Today’s topics

- Decision tree
- Sorting
- Huffman encoding

Huffman encoding

Huffman codes

- Read section-16.3 from the Cormen et al. book

Huffman codes

- Huffman codes are used to compress data without losing any information

Huffman codes

- Suppose we want to store 100,000 characters in a file
  - What will be the size of the file?

Huffman codes

- Suppose we want to store 100,000 characters in a file
  - What will be the size of the file?
    - 100000 * 8 bits

- What if the characters used in the file belong to the set (a,b,c,d,e,f)
  - Can we do better?

Huffman codes

- Suppose we want to store 100,000 characters in a file
  - What will be the size of the file?
    - 100000 * 8 bits

- What if the characters used in the file belong to the set (a,b,c,d,e,f)
  - Can we do better?
    - Yes, we can use 3-bit encoding for characters
  - How many bits will be required?

Huffman codes

- Suppose we want to store 100,000 characters in a file
  - What will be the size of the file?
    - 100000 * 8 bits
- What if the characters used in the file belong to the set (a,b,c,d,e,f)
  - Can we do better?
    - Yes, we can use 3-bit encoding for characters
  - How many bits will be required?
    - 100000 * 3 bits

Huffman codes

- Suppose we know the frequency of each character
- Can we do better?

	a	b	c	d	e	f
Freq (thousands)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed-length code	**000**	**001**	**010**	**011**	**100**	**101**

Huffman codes

- Suppose we know the frequency of each character
- Can we do better?

encode: face

11 0 00 1

ee a aa e

	a	b	c	d	e	f
Freq (thousands)	45	13	12	16	9	5
Fixed-length code	000	001	010	011	100	101
variable length	0	01	00	010	1	11

Huffman codes

- Suppose we know the frequency of each character
- Can we do better?

```
face will be encoded as 11 0 00 1
Can be decoded as ee a aa e
```

	a	b	c	d	e	f
Freq (thousands)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed-length code	**000**	**001**	**010**	**011**	**100**	**101**
variable length	**0**	**01**	**00**	**010**	**1**	**11**

If we use a variable-length encoding in which the encoding of one character can be the prefix of another character, it will be difficult to decode the message.

Huffman codes

- Suppose we know the frequency of each character
- Can we do better?

`encode: face`

	a	b	c	d	e	f
Freq (thousands)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed-length code	**000**	**001**	**010**	**011**	**100**	**101**
variable length	**0**	**101**	**100**	**111**	**1101**	**1100**

If we use a variable-length encoding in which the encoding of a character is never a prefix of the encoding of another character, then there will be no ambiguity.

$45 \times 3 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 3$
$+ 5 \times 3$
$= 300$ K

224 K

Huffman codes

- Suppose we know the frequency of each character
- Can we do better?

```
face will be encoded as 1100 0 100 1101
only possible decoding is f a c e
```

	a	b	c	d	e	f
Freq (thousands)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed-length code	**000**	**001**	**010**	**011**	**100**	**101**
variable length	**0**	**101**	**100**	**111**	**1101**	**1100**

If we use a variable-length encoding in which the encoding of a character is never a prefix of the encoding of another character, then there will be no ambiguity.

Prefix-free codes

- No codeword is a prefix of some other codeword

	a	b	c	d	e	f
Freq (thousands)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed-length code	**000**	**001**	**010**	**011**	**100**	**101**
variable length	**0**	**101**	**100**	**111**	**1101**	**1100**

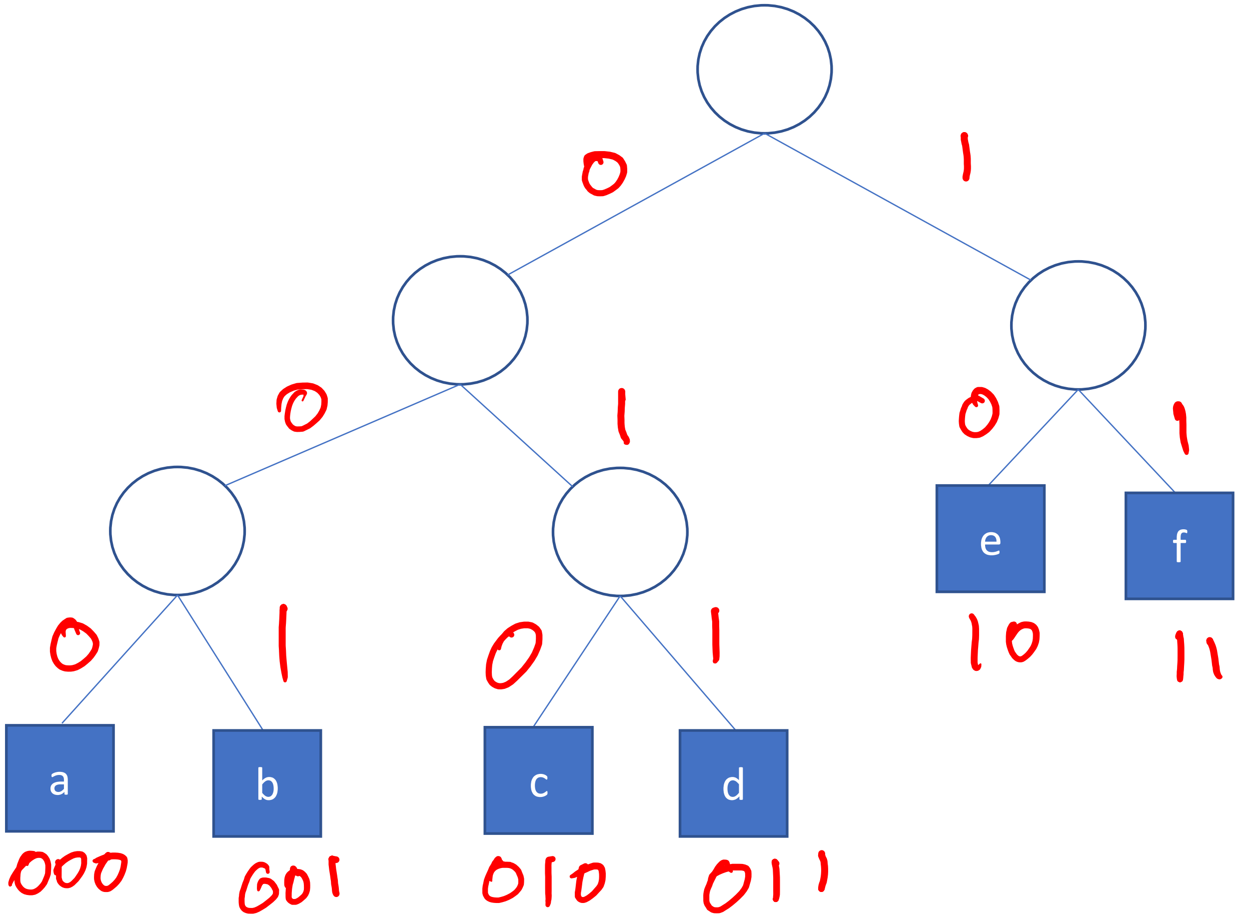
Prefix-free codes

- Using prefix-free codes, we can uniquely identify the next character during decoding

- No ambiguity during decoding

Prefix-free codes

- A binary tree can be used to generate prefix-free codes

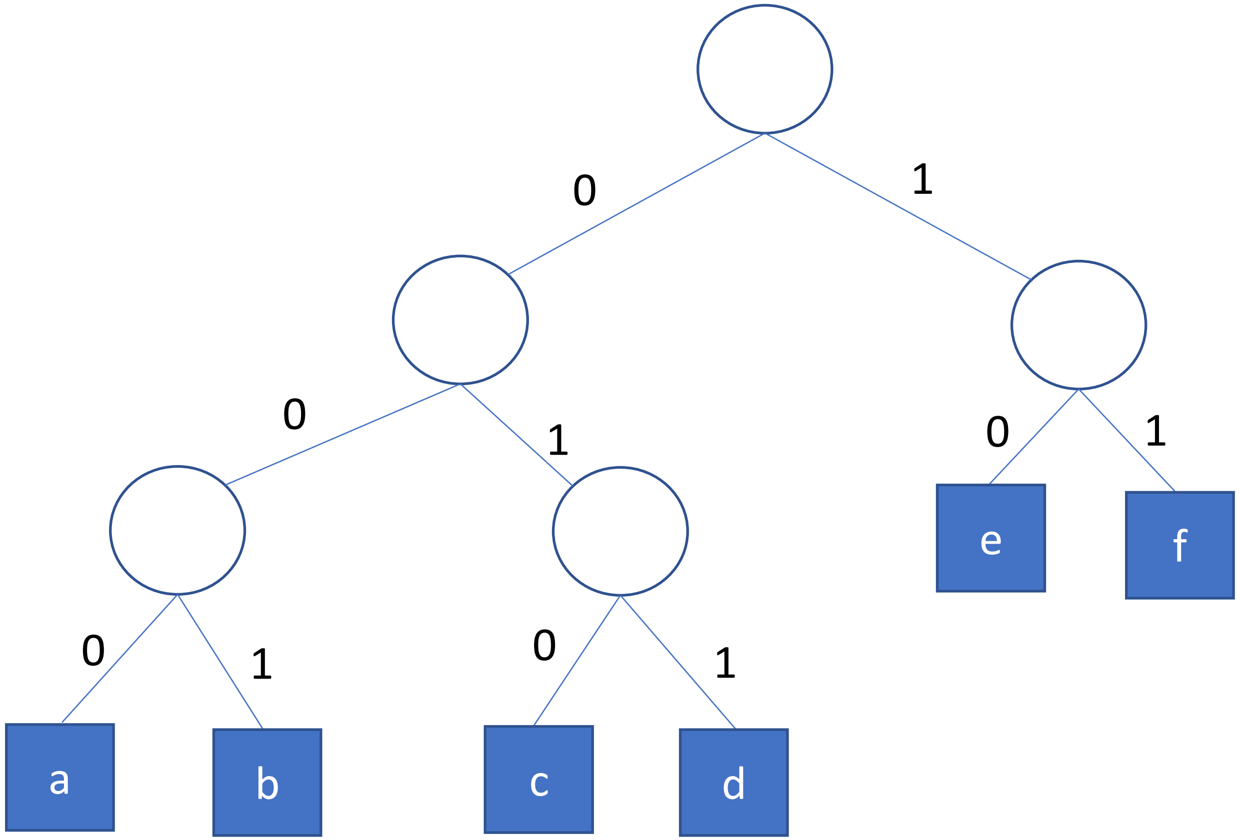


The square nodes or leaves represent the characters in the file that we want to compress. There is a unique path to each square node from the root.

We can assign prefix-free codes to these paths.

Prefix-free codes

- A binary tree can be used to generate prefix-free codes



Each edge on the path is a bit in the corresponding encoding. If we assign different bits to the left and right children, then none of the nodes in the left subtree will be the prefix of a node in the right subtree and vice-versa. If we apply this rule for all nodes, we will get prefix-free codes for the leaf nodes.

Prefix-free codes

300,000 − 14,000

- No codeword is a prefix of some other codeword
  - encoding-1 number of bits
  - encoding-2 number of bits

	a	b	c	d	e	f
Freq (thousands)	45	13	12	16	9	5
Fixed-length code	000	001	010	011	100	101
variable length-1	0	101	100	111	1101	1100
variable length-2	000	001	010	011	10	11

Prefix-free codes

$$\begin{array}{r} 300 \\ -\ 58 \\ \hline 242 \end{array}$$

- No codeword is a prefix of some other codeword
  - encoding-1 number of bits
    - (45 * 1) + (13 * 3) + (12 * 3) + (16 * 3) + (9 * 4) + (5 * 4) = 224 thousands
  - encoding-2 number of bits
    - (45 * 3) + (13 * 3) + (12 * 3) + (16 * 3) + (9 * 2) + (5 * 2) = 286 thousands
  - Even though the encoding-1 scheme uses up to four bits, it is better than the encoding-2 scheme, which uses two and three bits codes

	a	b	c	d	e	f
Freq (thousands)	45	13	12	16	9	5
Fixed-length code	000	001	010	011	100	101
encoding-1	0	101	100	111	1101	1100
encoding-2	000	001	010	011	10	11

Prefix-free codes

- For better compression, we need to generate smaller length codes for those characters whose frequencies are high

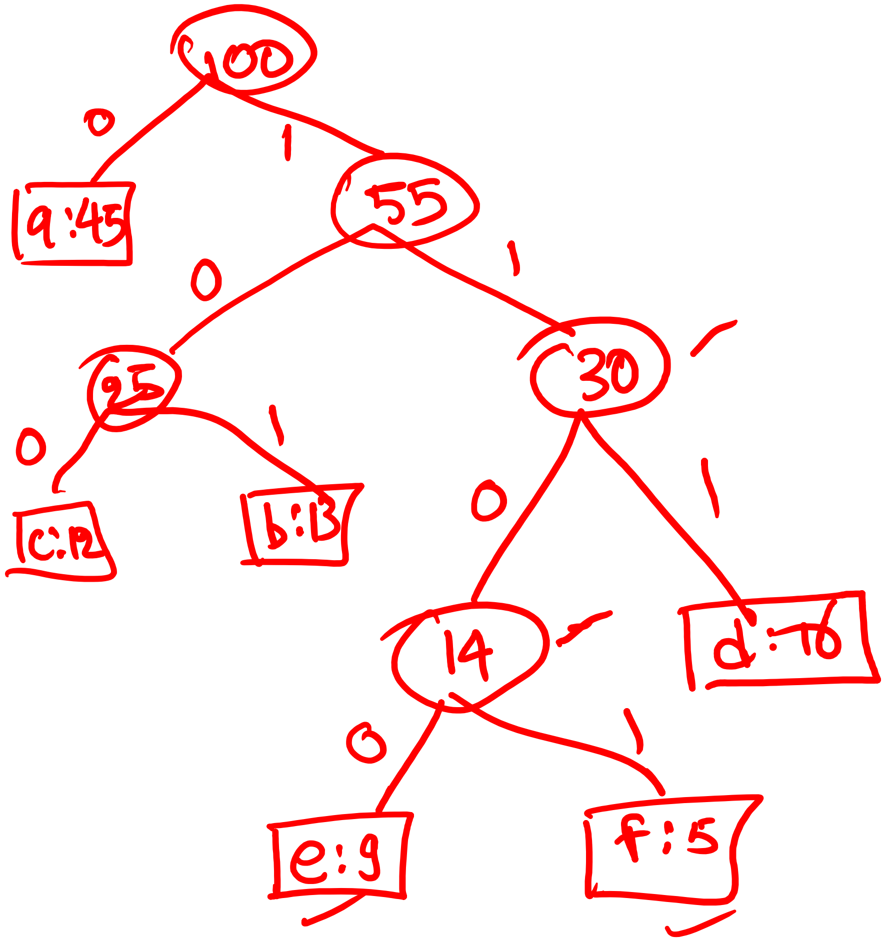
- For low-frequency characters, we can use larger codes

Huffman code

- Huffman algorithm can be used to generate an optimal prefix-free code

Huffman encoding

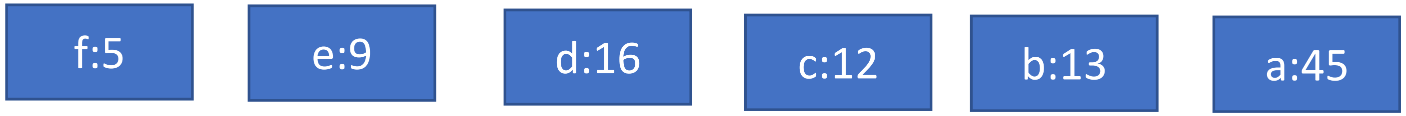
CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1100
f	5	1101



Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	




```
Keep connecting two nodes with
minimum frequencies unless all
nodes are connected.
```

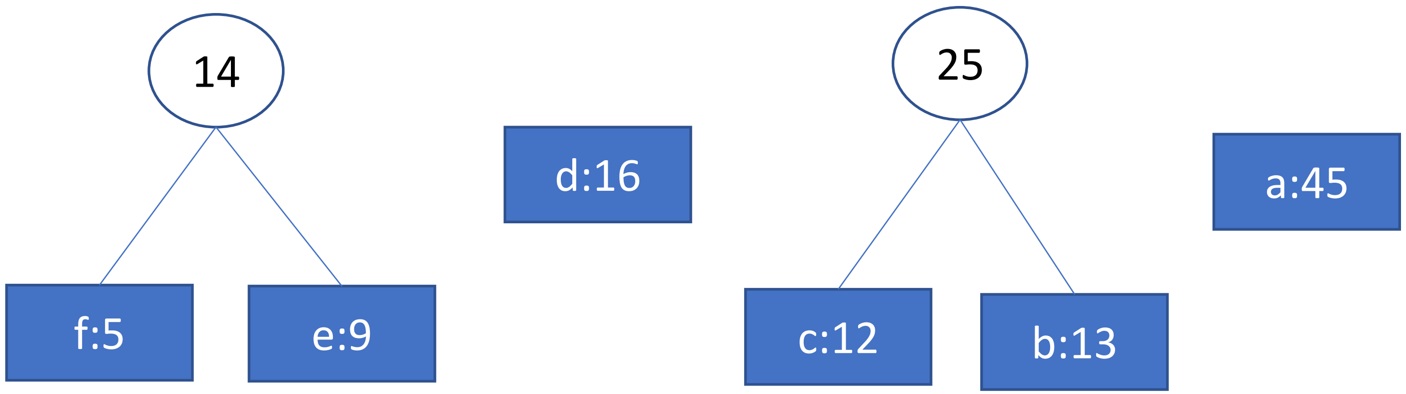
Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	


```
Keep connecting two nodes with
minimum frequencies unless all
nodes are connected.
```

Huffman encoding

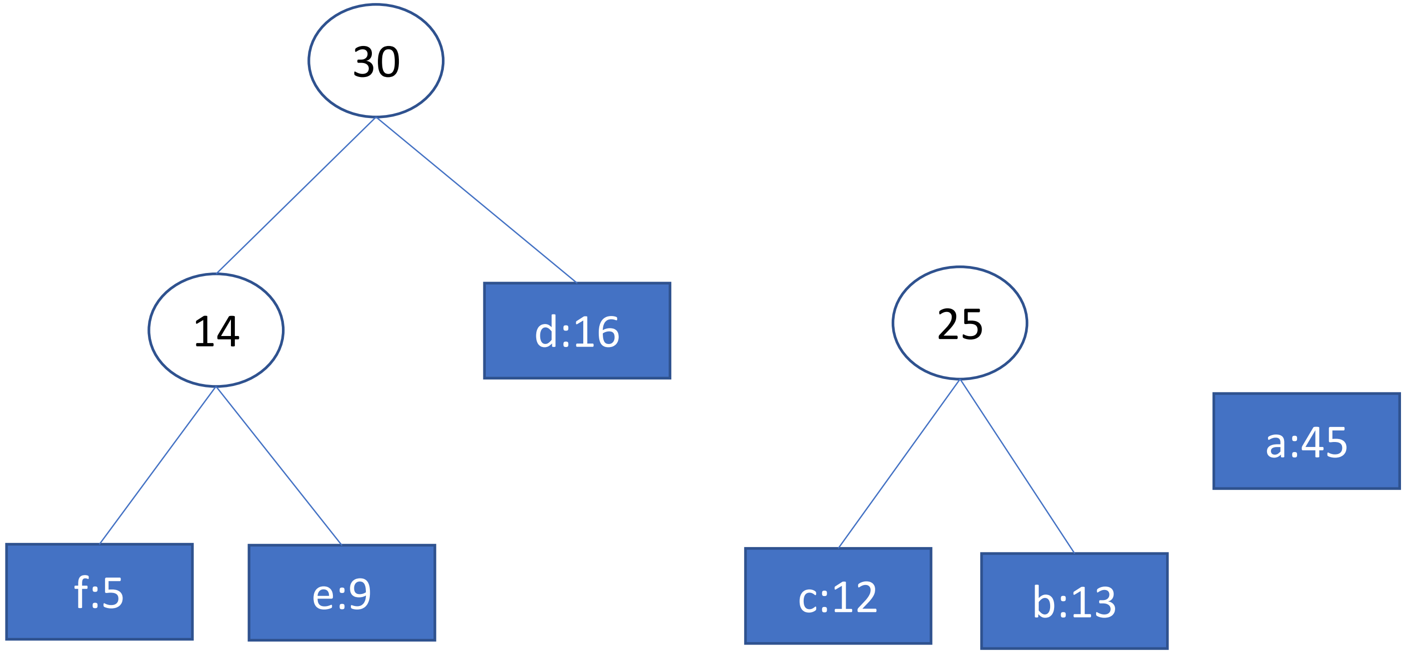
CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	



Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Huffman encoding

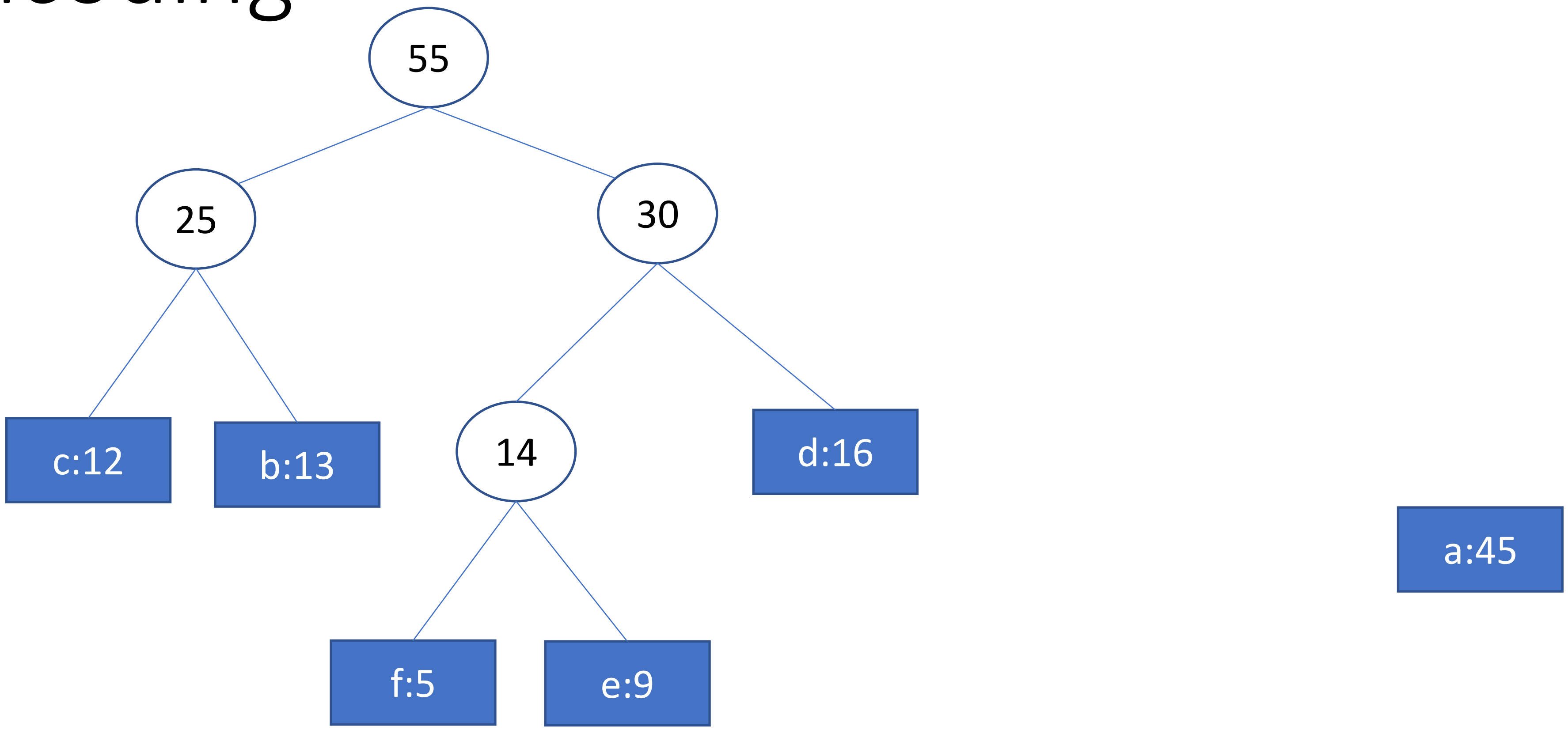
CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	




```
Keep connecting two nodes with
minimum frequencies unless all
nodes are connected.
```

Huffman encoding

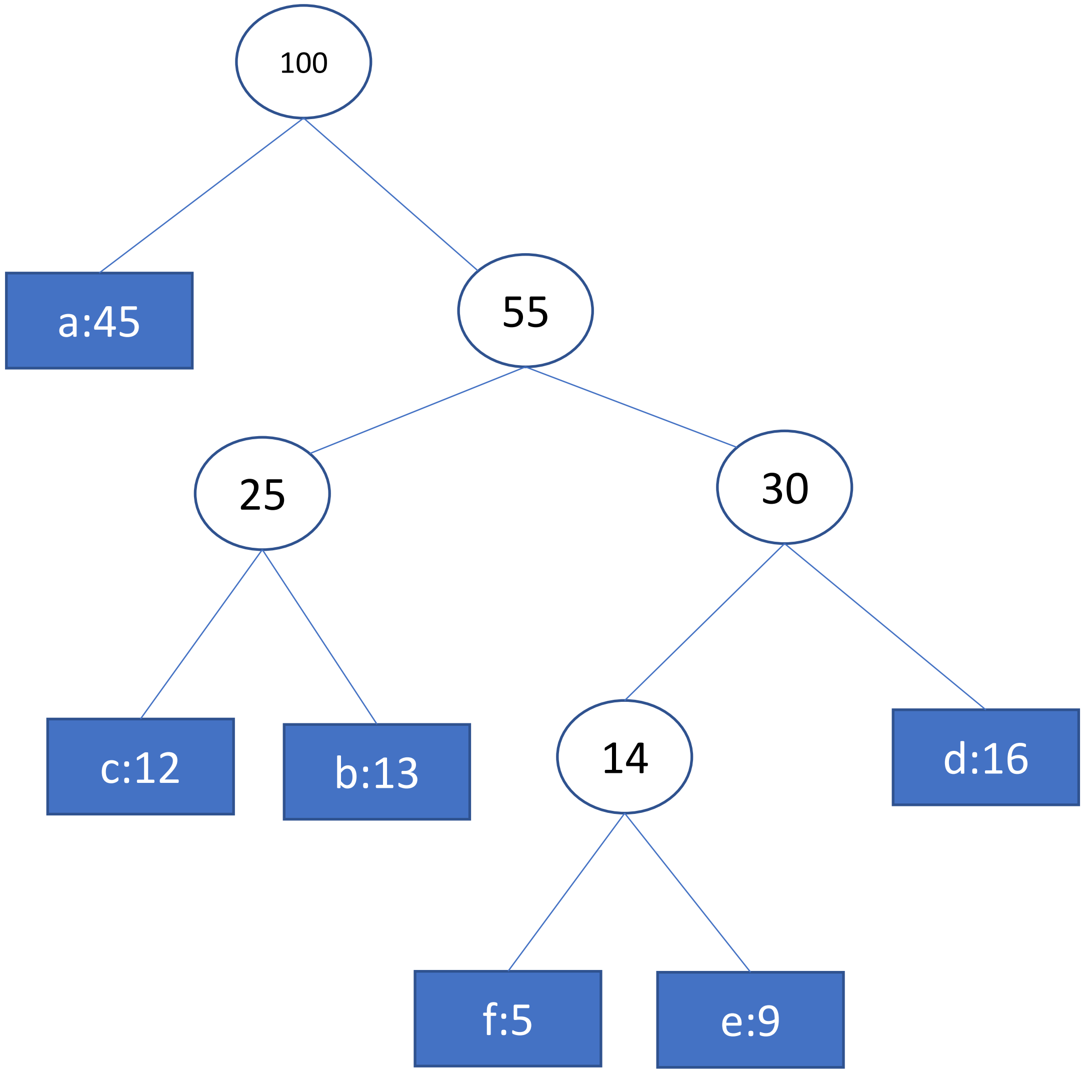
CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	



Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Huffman encoding

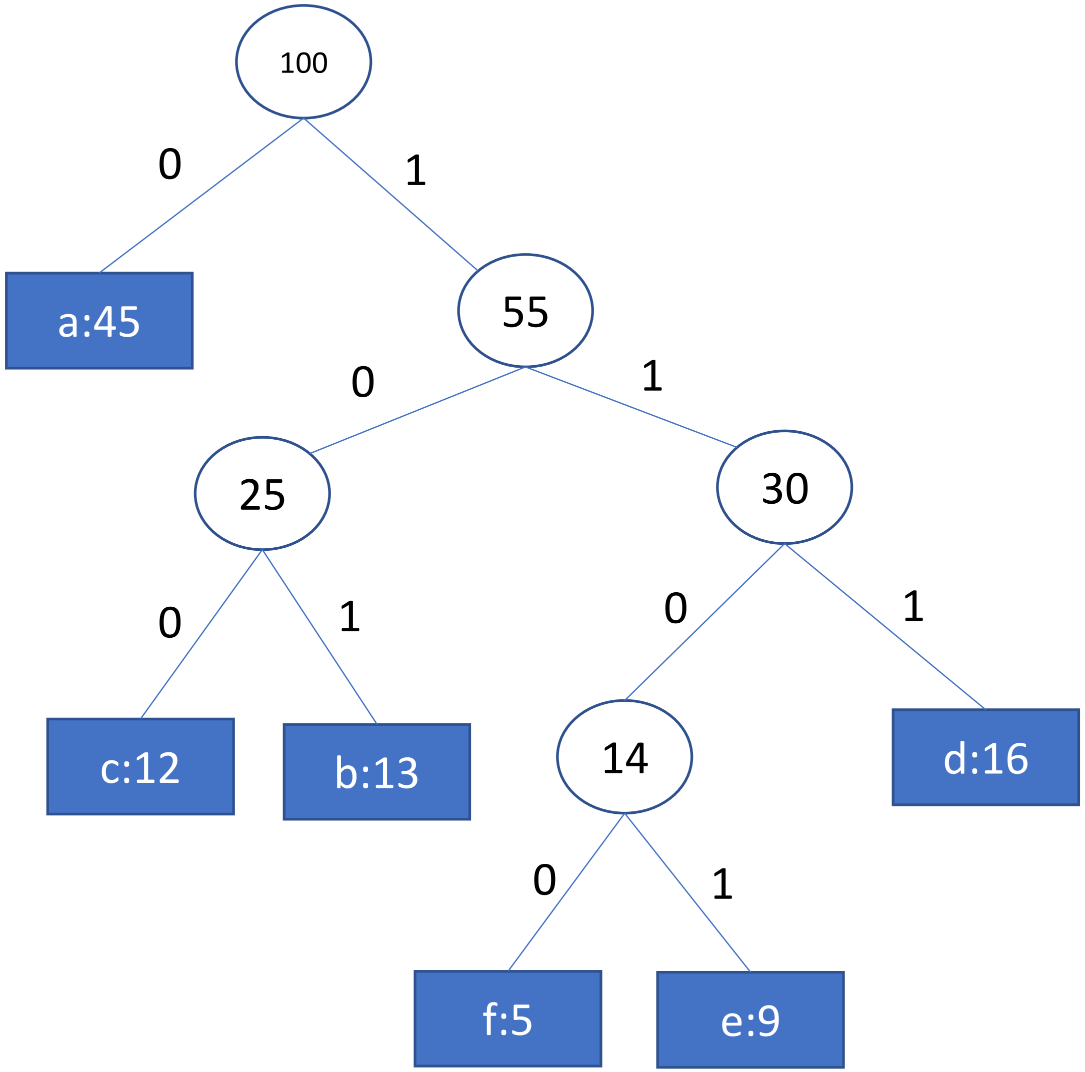
CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	




```
Keep connecting two nodes with
minimum frequencies unless all
nodes are connected.
```

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	



Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Huffman encoding

- Implementation

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	

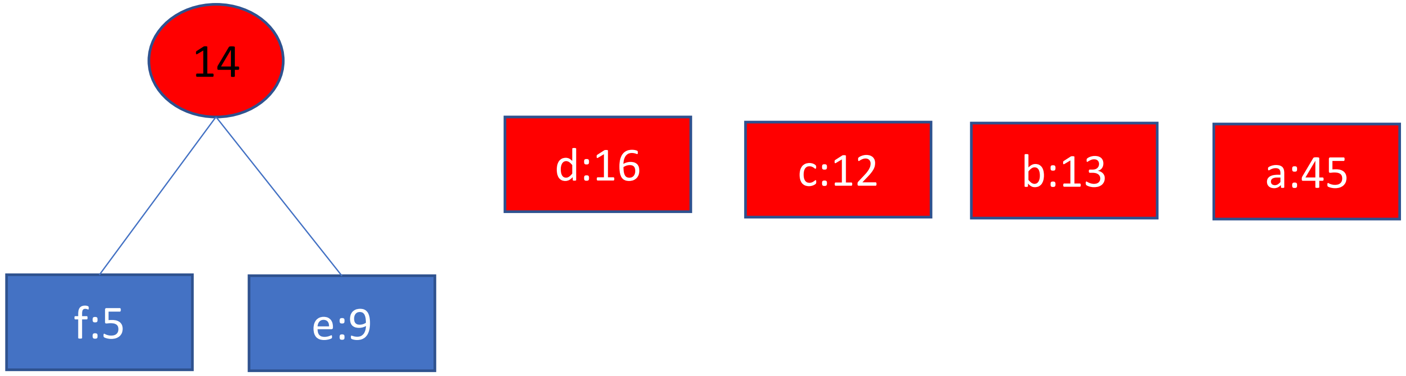
f:5 e:9 d:16 c:12 b:13 a:45

Red nodes are in the min heap.

Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	

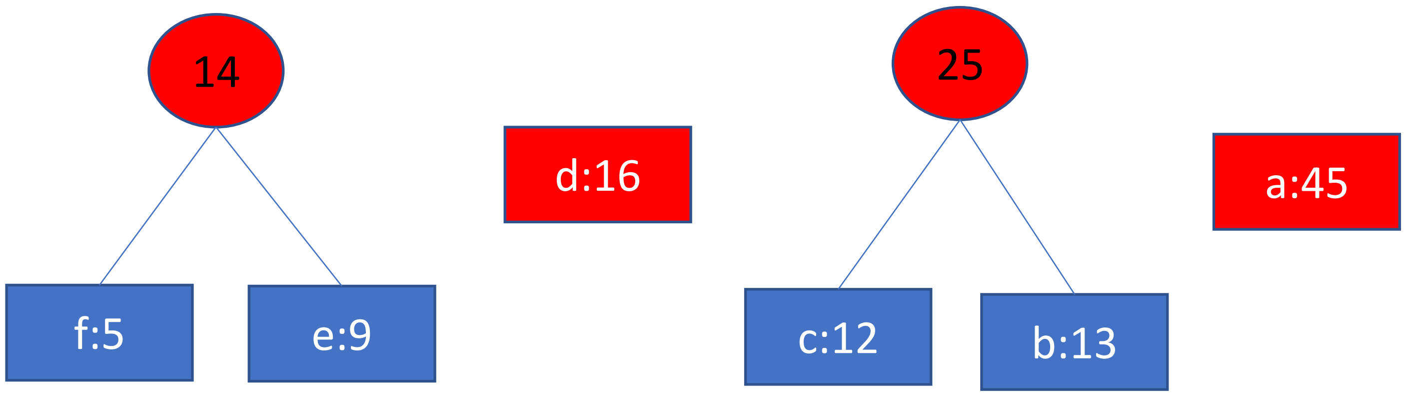


Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Red nodes are in the min heap.
Extract two nodes from the min heap, connect them, and add the new node to the min heap.

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	

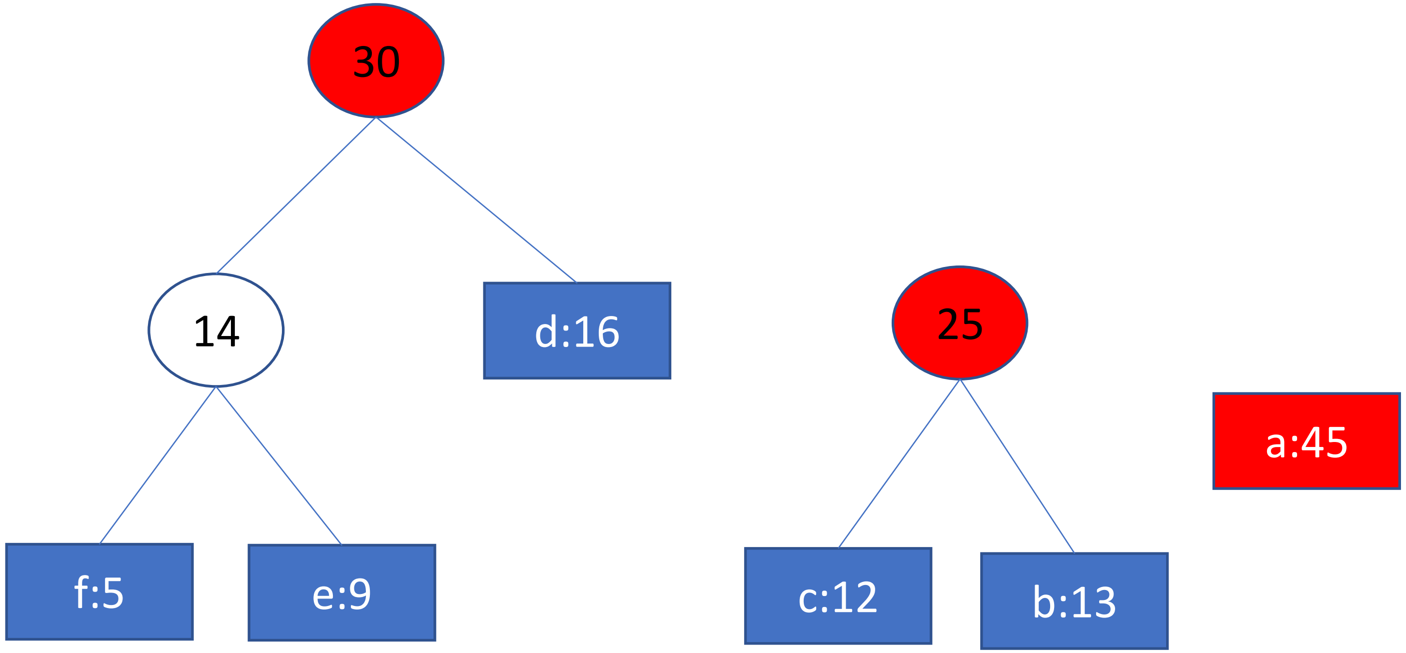


Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Red nodes are in the min heap.
Extract two nodes from the min heap, connect them, and add the new node to the min heap.

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	

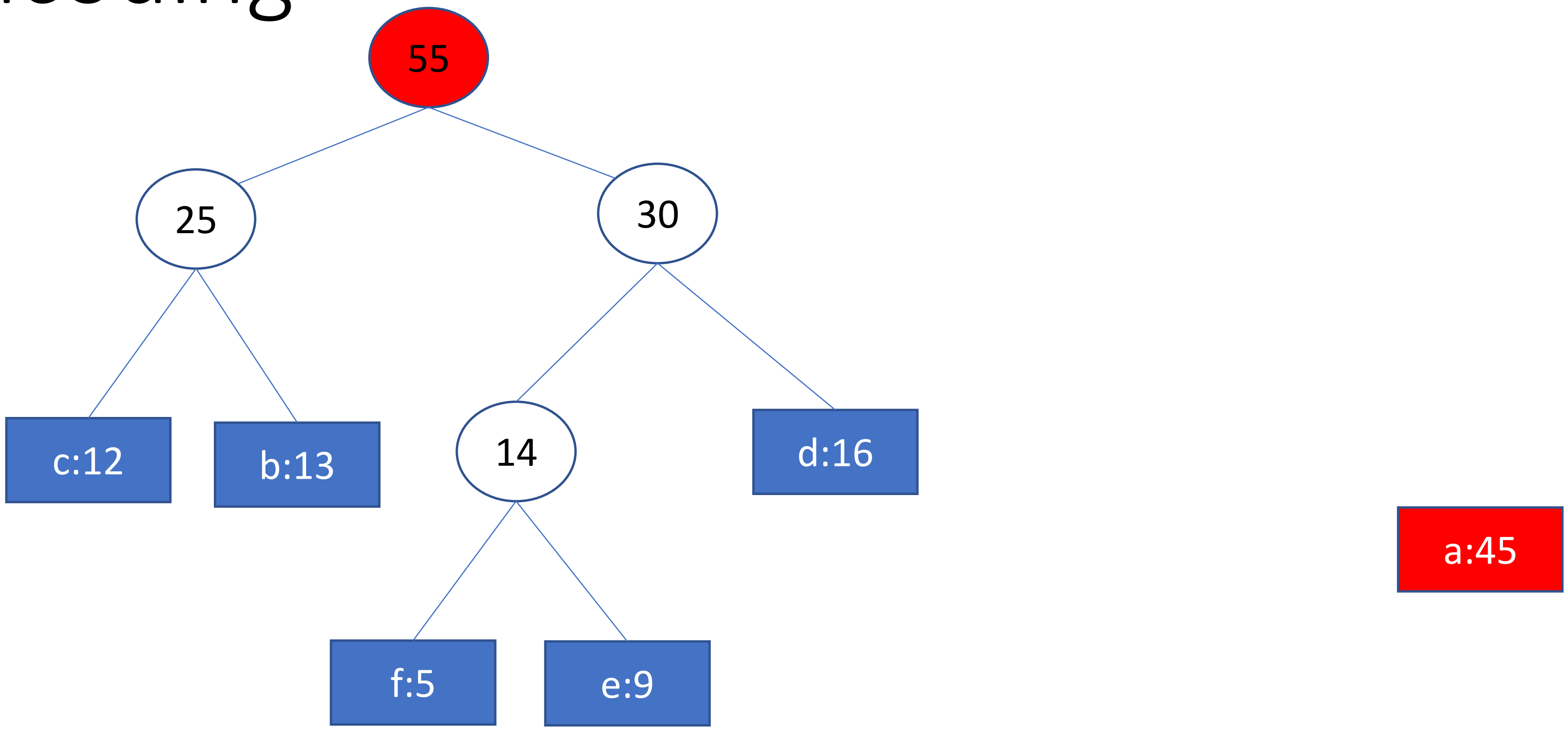


Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Red nodes are in the min heap.
Extract two nodes from the min heap, connect them, and add the new node to the min heap.

Huffman encoding

CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	

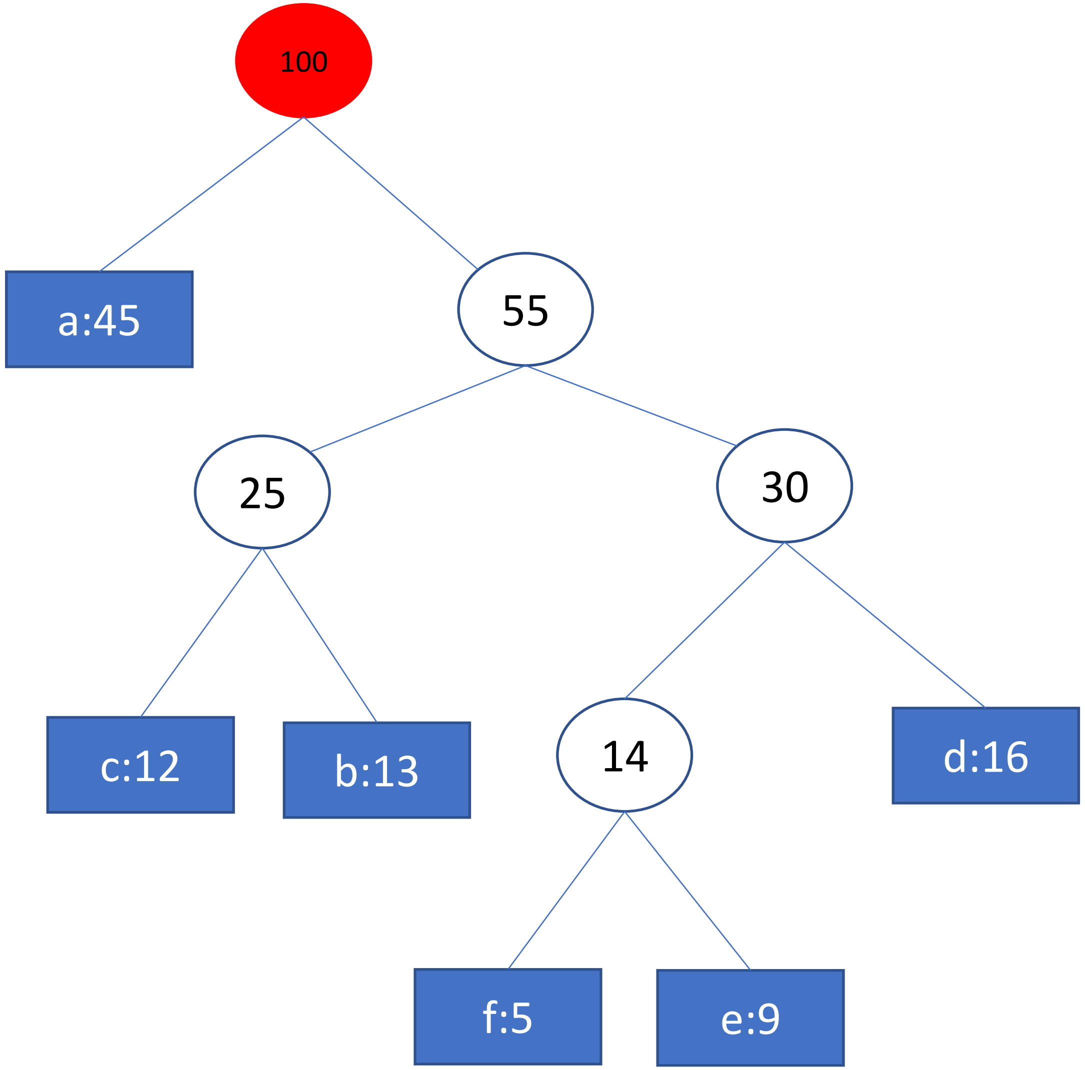


Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Red nodes are in the min heap.
Extract two nodes from the min heap, connect them, and add the new node to the min heap.

Huffman encoding

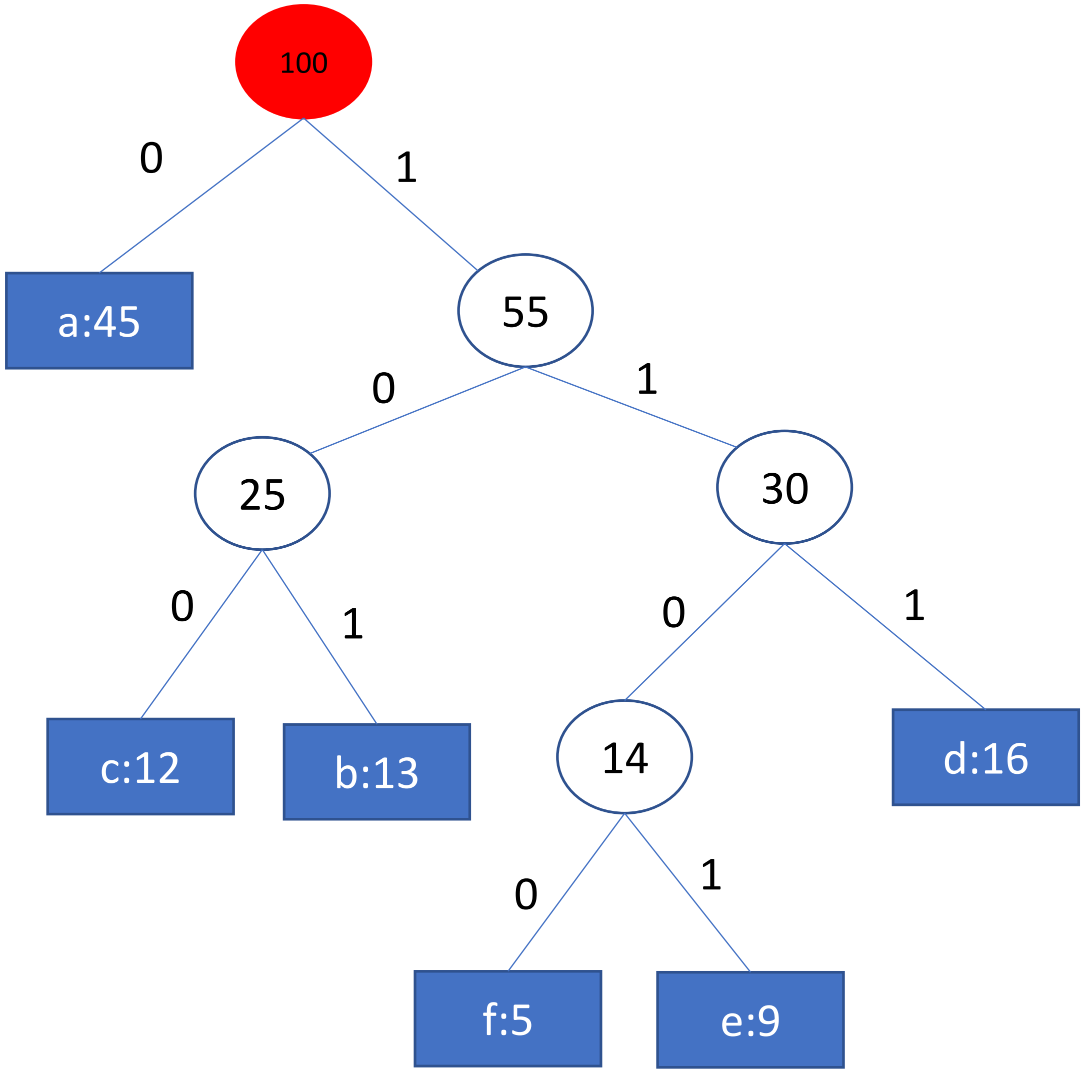
CHAR	FREQ	ENC
a	45	
b	13	
c	12	
d	16	
e	9	
f	5	



Keep connecting two nodes with minimum frequencies unless all nodes are connected.

Huffman encoding

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100




```
Keep connecting two nodes with
minimum frequencies unless all
nodes are connected.
```

Huffman encoding algorithm

```
Algorithm Huffman(C):
Input: C is an array of n nodes; each node contains a character and its
corresponding frequency, and NULL left and right fields
Output: The root of a tree representing the Huffman codes
    n = Num_Elements(C)
    Q = Build_Min_Heap(C)
    for i = 1 to n – 1 do
        z = allocate_node() // allocate a node with empty fields
        z.left = x = Extract_Min(Q)
        z.right = y = Extract_Min(Q)
        z.freq = x.freq + y.freq
        Insert(Q, z)
    return Extract_Min(Q)
```

Time complexity:nlog(n)

n log n

Time complexity

- All the heap operations inside the loop take $O(\log n)$ time
- The loop runs $n-1$ time
- Build_Min_Heap takes $O(n)$ time
- Therefore, the time complexity is $O(n * \log(n))$

Huffman codes

- Initially, the resulting code is empty
- To generate the Huffman code for a given leaf node x, walk from the root node to x, visiting each node in the path exactly once
- For every left turn, append zero to the resulting code
- For every right turn, append one to the resulting code

Huffman decoding

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100

f a c e

1100 0 100 1101

Huffman encoding

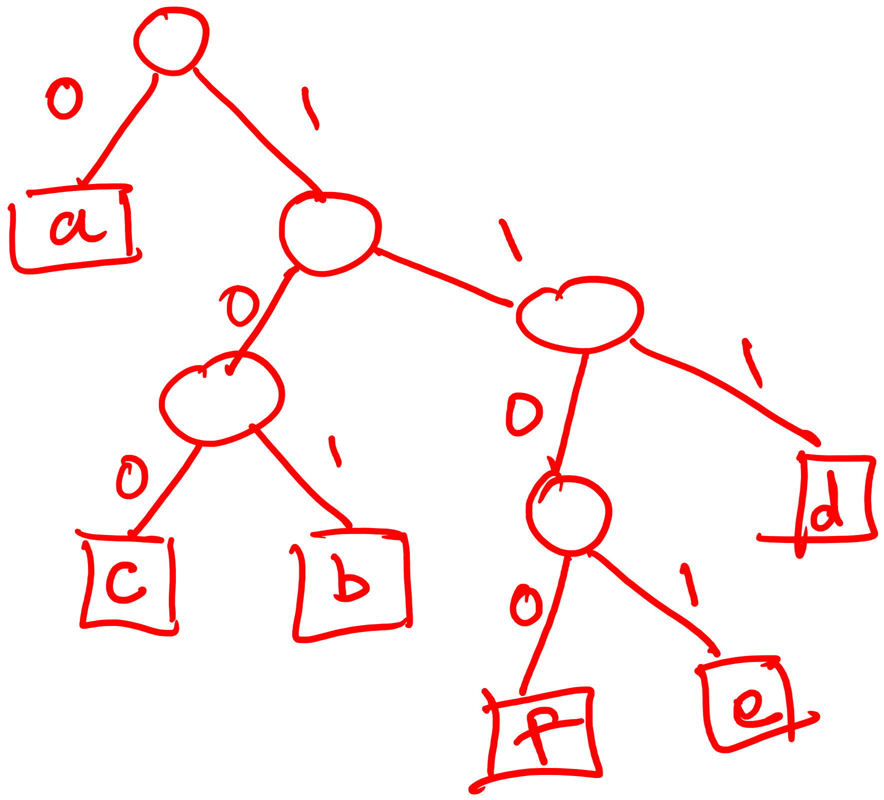
- Store the mapping from an original sequence to the encoded sequence in the compressed file

Huffman decoding

- Retrieve the character encoding from the compressed file
- Build the corresponding tree
- Read the encoded bits in an array
- call HuffmanDecoding until all characters are decoded

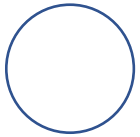
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



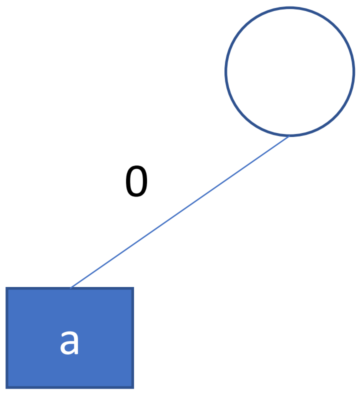
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



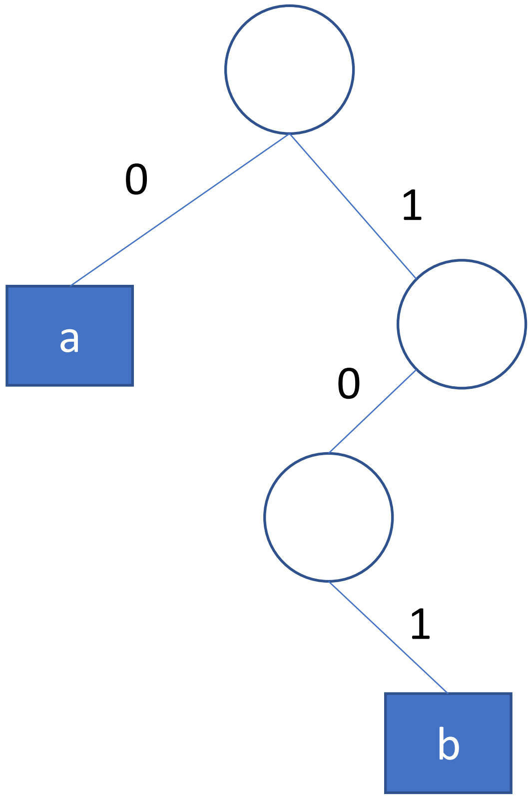
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



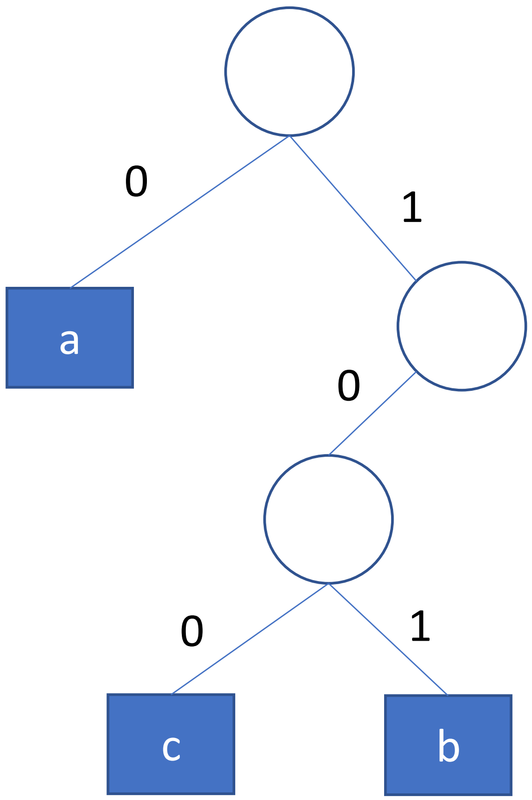
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



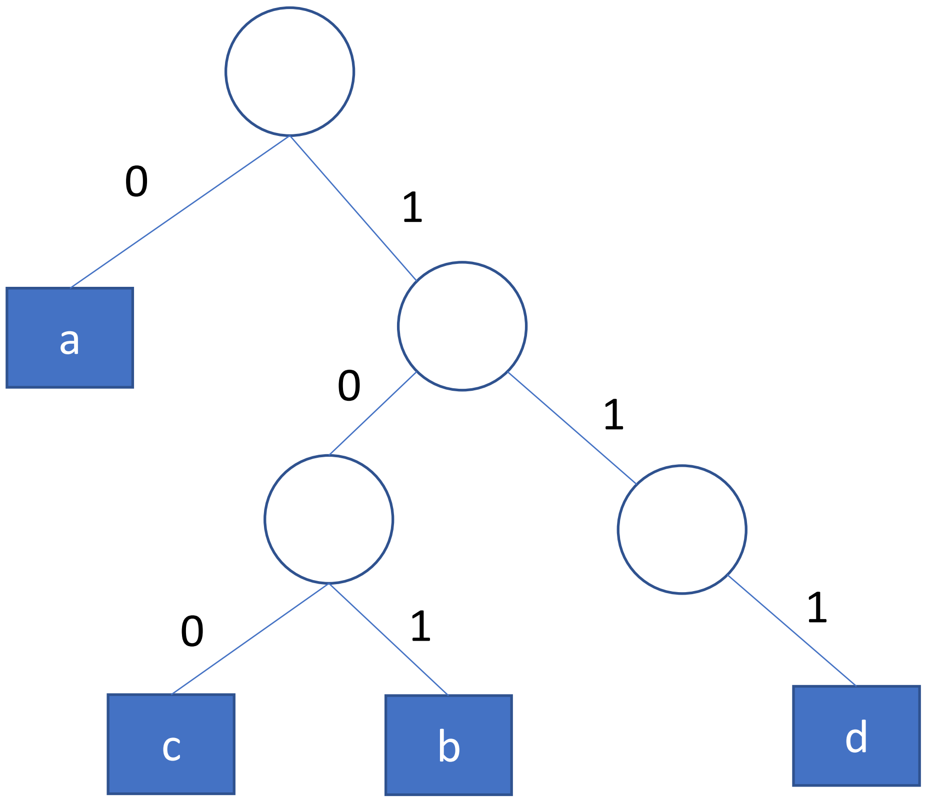
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



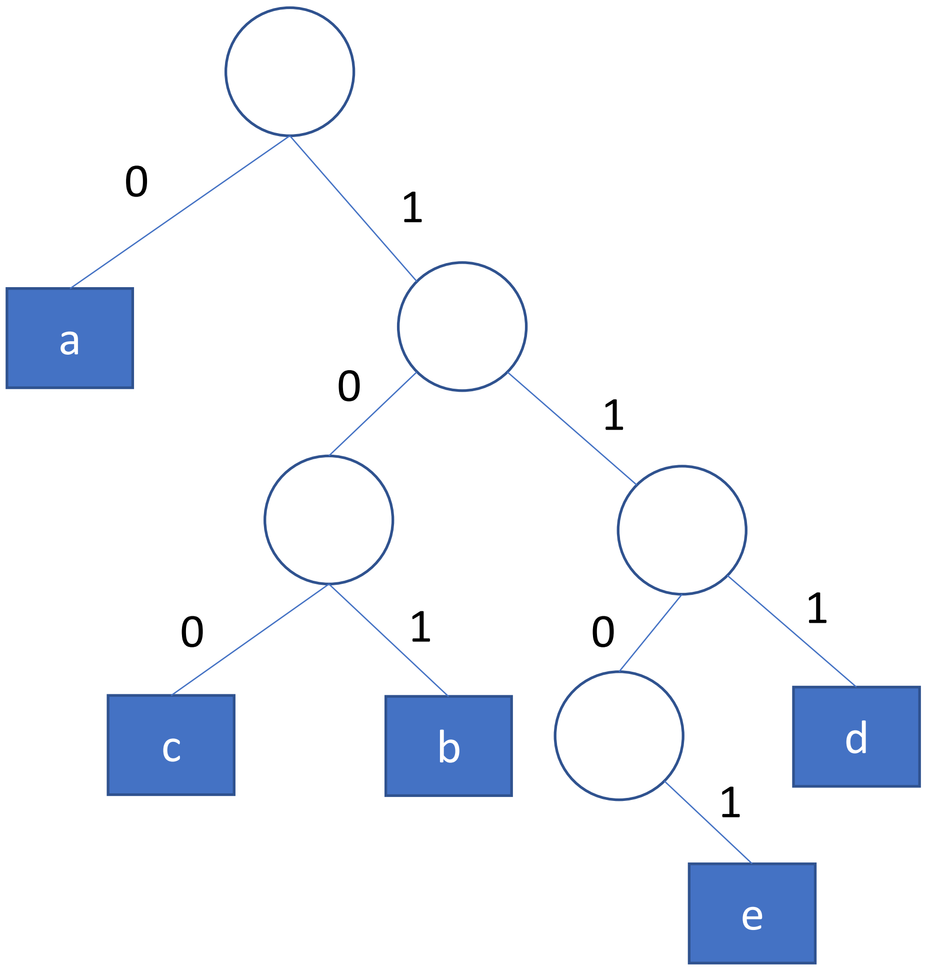
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



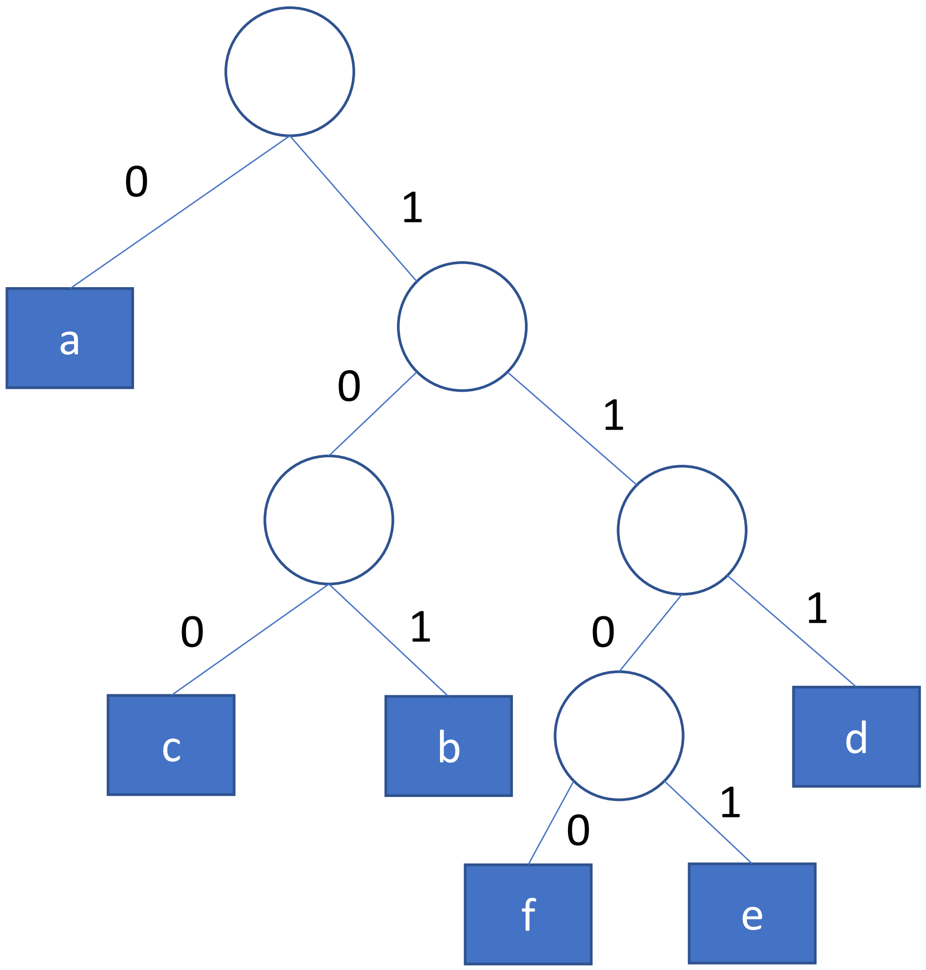
Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



Building tree

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



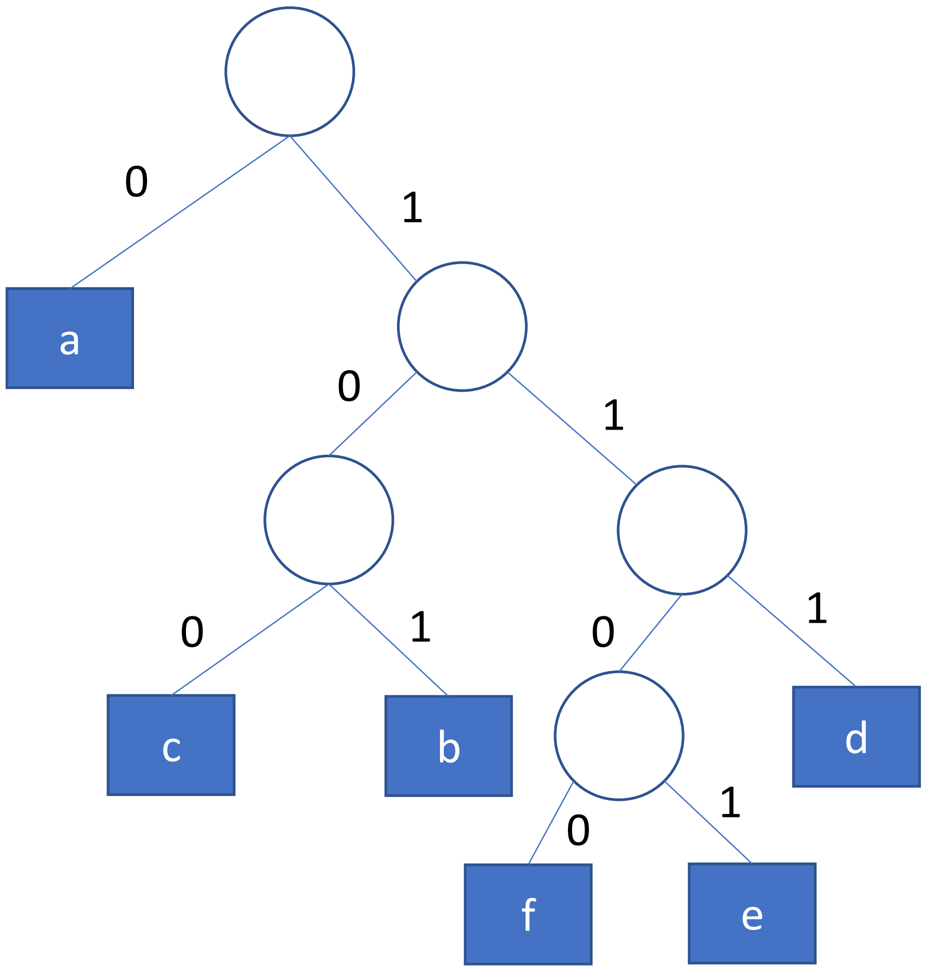
Building tree

- We can store the encoding differently in the compressed file to build the tree faster during the decoding; however, because the total number of characters is usually small, building the tree is not a very time-consuming step

Huffman decoding

CHAR	FREQ	ENC
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100

f a c e



`decode: 110001001101`

Huffman decoding

```
Algorithm HuffmanDecoding(root, in)
Input: root is the root of the tree, in points to next bit in the
encoded input sequence
Output: decodes and prints one character, returns the pointer to
the first bit of the next character in the encoded sequence
  if (is_leaf(root)) then
      print(root->val);
      return in;
  if (in[0] = 0) then
      return HuffmanDecoding(root->left, in+1)
  else
      return HuffmanDecoding(root->right, in+1)
```

Decision tree

References

- Chapter-7.9 from Mark Allen Weiss
- Chapter-4.4 from Goodrich and Tamassia

Decision tree

- The binary search take O(log n) time to search an element from the sorted array
- If the hash table is sufficiently large, the search can be done in O(1)
- What is the key difference between binary search and the hash table search?

Decision tree

- The binary search doesn’t care about the type of the key
  - It needs a comparison function that acts as a black box
  - The return value of the comparison function could be <, >, <=, >=, ==, !=

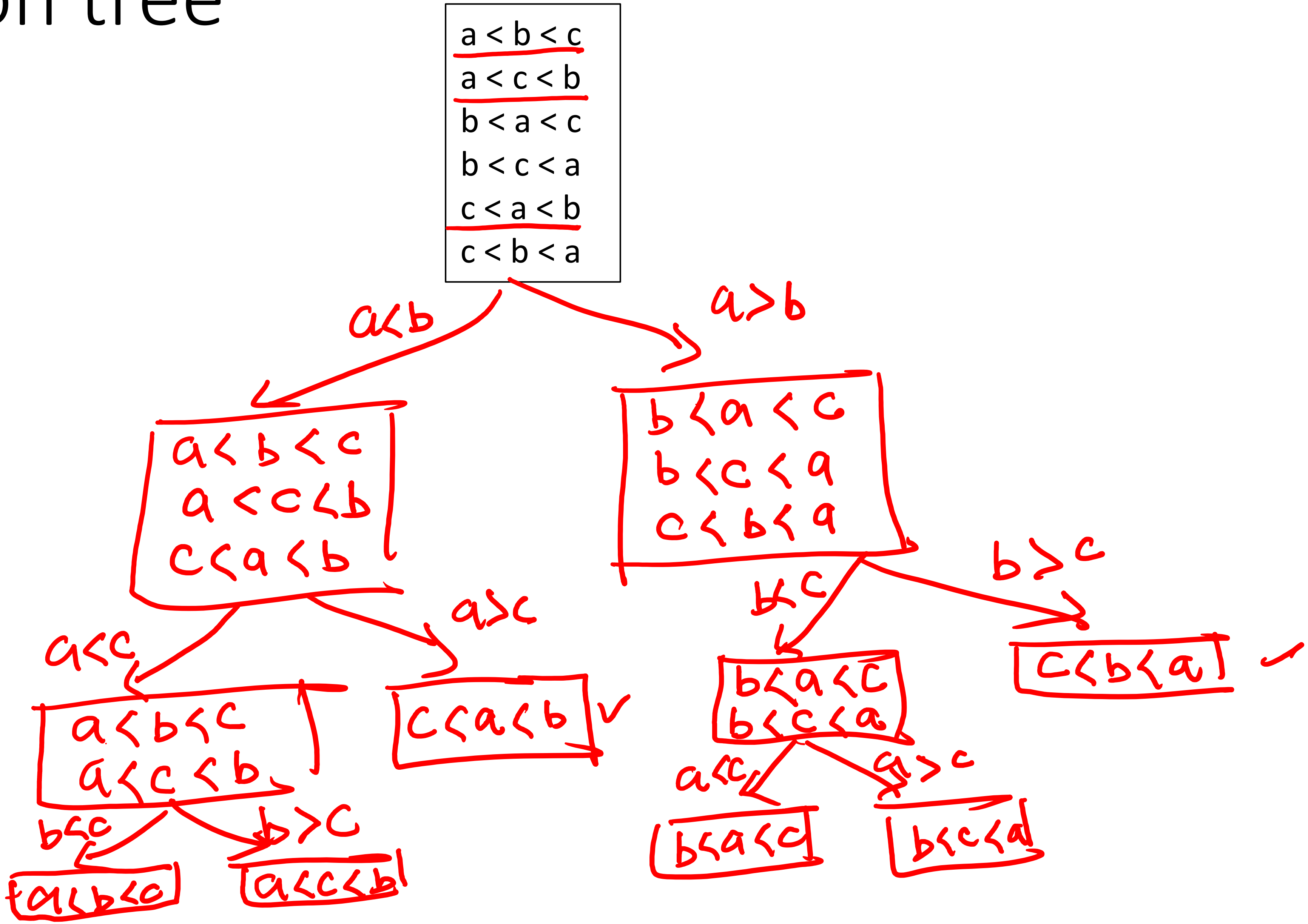
- The hash table will not work if the key can’t be mapped to an integer

Decision tree

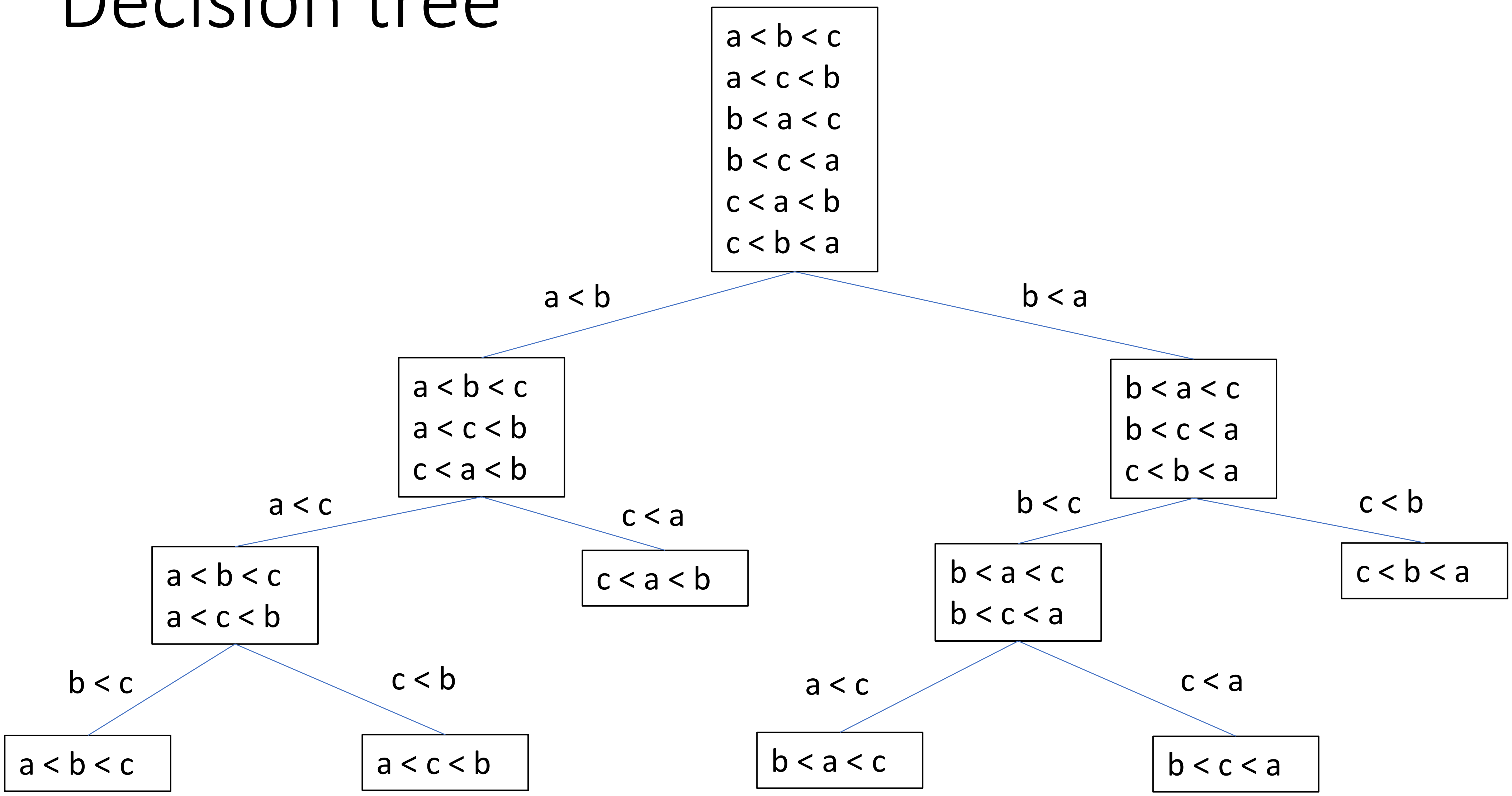
- The decision tree is an abstraction to find a lower bound
- We'll use a binary decision tree to find the lower bound of sorting algorithms
- Each internal node in the decision tree is a comparison operation that may take place during the execution of the algorithm
- The leaf nodes are the output of the algorithm
- The goal is to obtain a lower bound on the number of comparisons to generate the output

Decision tree

2^h



Decision tree



Decision tree

- The previous slide shows a decision tree for sorting three distinct elements a, b, c using just the comparison operations
- The leaf nodes correspond to the possible outputs of the program
- Notice that for three elements, there could be six different possible outcomes; the program may take different paths during runtime to generate an outcome

Decision tree

- Notice that, in general, a program may take different paths to derive the same output

- Therefore, the number of leaves is at least the number of all possible outcomes

- The internal nodes of the decision tree are comparison operations

- After every comparison operation, the program may take two different paths, and the only operations we care about are the comparison operations; therefore, the decision tree is a binary tree

Decision tree

- The lower bound on the number of comparisons is the height of the decision tree

- The height is minimum when the decision tree is a nearly complete or complete binary tree, and the leaves contain only the possible outputs

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?
  - n!

- What is the height, h, of the decision tree when the number of leaves is n!

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?
  - n!
- What is the height, h, of the decision tree when the number of leaves is n!
  - $2^h >= n!$
  - $h >= \log(n!)$

  The minimum number of comparisons = $\log(n!)$

Decision tree

- What is the number of leaf nodes in the decision tree for sorting when we want to sort n numbers?
  - n!

- What is the height, h, of the decision tree when the number of leaves is n!
  - 2^h >= n!
  - h >= log(n!)

The minimum number of comparisons = log(n!)

$$\log(n!) \geq \log\left(\frac{n}{2}^{\frac{n}{2}}\right) \geq \frac{n}{2} * \log\left(\frac{n}{2}\right)$$

$$\geq \frac{n}{2} * (\log(n) - 1) = \Omega(n * \log(n))$$

Decision tree

- Decision tree for searching

Decision tree

- Decision tree for searching
  - How many different outcomes are possible to search an element from an array of n elements?
  - What is the number of leaves?

- What would be the height of the decision tree?

- What is the lower bound on the number of comparisons?

Sorting

References

- Read chapter-4.5 from the Goodrich and Tamassia book

Sorting

- The comparison based sorting algorithms take at least O(n*log(n)) time

- The order sorting algorithms that are not based on the comparisons may execute faster than O(n * log(n))

Sorting

- Consider the following sequence S:

S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))

is a sequence of (k, l) pairs of keys

such that, $0 <= k <= 9$ and $0 <= l <= 9$

An element $(k_1, l_1) < (k_2, l_2)$, if

$k_1 < k_2$ or

$k_1 = k_2$ and $l_1 < l_2$

(1, 2), (1, 5), (1, 7)
(2, 2), (2, 3), (2, 5)
(3, 2), (3, 3)

Sorting

- Consider the following sequence S:

S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))
is a sequence of (k, l) pairs of keys
such that, $0 <= k <= 9$ and $0 <= l <= 9$

An element $(k_1, l_1) < (k_2, l_2)$, if
$k_1 < k_2$ or
$k_1 = k_2$ and $l_1 < l_2$

Sorted sequence:

Sorting

- How to sort $S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))$

1 3 5
7 3 2
1 7 0

(1, 3, 5), (7, 3, 2),
(1, 7, 0)

Sorting

- To sort S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))
- We can first sort the second component

  S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))
- Then we can sort the first component

  $S_{2,1}$ = ((1,2), (1,5), (1,7), (2,2), (2,3), (2,5), (3,2), (3, 3))

- How much time is required to sort the second component?

Sorting

- How much time is required to sort the second component?
  - Store the occurrences of 0, 1, 2, 3, …, 9 in the second component in array count[0:9]
  - Create another sequence S_2
  - copy all elements with zero in the second component at indices 0 to count[0] in S_2
  - copy all elements with one in the second component at indices count[0] to count[1] in S_2
  - copy all elements with two in the second component at indices count[1] to count[2] in S_2
  - …
  - copy all elements with nine in the second component at indices count[8] to count[9] in S_2
  - Total time is O(n)

Sorting

- What is the time complexity of sorting S
    - O(2*n)
- What is the time complexity if the number of keys in an element of the sequence is d
    - O(d*n)
- What is the time complexity of sorting n integers with maximum digits d
    - O(d*n)

Sorting

- How can we sort 32-bit integers?

Sorting

- To sort 32-bit integers, we can first sort the lowest 8-bits, followed by the next 8-bits, followed by the next 8-bits, followed by the most significant 8-bits
  - The size of the count array will be 256 in this case