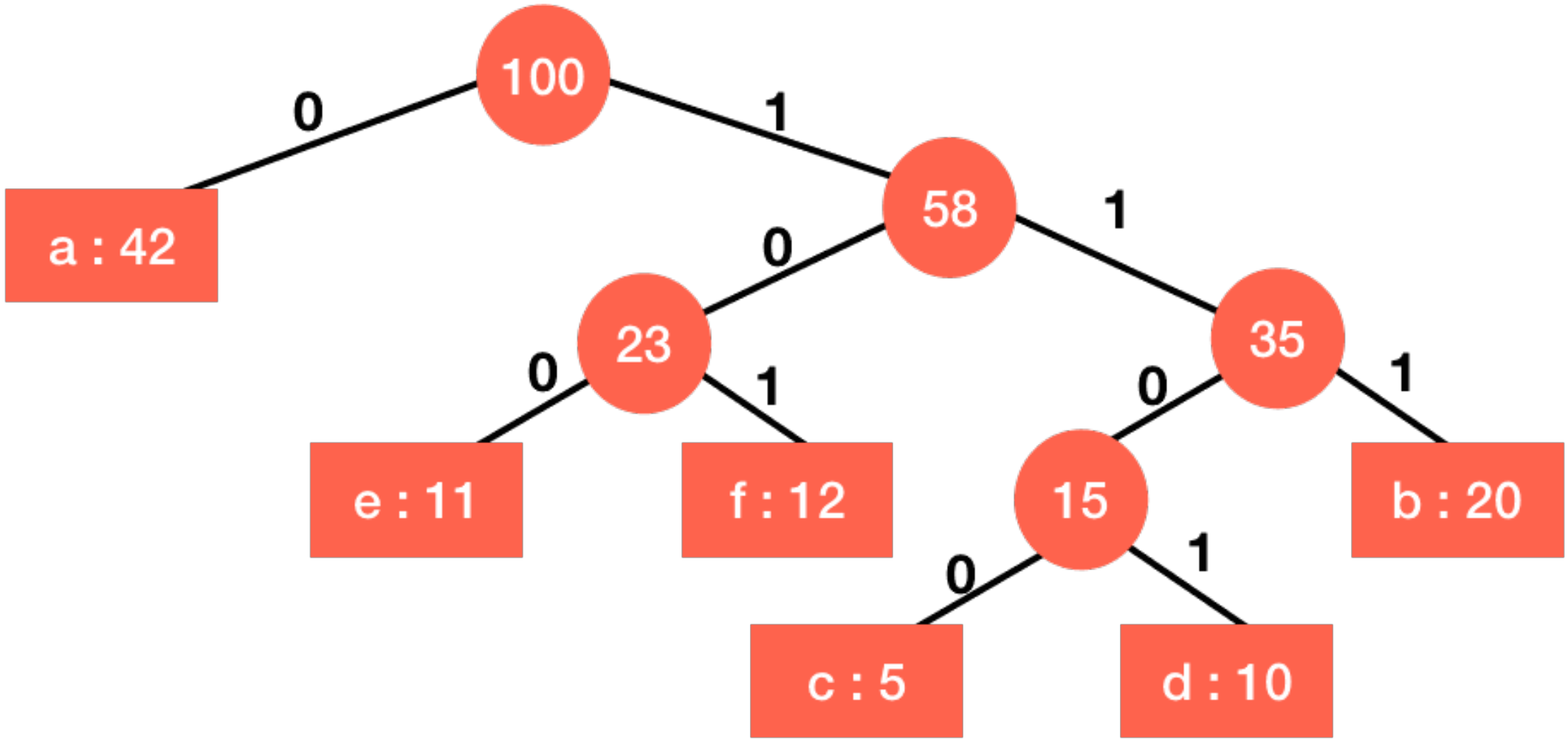

e : 11
f : 12
15
b : 20
a : 42
c : 5
d : 10

- We again sort the nodes according to the values and repeat the same process with the first two nodes.

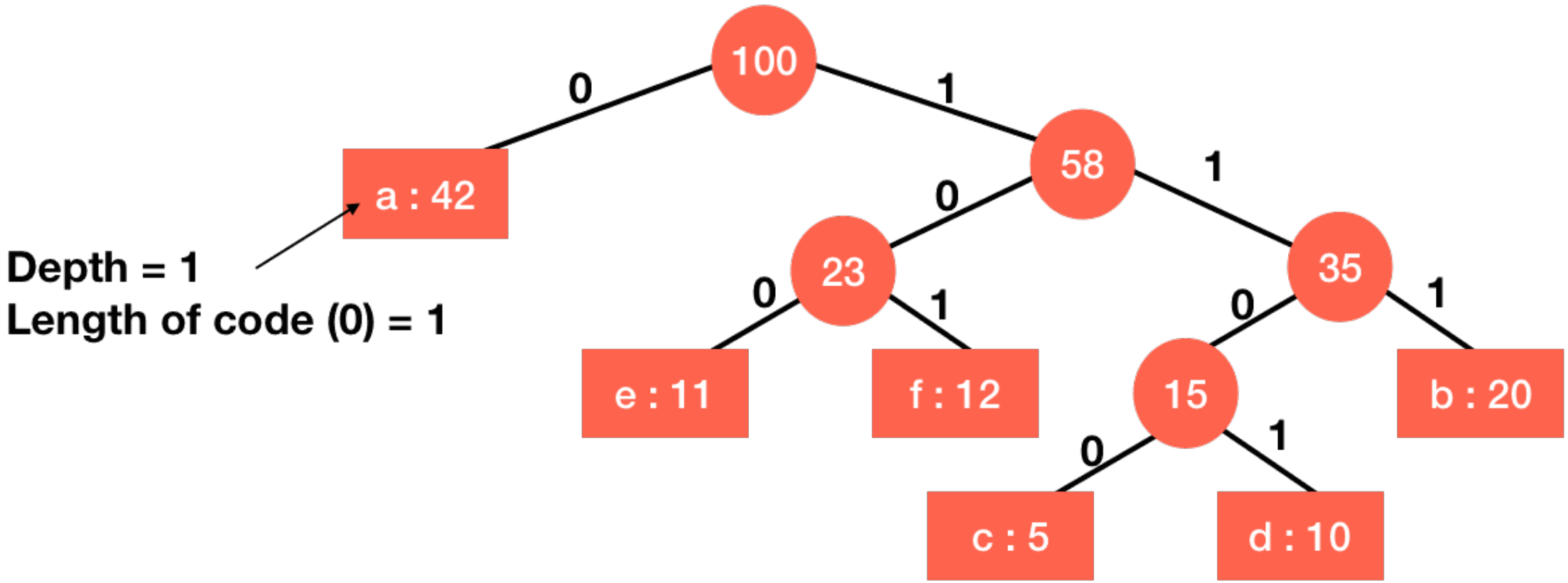



23
e : 11
f : 12
35
15
b : 20
c : 5
d : 10
a : 42


a : 42
58
23
35
e : 11
f : 12
15
b : 20
c : 5
d : 10


100
0
1
a : 42
58
0
1
23
35
0
1
0
1
e : 11
f : 12
15
b : 20
0
1
c : 5
d : 10

- In this way, we construct the tree for optimal prefix code.
- We can see that the depth of a leaf is also the length of the prefix code of the character.
- So, the depth d_i of leaf i is the length of the prefix code of the character at leaf i.
- If we multiply it by its frequency i.e., $d_i * i.freq$ then this is the number of bits used by this character in the entire file.


Depth = 1
Length of code (0) = 1
100
0
1
a : 42
58
0
1
23
35
0
1
0
1
e : 11
f : 12
15
b : 20
0
1
c : 5
d : 10

- We can sum all these for each character to get the total number of bits used to store the file.

$$Total\ bits = \sum_{i} d_i * i.freq$$

- The Huffman tree building algorithm is:
  1. Place the elements into minimum heap (by frequency).
  2. Remove the first two elements from the heap.
  3. Combine these two elements into one.
  4. Insert the new element back into the heap.

- Here is the Huffman tree building algorithm pseudocode is:

```
HUFFMAN (N, symbol[1..N], freq[1..N])
1  for i ← 1 to N
2    t ← TreeNode(symbol[i], freq[i])
3    pq.insert(t, freq[i])      // priority queue
4  for i ← 1 to N – 1
5    x ← pq.remove()
6    y ← pq.remove()
7    z ← new TreeNode
8    z.left ← x
9    z.right ← y
10   z.freq ← x.freq + y.freq
11   pq.insert(z, z.freq)
12 return pq.remove()          // root
```

Prefix Property

- The codewords assigned to characters by the Huffman algorithm have the property that no codeword is a prefix of any other.
- For example, the string “abacdaacac”:

character	a	b	c	d
frequency	5	1	3	1
probability	0.5	0.1	0.3	0.1
codeword	0	110	10	111

- The prefix property is evident by the fact that codewords are leaves of the binary tree.
- It means the prefix property is achieved due to the full binary tree.
- Decoding a prefix code is simple.
- We traverse the root to the leaf letting the input 0 or 1 tell us which branch to take.

Expected Encoding Length

- If a string of n characters over the alphabet $C = \{a, b, c, d\}$ is encoded using 8-bit ASCII, the length of encoded string is $8n$.
- For example, the string “abacdaacac” will require $8 \times 10 = 80$ bits.
- The same string encoded with Huffman codes will yield:

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

- This is just 17 bits, a significant saving!
- For a string of n characters over this alphabet, the expected encoded string length is:

$$n(0.5 \cdot 1 + 0.1 \cdot 3 + 0.3 \cdot 2 + 0.1 \cdot 3) = 1.7n$$

- In general, let $p(x)$ be the probability of occurrence of a character, and let $d_T(x)$ denote the length of the codeword relative to some prefix tree T.
- The expected number of bits needed to encode a text with n characters is given by:

$$\mathrm{B(T)} = \mathrm{n} \sum_{x \in \mathrm{C}} \mathrm{p(x).\, d_T(x)}$$

Huffman Encoding: Correctness

- Here correctness means:
  - Huffman encoding generates optimal solution.
  - And also generate unique codes solving prefix codes problem.

- Huffman algorithm uses a greedy approach to generate a prefix code T that minimizes the expected length $B(T)$ of the encoded string.

- In other words, Huffman algorithm generates an optimum prefix code.
- The question that remains is that: is the algorithm correct?
- Recall that the cost of any encoding tree T is:

$$\mathrm{B(T)} = \mathrm{n} \sum_{x \in \mathrm{C}} \mathrm{p(x).\, d_T\,(x)}$$

- Our approach to prove the correctness of Huffman Encoding will be to show that:
  - Any tree that differs from the one constructed by Huffman algorithm can be converted into one that is equal to Huffman's tree without increasing its costs (e.g., storage cost).

- Note that the binary tree constructed by Huffman algorithm is a full binary tree.