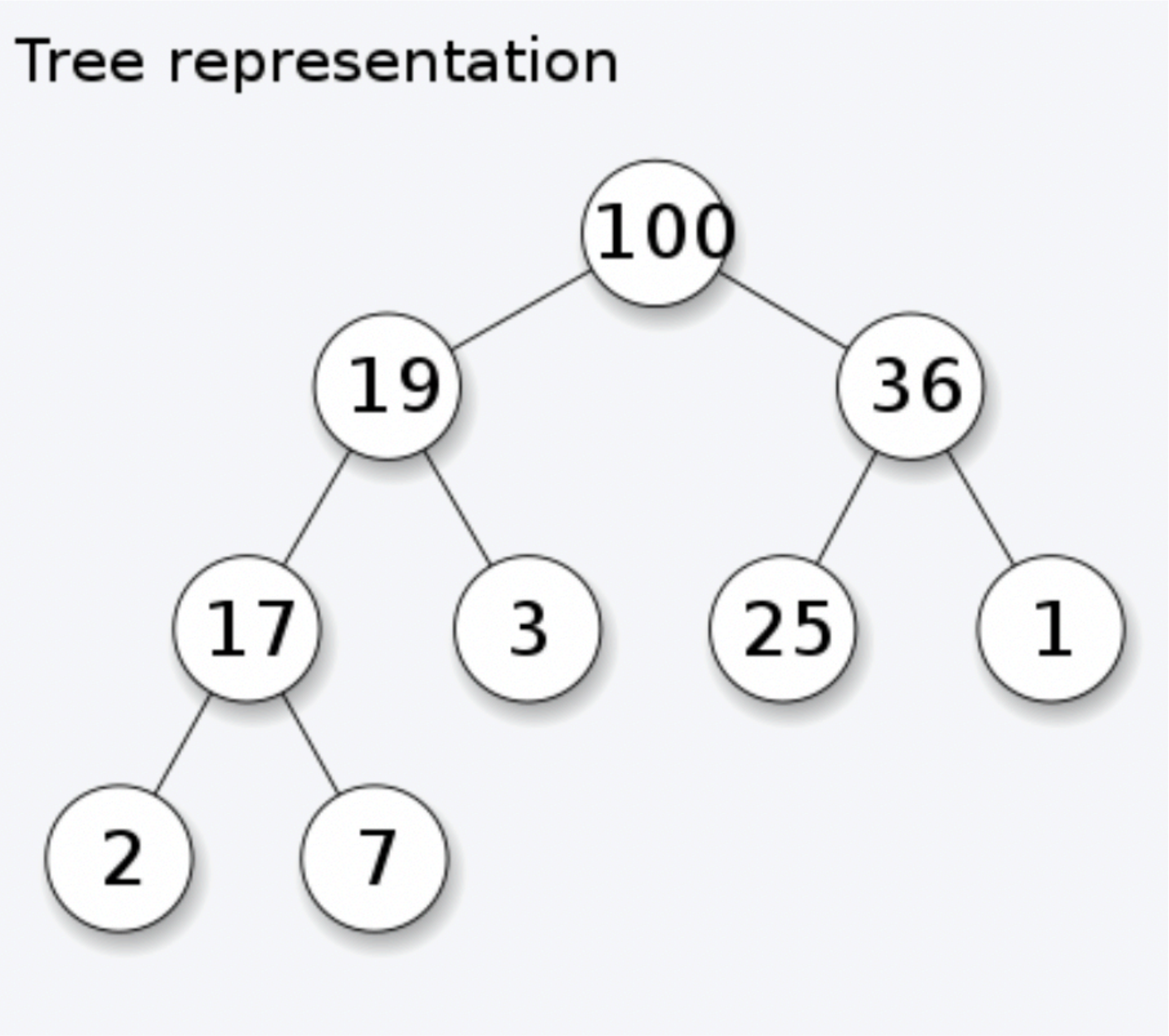
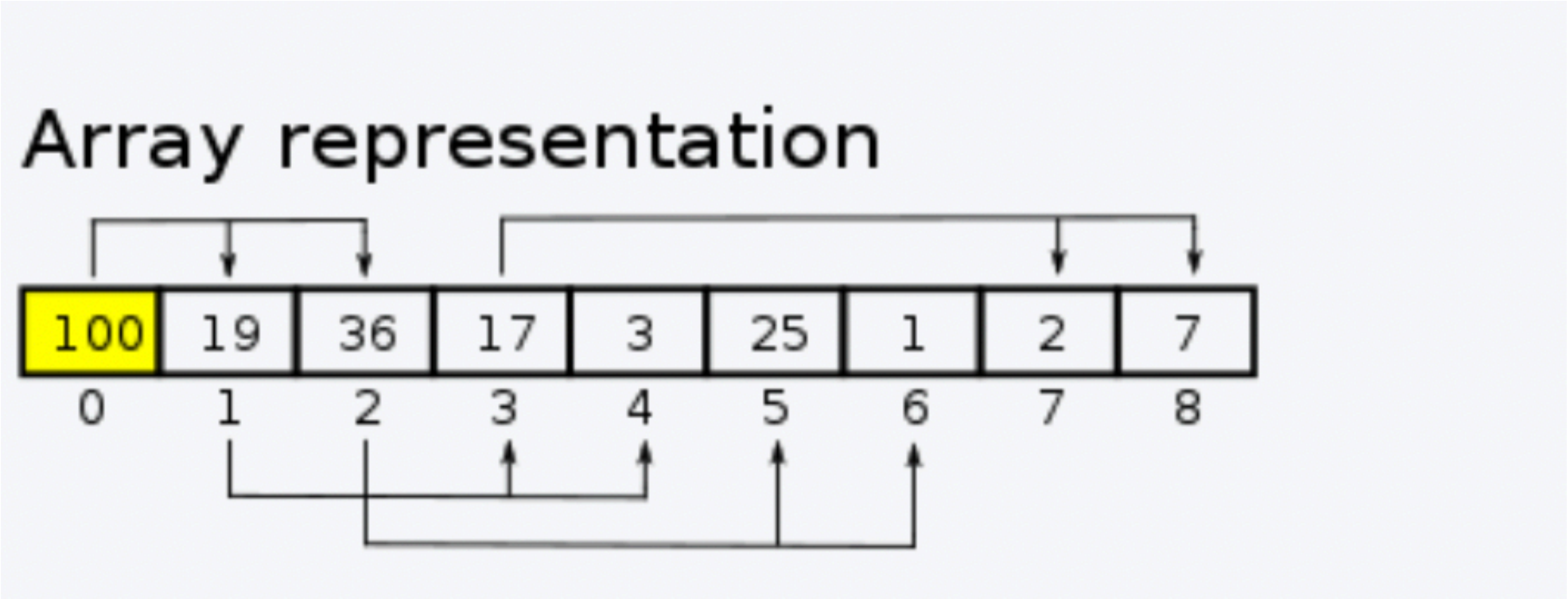
Module 2-3 Heap and its applications





Topic' s Outlines

- ★ Heap
- ★ Heap Sort
- ★ Huffman's Code

Heap ADT

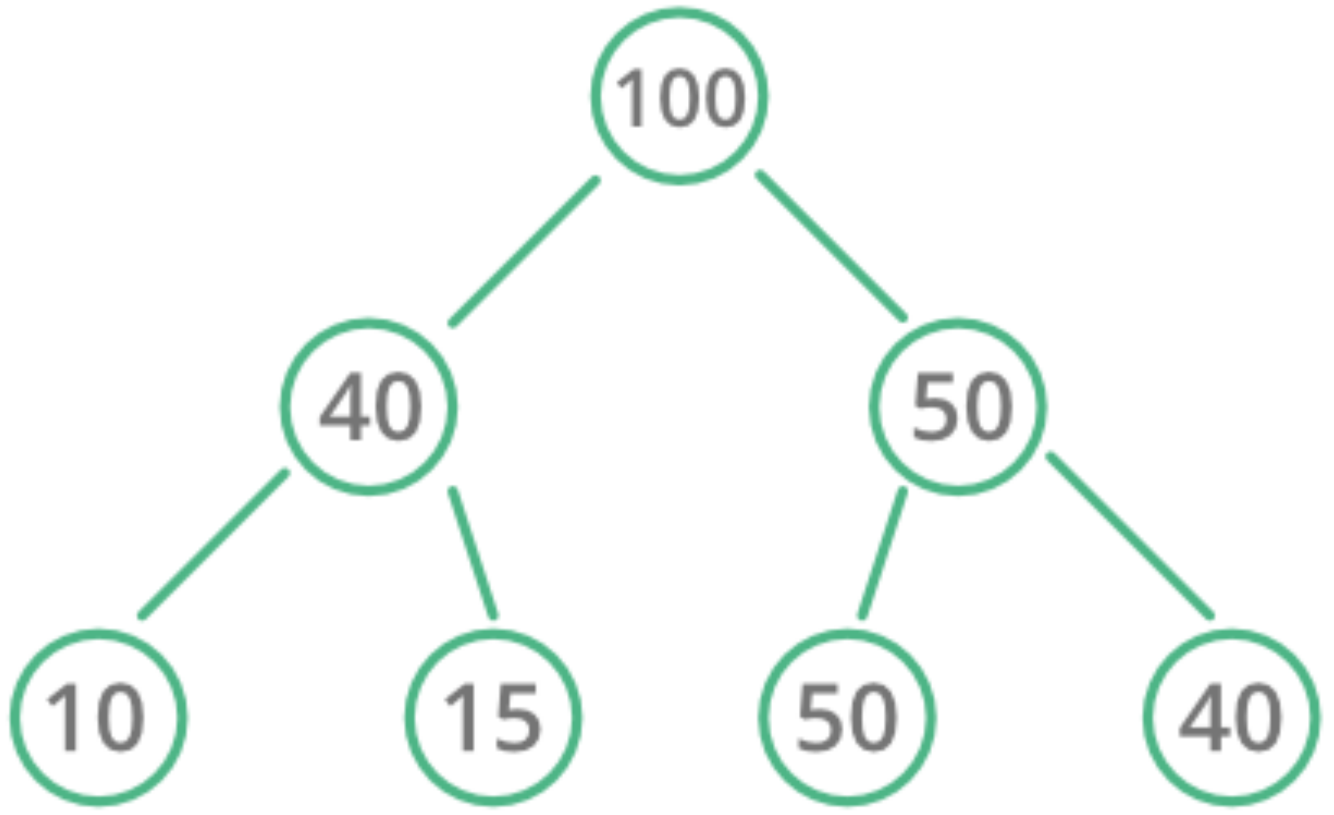
- Heap data structure is a complete binary tree that satisfies the heap property, where any given node is
  - always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
  - always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

Heap ADT

★ Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

★ always greater than its child node/s and the key of the root node is the largest among all other nodes.
This property is also called max heap property.

★ always smaller than the child node/s and the key of the root node is the smallest among all other nodes.
This property is also called min heap property.

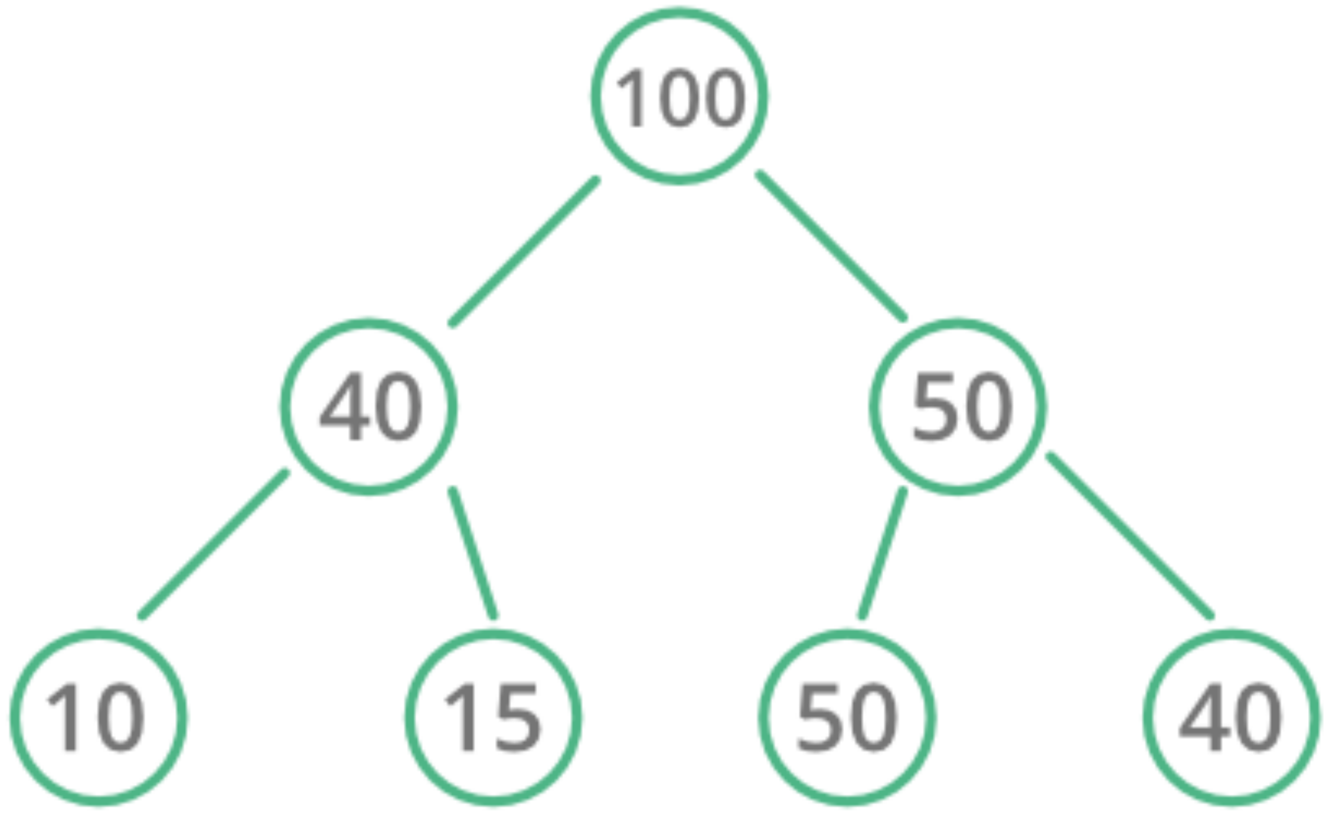


Max Heap

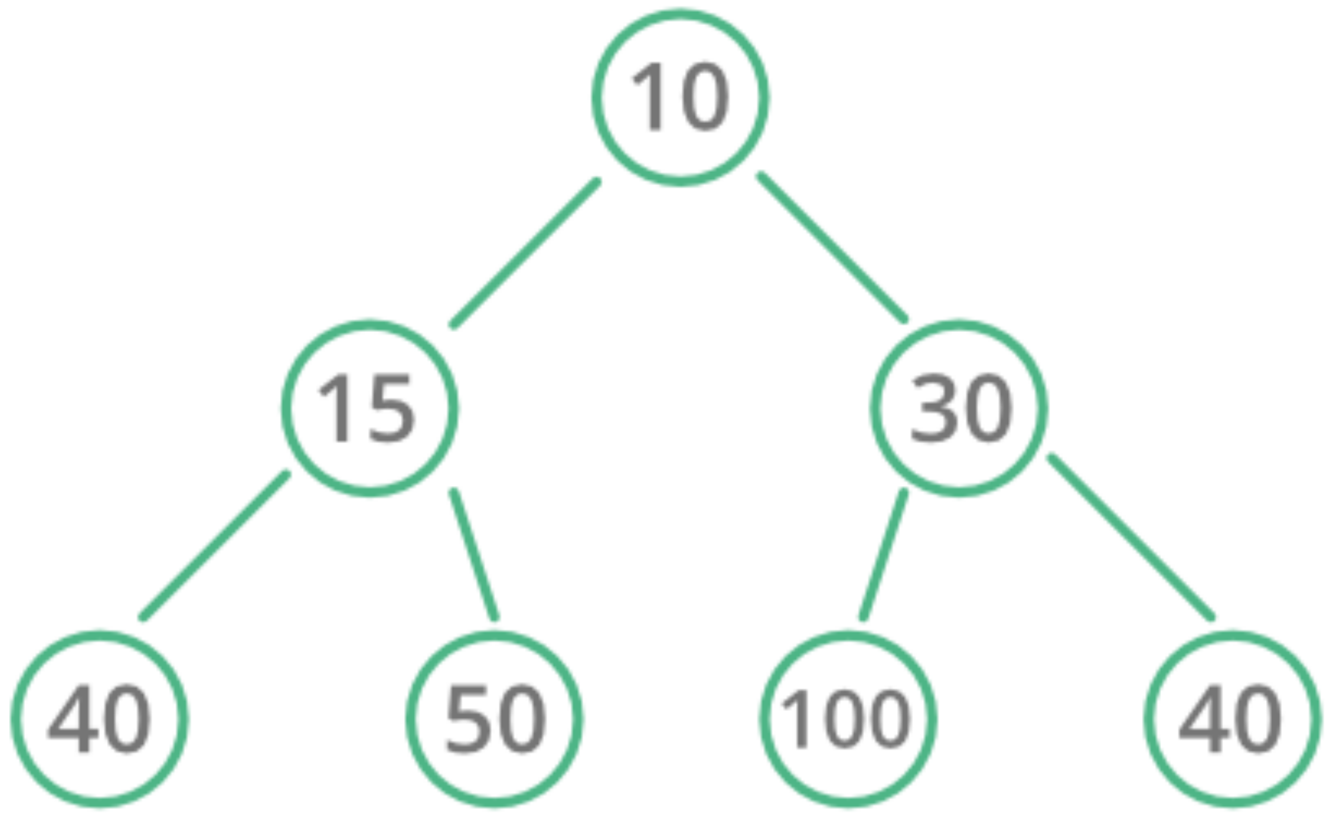
Heap ADT

★ Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- ★ always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- ★ always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

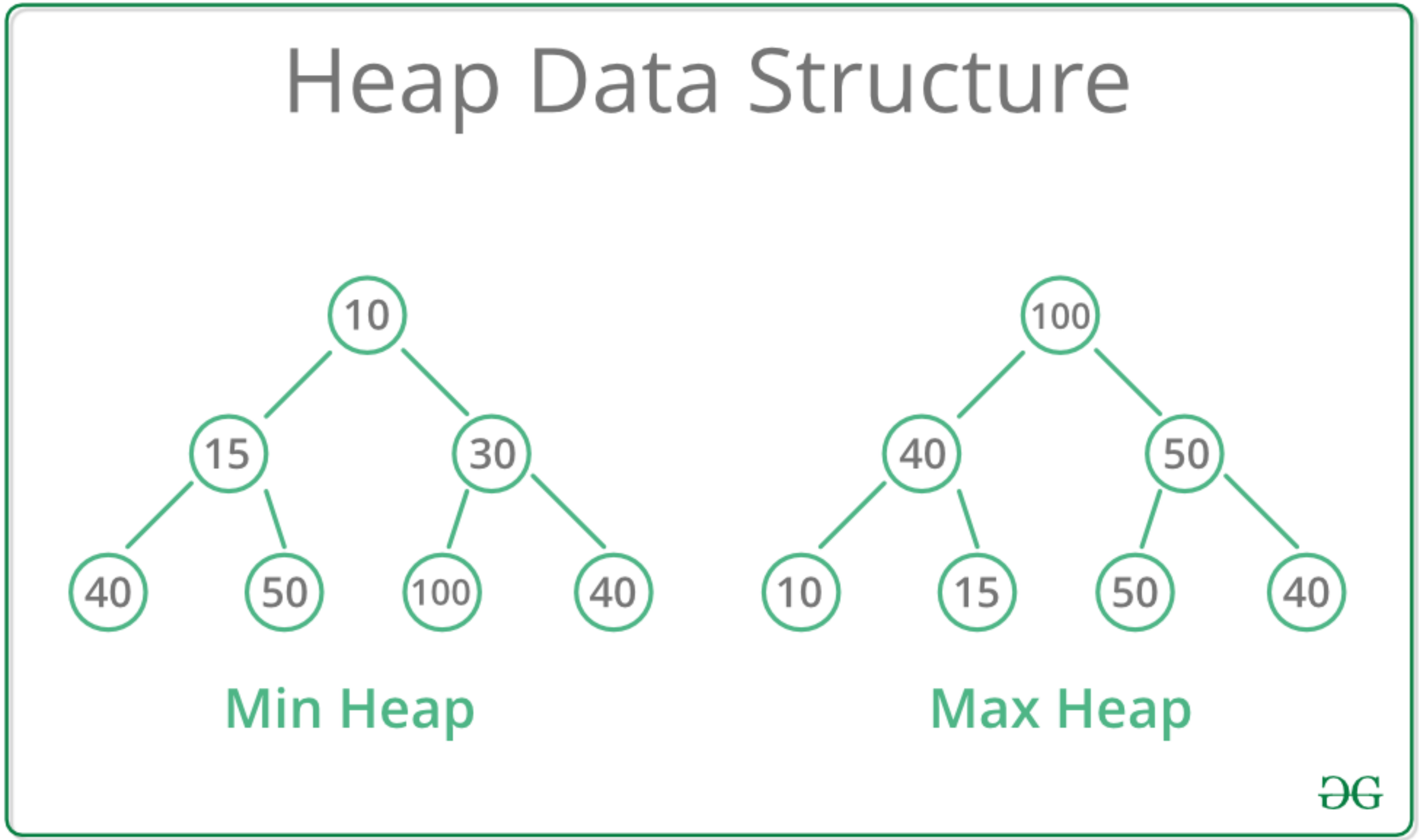


Max Heap



Min Heap

Heap ADT


Heap Data Structure
10
15
30
40
50
100
40
Min Heap
100
40
50
10
15
50
40
Max Heap

Heap Construction

- ★ Initialize an essentially complete binary tree from the keys given
- ★ Heapify the tree as following:
  - Start with the last (rightmost) parental node,
  - Exchange key K with its children if the parental dominance does not hold.
  - Check the parental dominance for the new position of K and repeat until the key K goes to right location.
  - Continue with other parental nodes up to the root.

Heap Construction

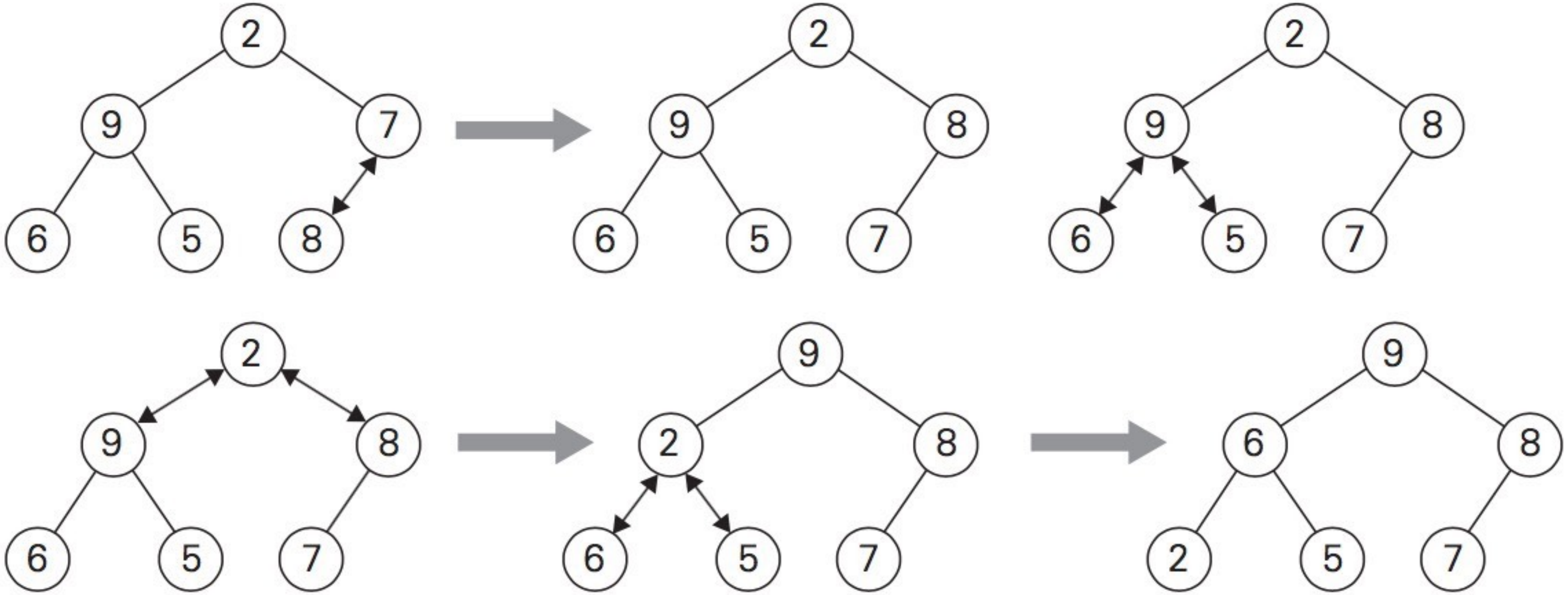
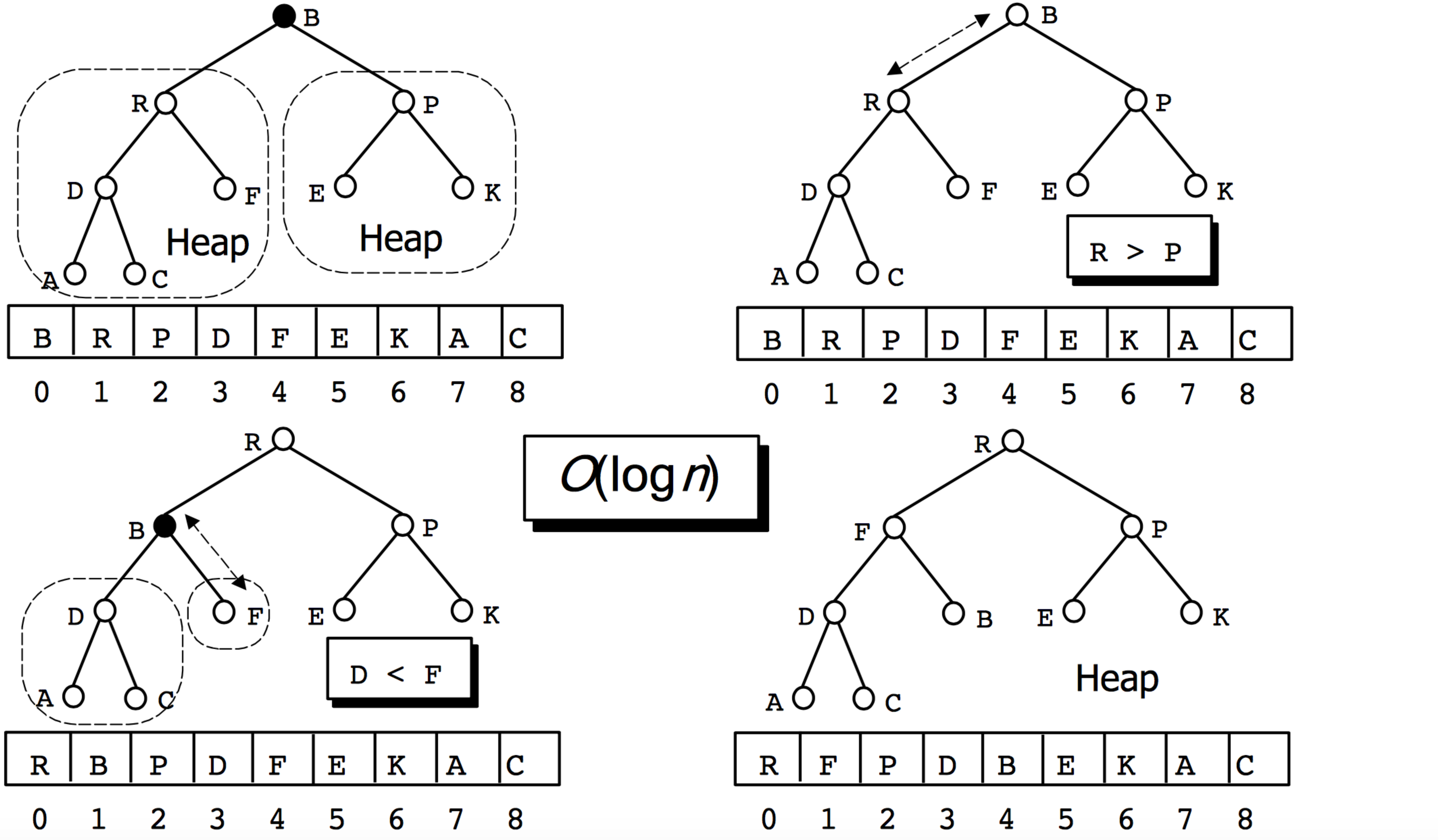


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

Heapify

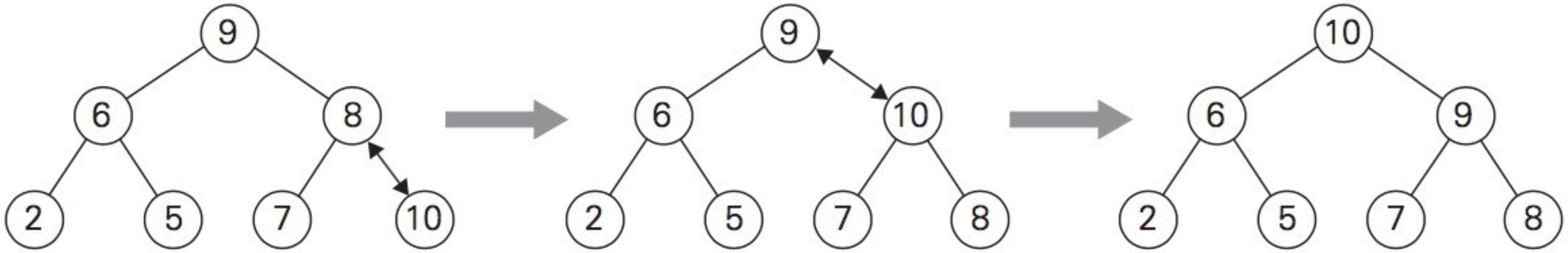

B
R
P
D
F
E
K
A
C
Heap
Heap
B R P D F E K A C
0 1 2 3 4 5 6 7 8
R > P
B R P D F E K A C
0 1 2 3 4 5 6 7 8
D < F
R B P D F E K A C
0 1 2 3 4 5 6 7 8
O(log n)
Heap
R F P D B E K A C
0 1 2 3 4 5 6 7 8

ALGORITHM *HeapBottomUp*$(H[1..n])$

//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
    $k \leftarrow i$; $v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    while not $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        if $j < n$ //there are two children
            if $H[j] < H[j+1]$ $j \leftarrow j+1$
        if $v \geq H[j]$
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j]$; $k \leftarrow j$
    $H[k] \leftarrow v$

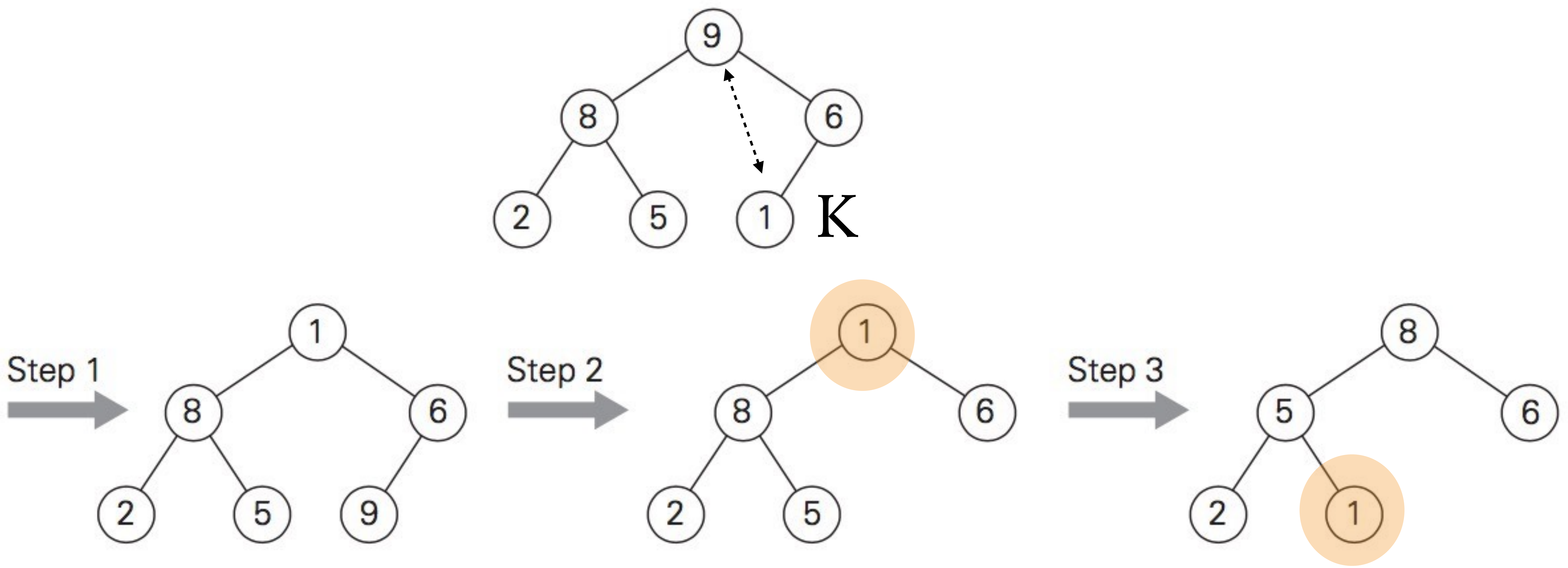
Inserting Key into Heap

- Insert the new key to the last leaf.
- Repeatedly swap the new key with its parent until the parental dominance is satisfied.
- Also be used as *top-down heap construction*



Deleting Key from Heap

- Swap the key to delete with the key K in last leaf.
- Delete the last leaf.
- Heapify the tree by sifting K down the tree

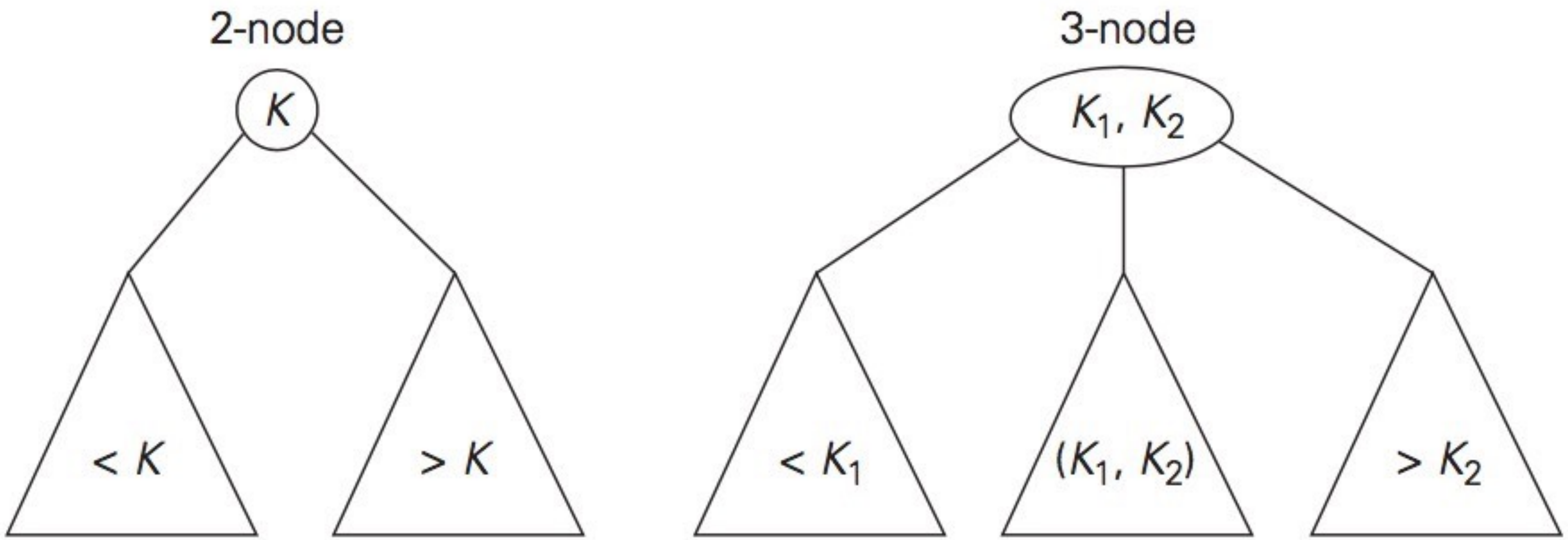


Heapsort

- ★ Stage 1 (heap construction):
  Construct a heap for a given array with HeapBottomUP().
- ★ Stage 2 (maximum deletion):
  Apply the root-deletion operation n−1 times to the remaining heap.
  - ★ The largest value (root) will be deleted first.
  - ★ Array elements are eliminated in decreasing order.
- ★ Since an element being deleted is placed last, the resulting array will be exactly the original array sorted in an increasing order.

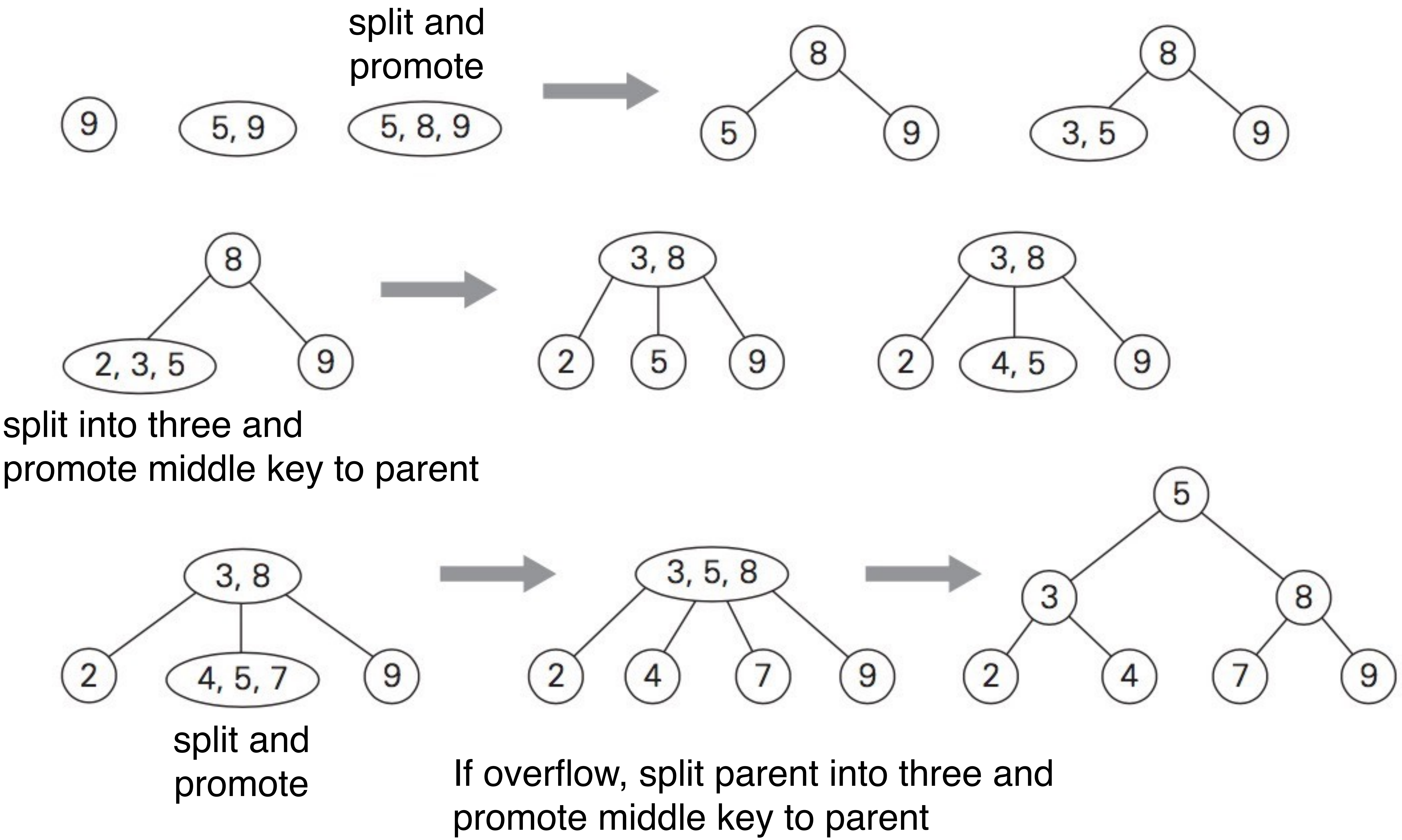
2-3 Tree

- A 2-3 tree is a search tree that
  - may have 2-nodes and 3-nodes
  - height-balanced (all leaves are on the same level)



2-3 Tree

Sequence Inserted: 9, 5, 8, 3, 2, 4, 7



Other Applications of Trees

- Red-Black Tree
- B Tree
- 2-3 Tree

ASCII Code

- ASCII characters use ***fixed-length codes*** (7 bits)
- For a textfile with 1000 characters, what is the file size?
- Suppose the file contains only 7 unique characters 'A' to 'G'
  - Can we use 3 bits instead of 8 bits?
  - What is the file size and ***compression ratio*** now?

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

Huffman Code

- Can do better by using ***Prefix-free codes*** like ***Huffman Code***

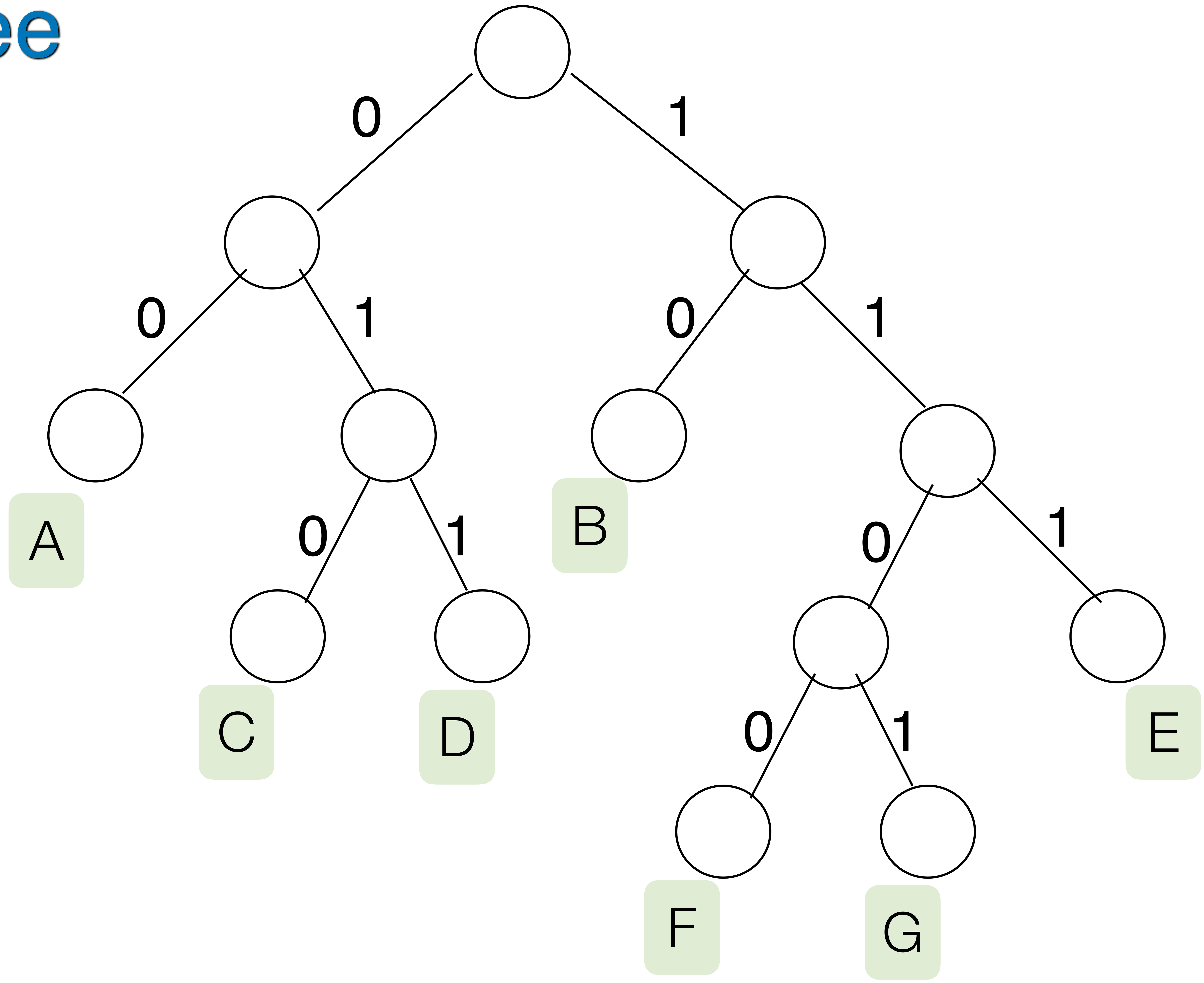
Symbol	Codewords
A	01
B	11
C	000
D	001
E	101
F	1000
G	1001

No codeword is a prefix of another codeword.

Average bits per symbol $\sum_{i=1}^{n} l_i p_i$

Huffman Tree

David Huffman (1952)

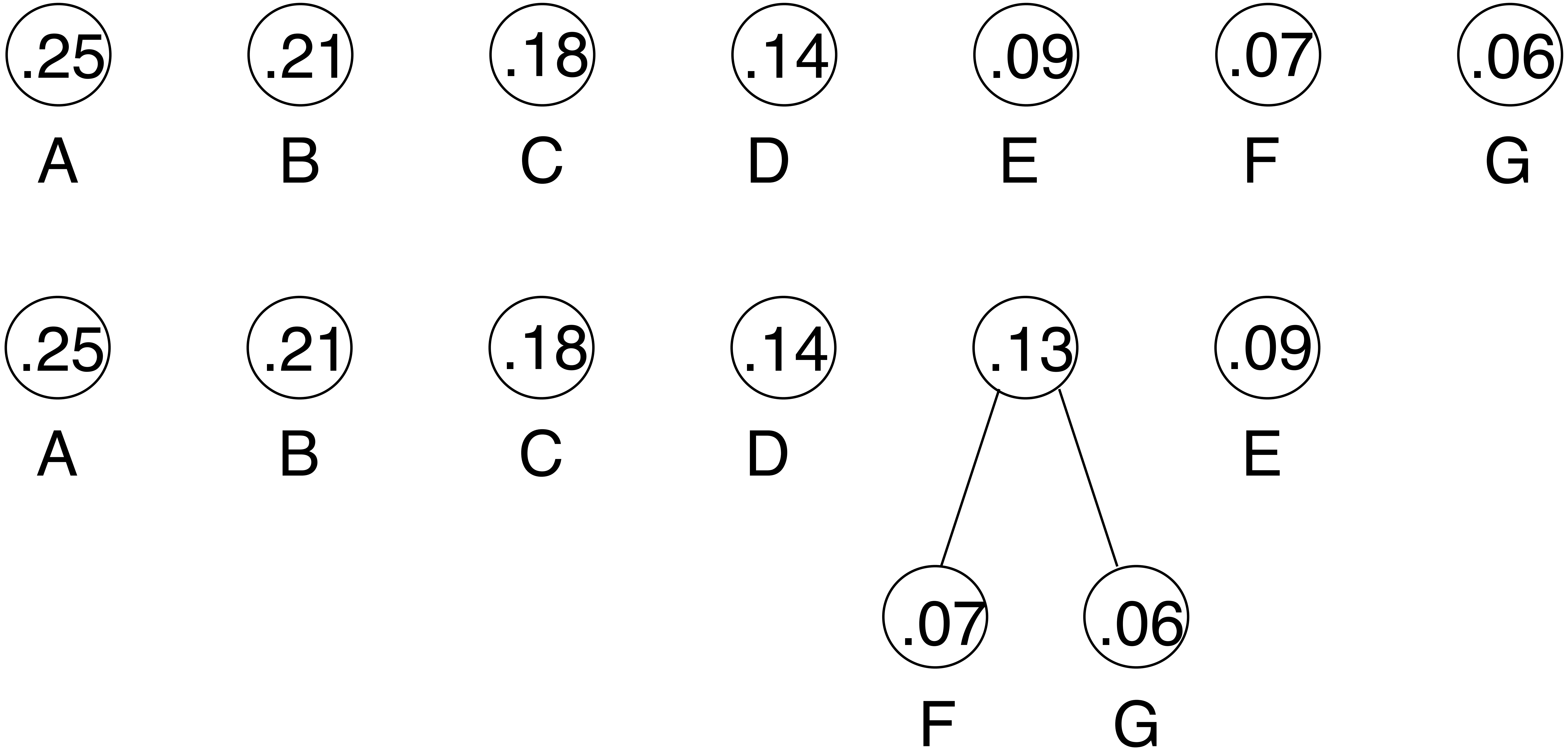


Huffman Algorithm

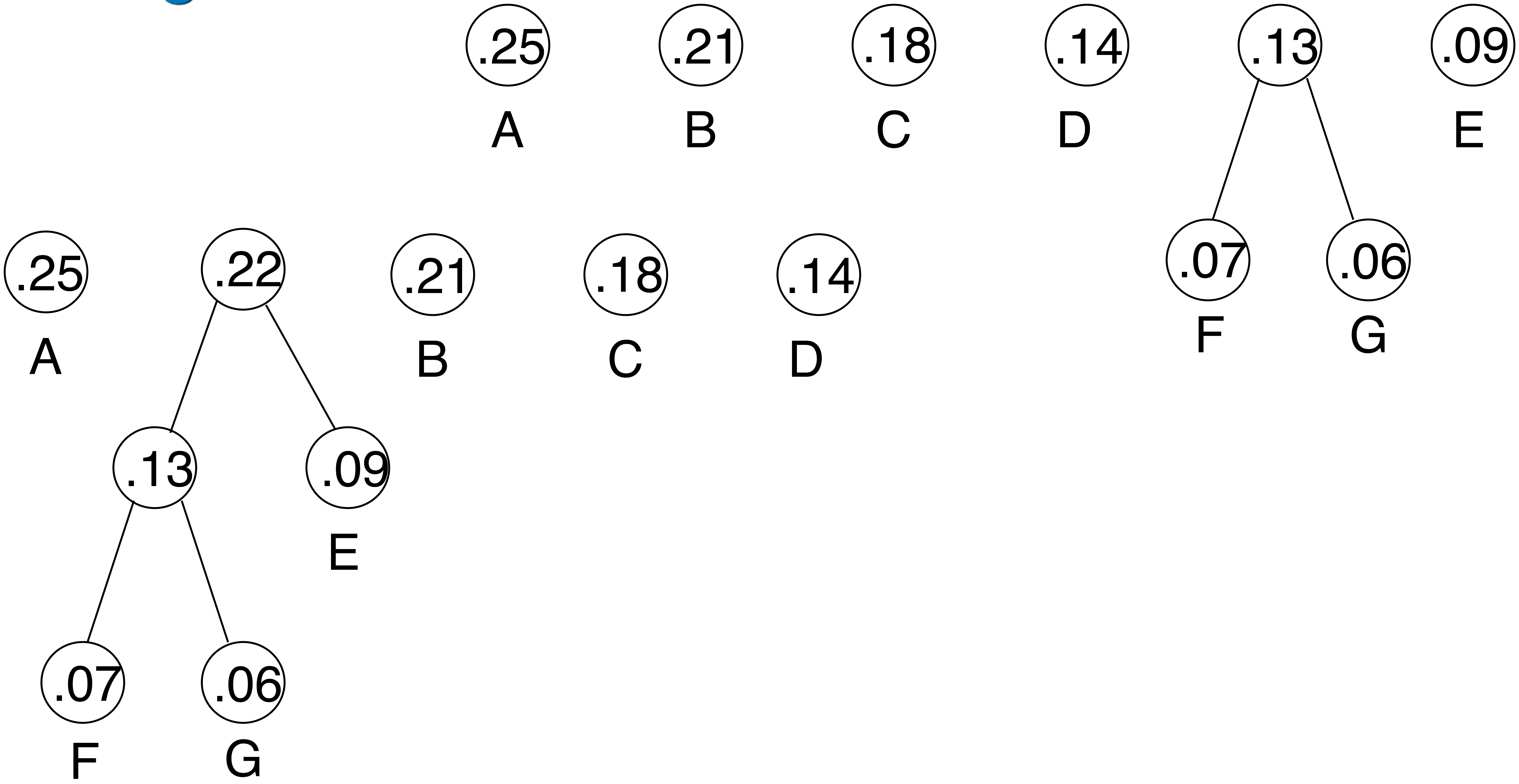
- Greedy algorithm to construct a Huffman tree of n symbols
  - Start from n single-node trees and combine them to a single tree
  - Generate short/long codewords to high/low frequency symbols

Symbol	Frequency
A	0.25
B	0.21
C	0.18
D	0.14
E	0.09
F	0.07
G	0.06

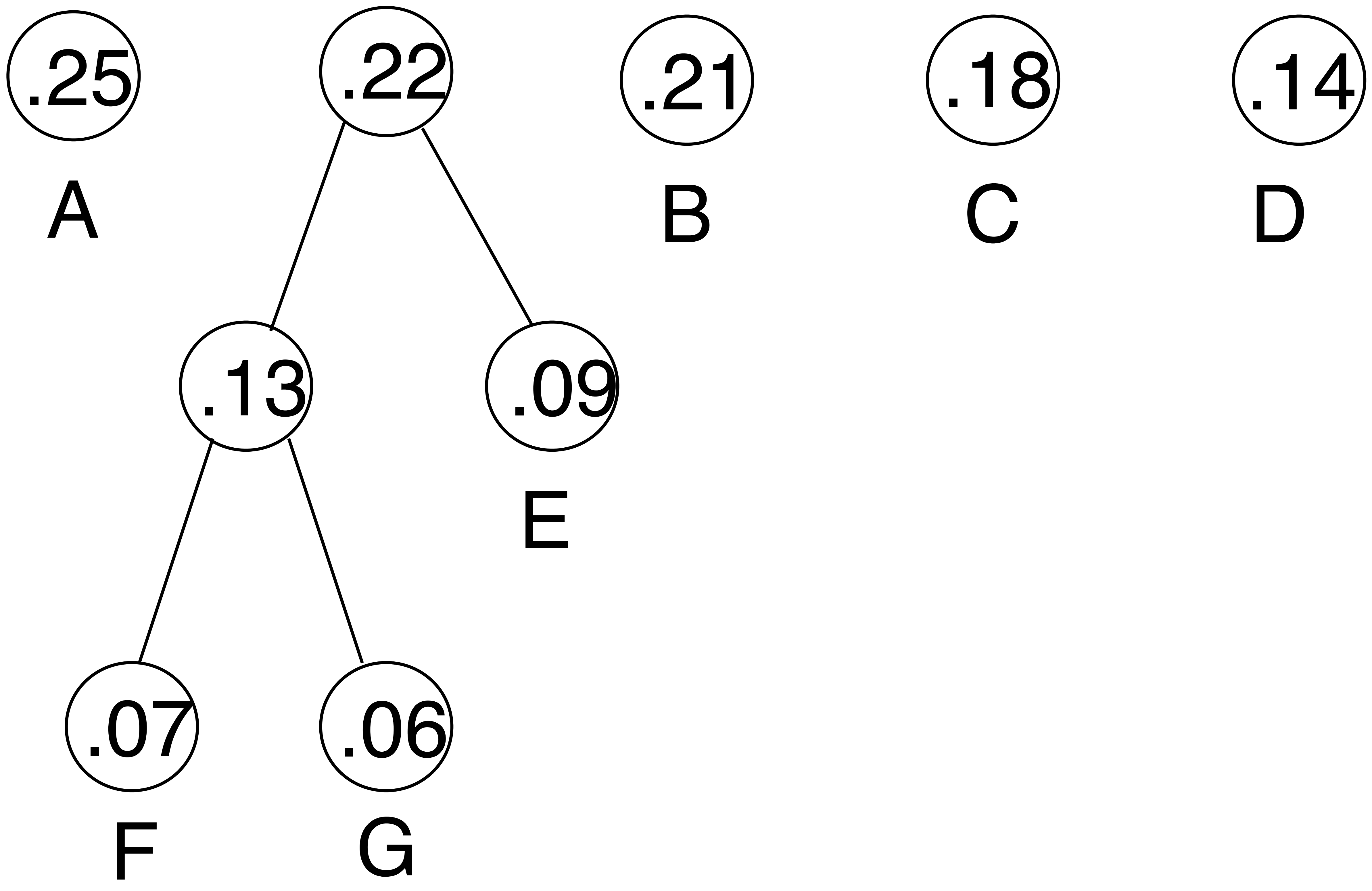
Huffman Algorithm

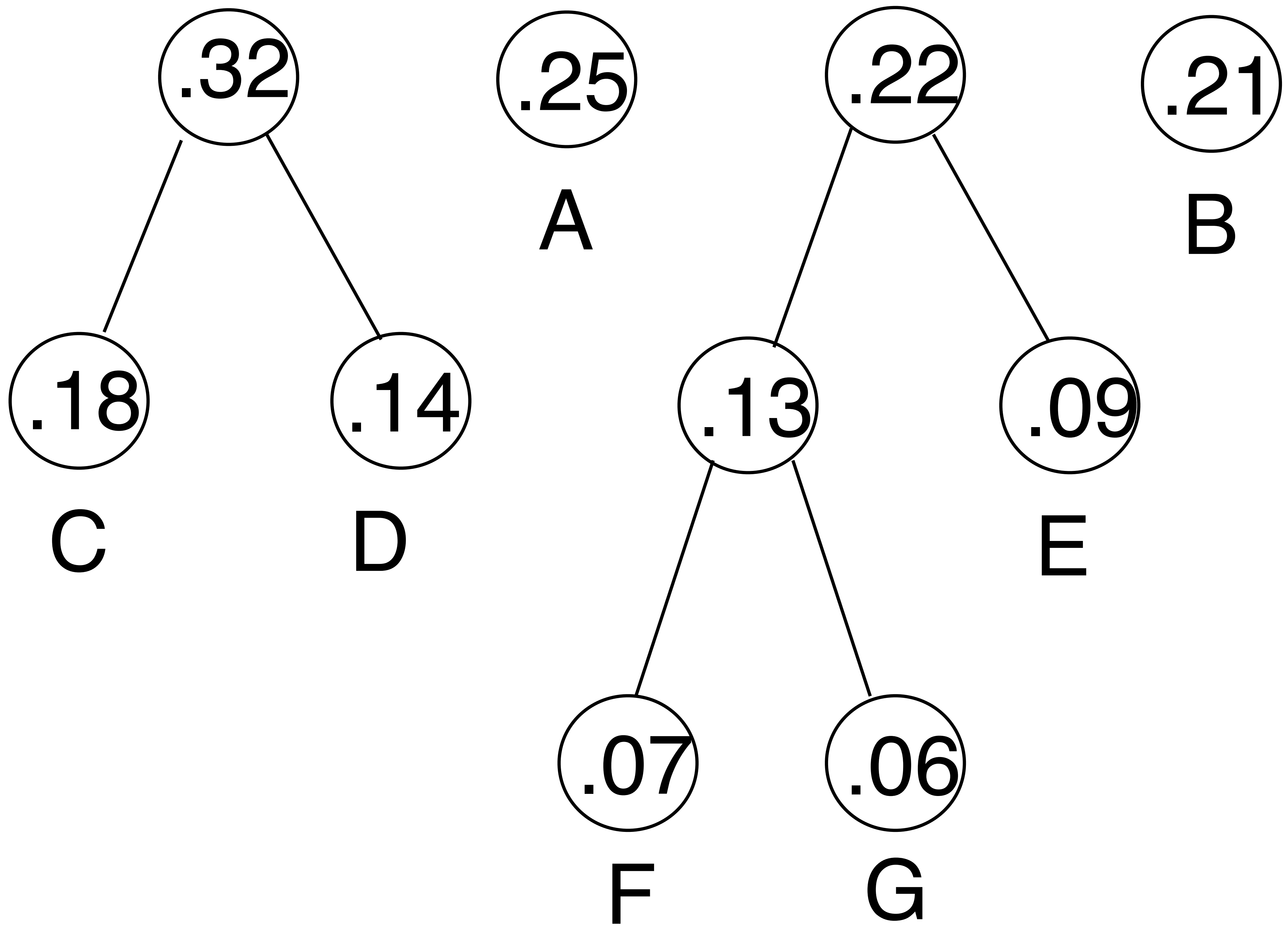


Huffman Algorithm

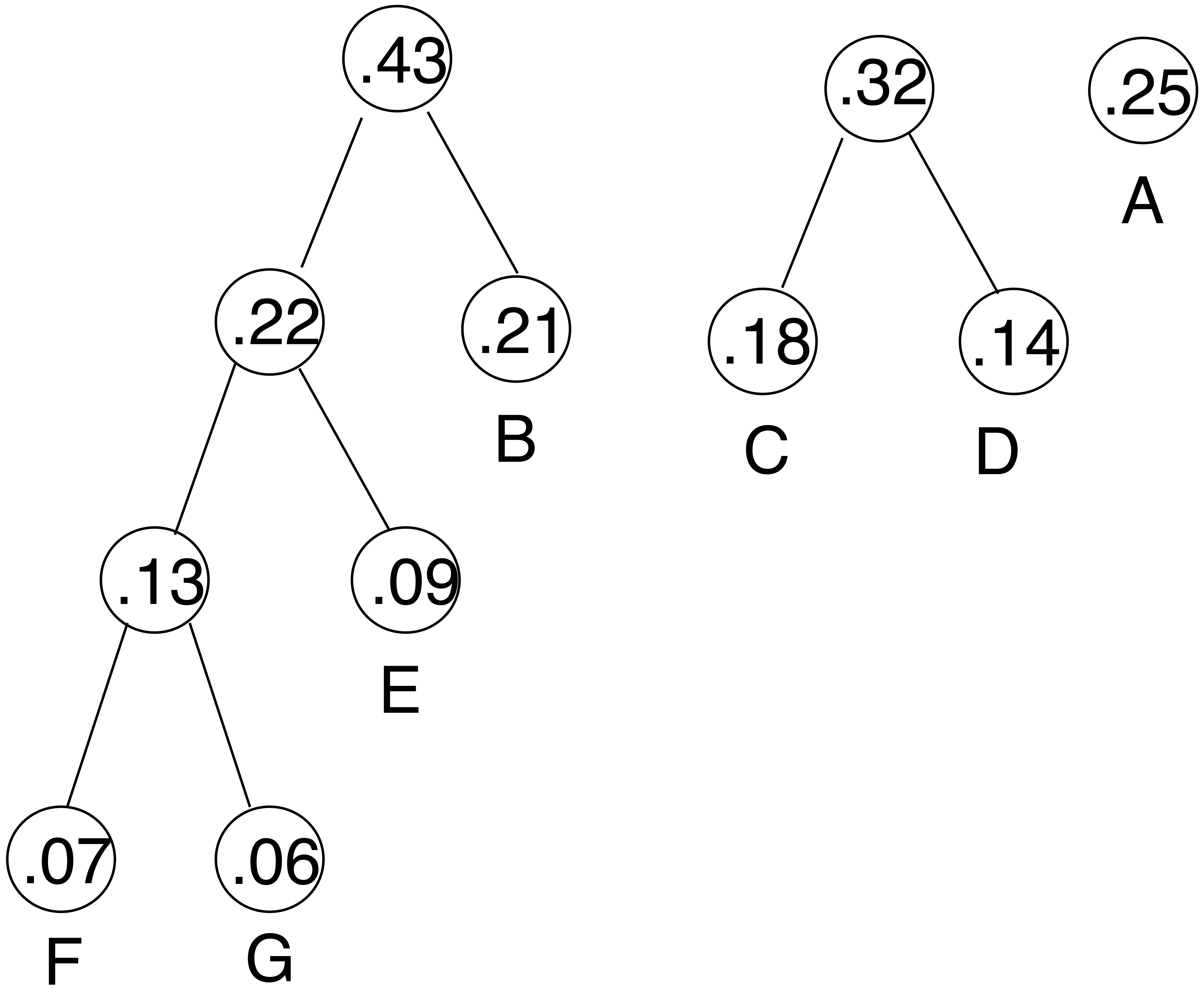


Huffman Algorithm

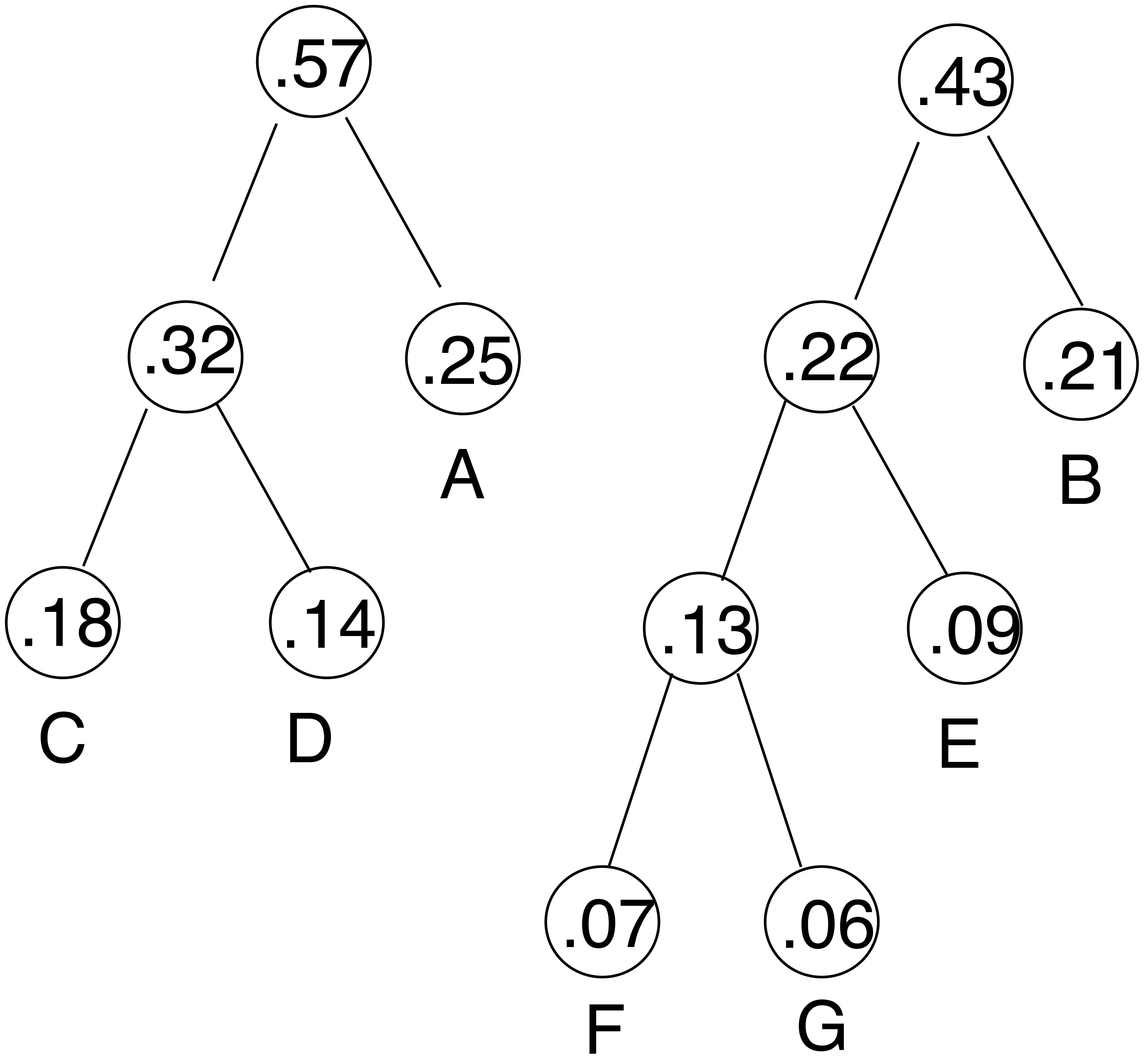




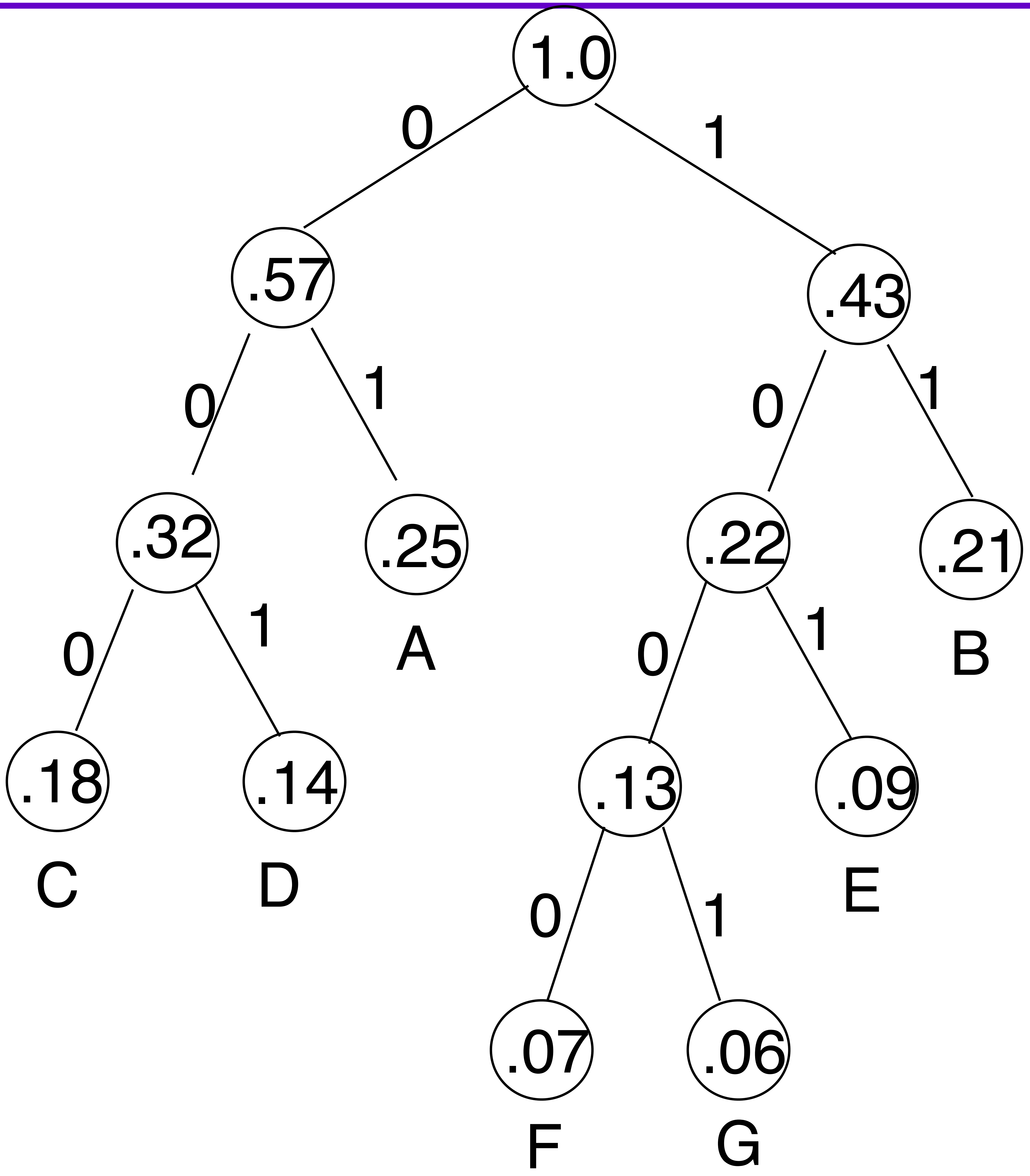
Huffman Algorithm



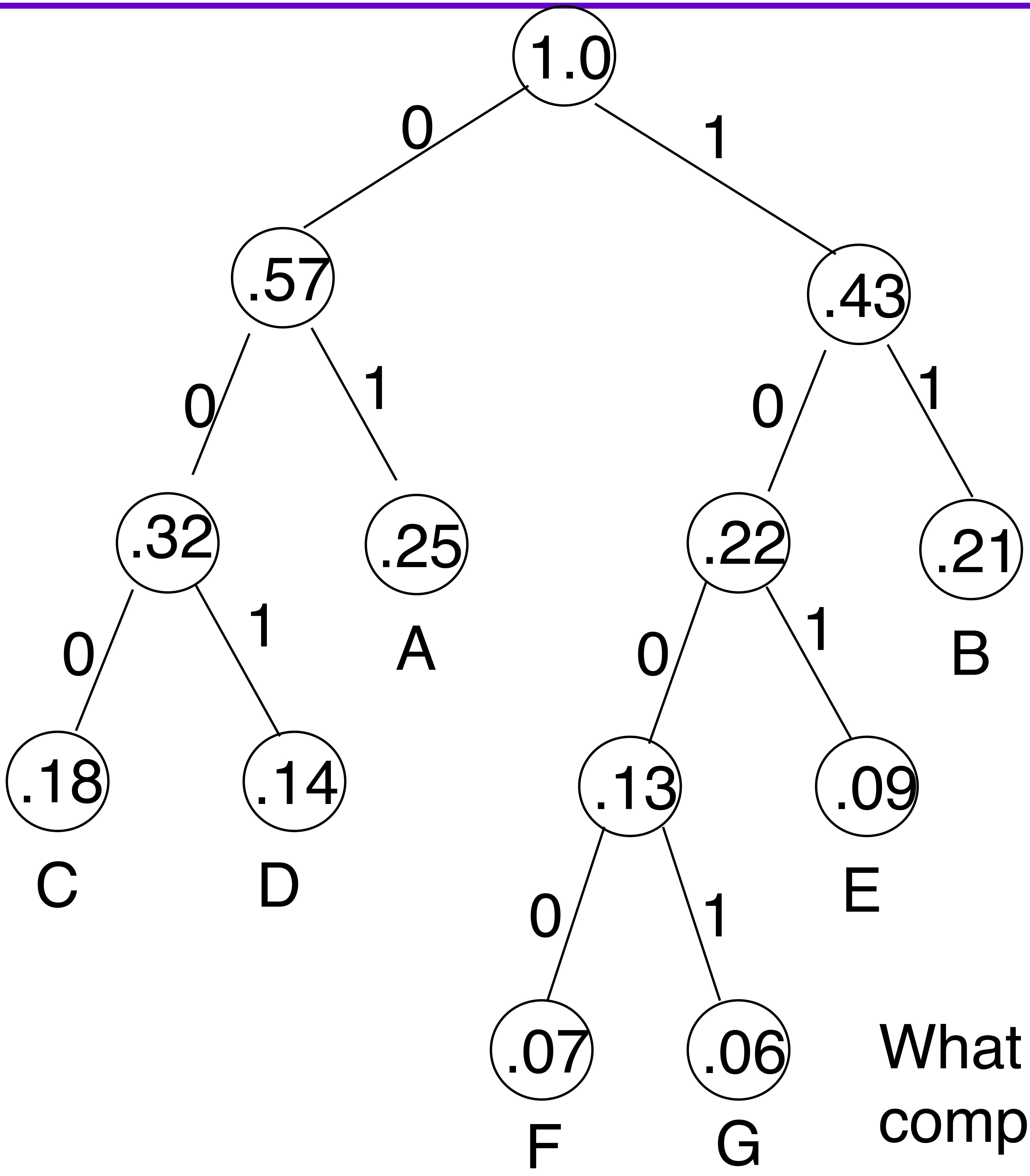
Huffman Algorithm



Result



Result



What is the best-case compression ratio for Huffman code?

Note on Huffman Algorithm

- ★ If you sort the initial trees from smallest to largest weight, a different huffman tree will be generated.
- ★ The same huffman tree must be used for both encoding and decoding Huffman tree

Huffman's Algorithm

Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

Huffman's Algorithm: In-Class exercise

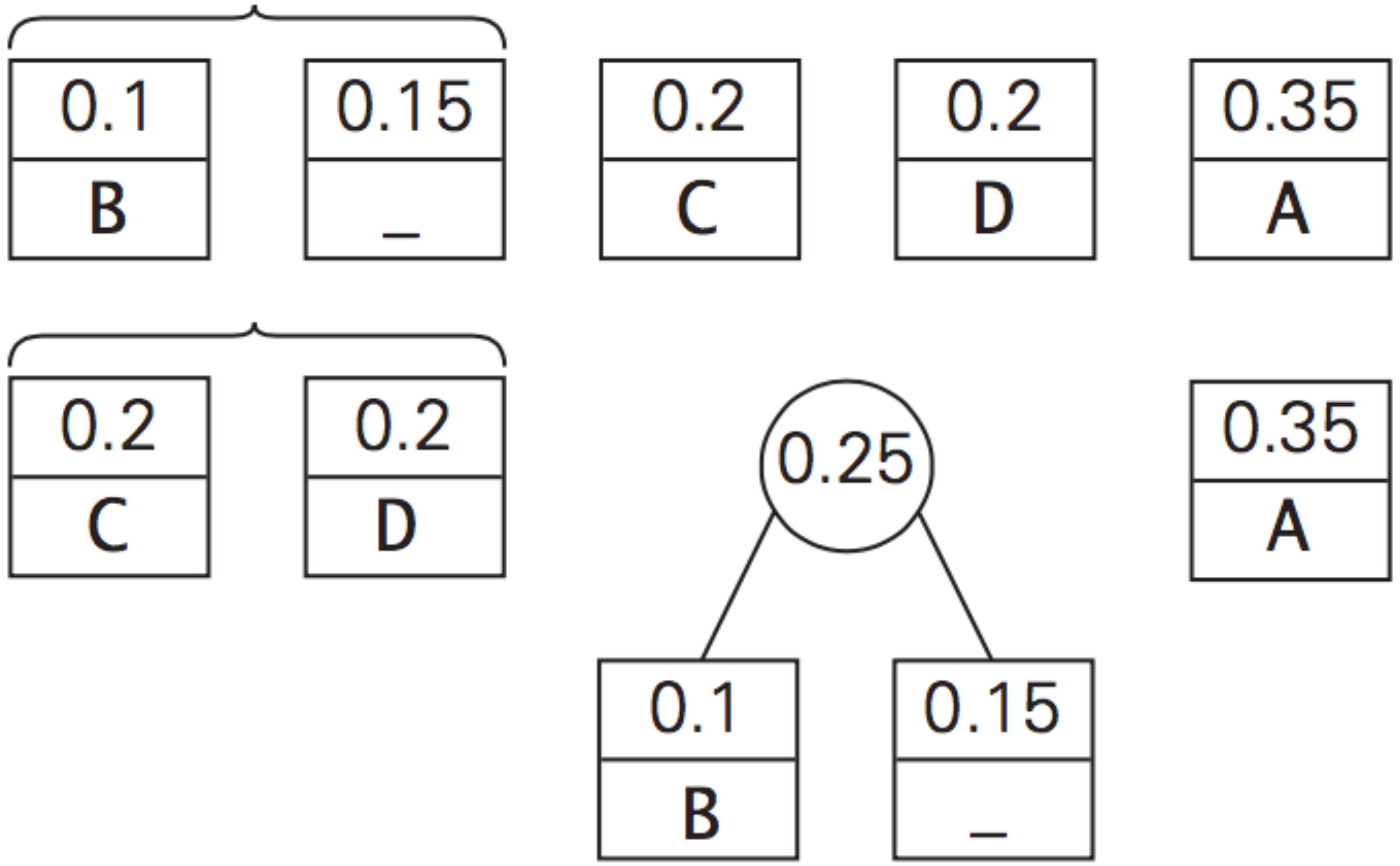
- Create a Huffman tree and the resulting codewords from the following occurrence frequencies

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

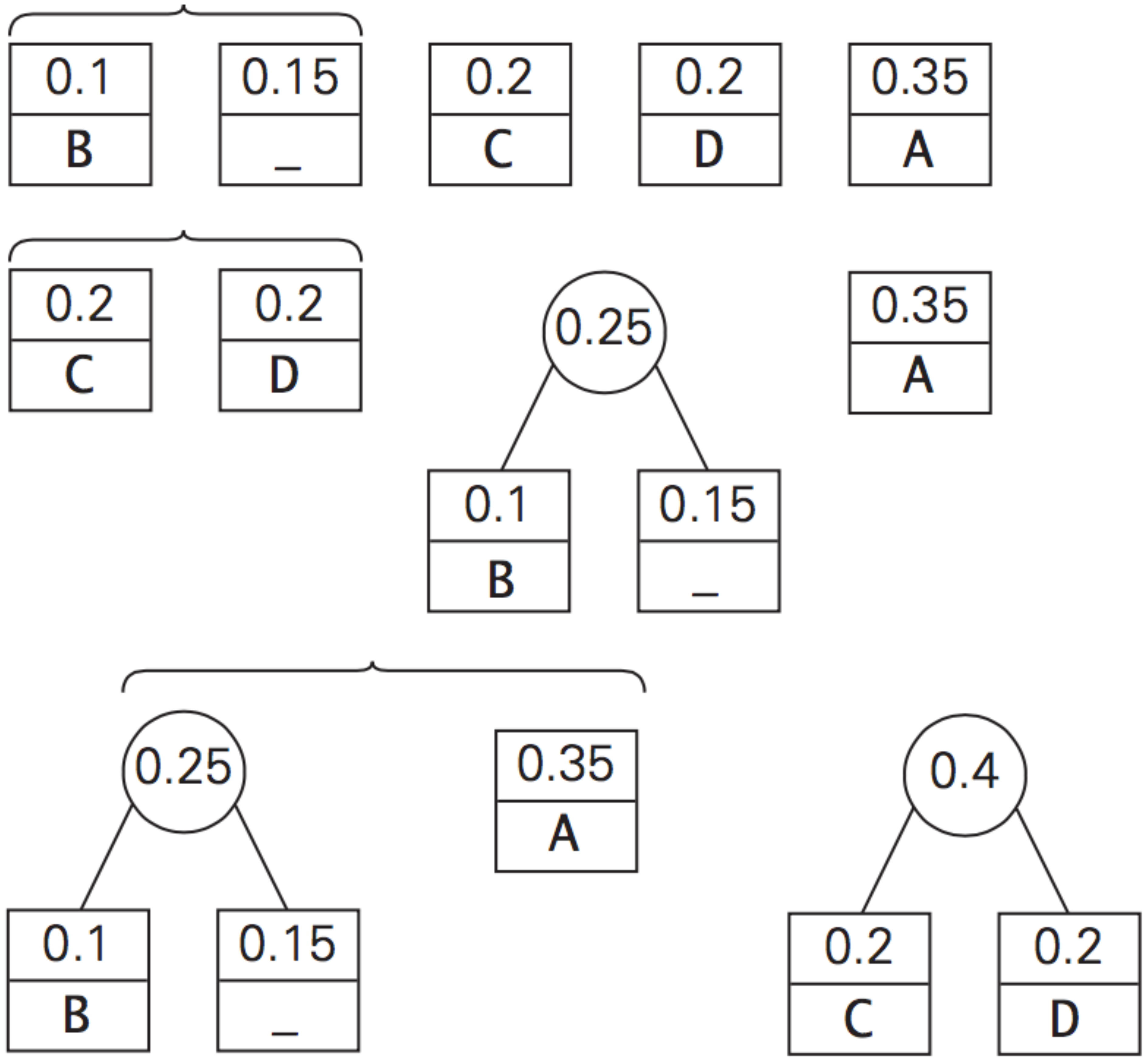
  - What are the huffman codes for this input?
  - What is the average number of bits per symbol?

Huffman’s Algorithm: In-Class exercise

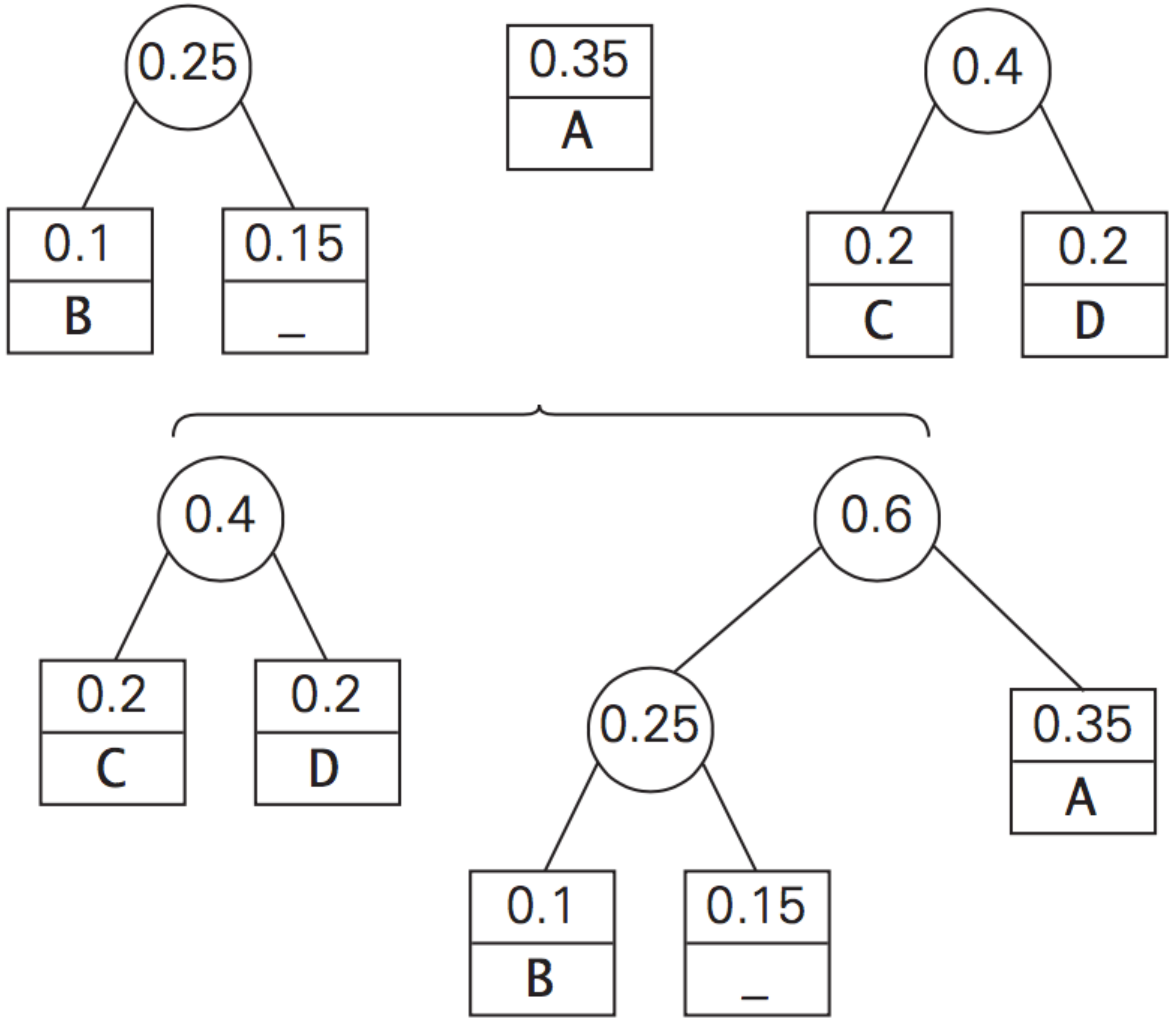
Greedy Technique



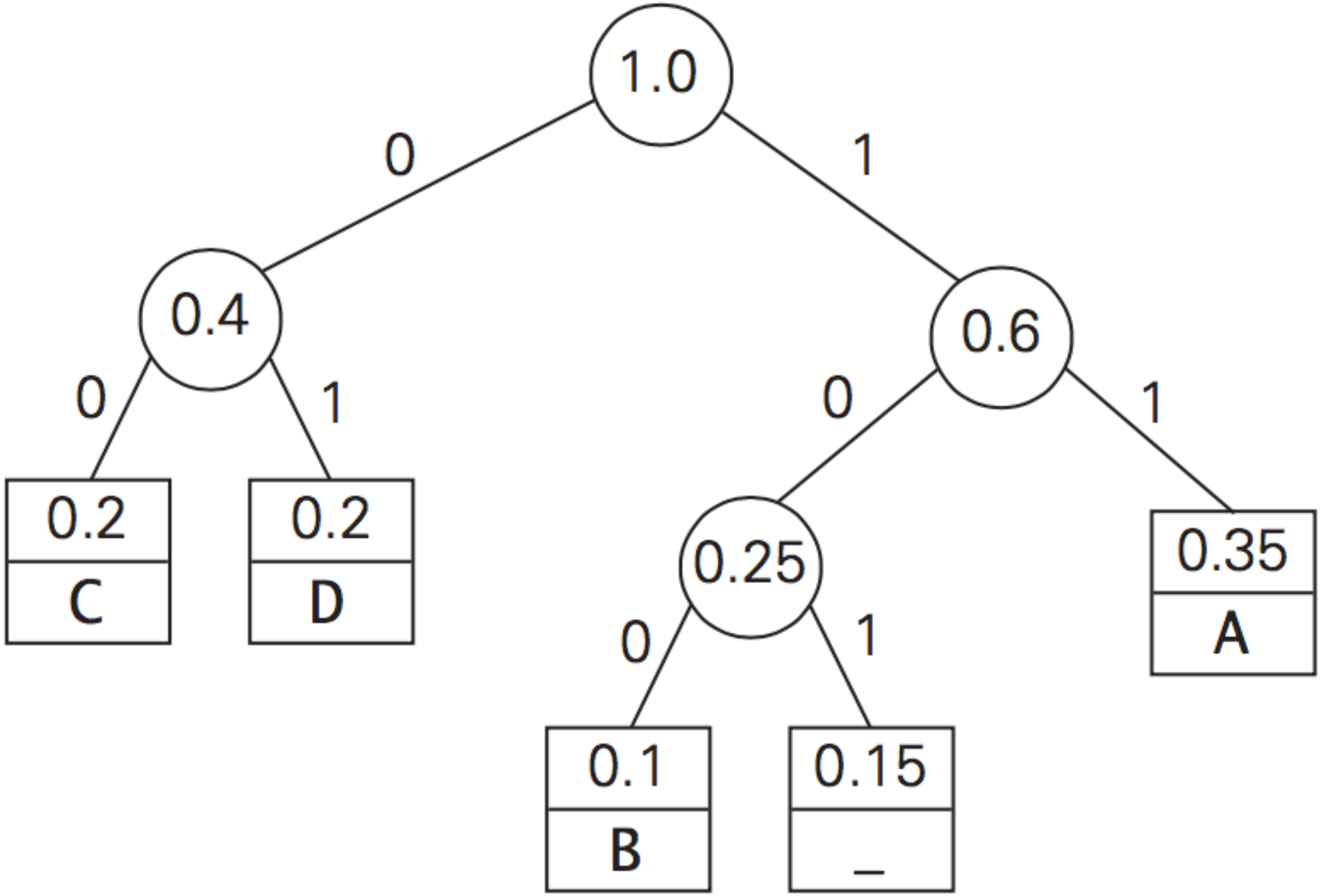
Huffman's Algorithm: In-Class exercise



Huffman's Algorithm: In-Class exercise



Huffman's Algorithm: In-Class exercise



Huffman's Algorithm: In-Class exercise

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

Questions and Answers