

# COMP90038

# Algorithms and Complexity

Lecture 21: Huffman Encoding for Data Compression  
(with thanks to Harald Søndergaard & Michael Kirley)

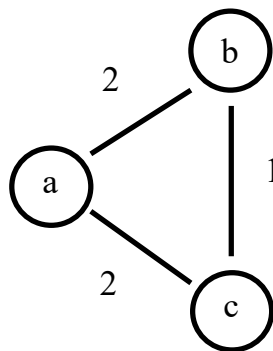
Casey Myers  
Casey.Myers@unimelb.edu.au  
David Caro Building (Physics) 274

## Review from Lecture 20: Greedy Algorithms

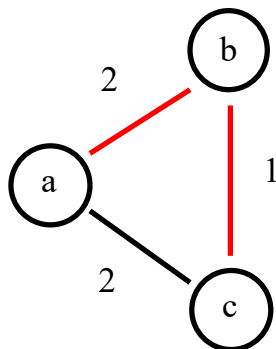
- In general we cannot expect **locally best** choices to yield **globally best** outcomes.
- However, there are some well-known algorithms that rely on the greedy approach, being both correct and fast.
- In other cases, for hard problems, a greedy algorithm can sometimes serve as an acceptable **approximation algorithm** (we will discuss approximation algorithms in Week 12).
- Here we shall look at
  - Prim's algorithm for finding **minimum spanning trees**
  - Dijkstra's algorithm for **single-source shortest paths**.

## Review from Lecture 20: Greedy Algorithms

- Prim's algorithm for finding **minimum spanning trees**
- Dijkstra's algorithm for **single-source shortest paths**.

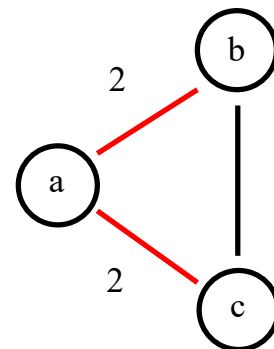


Prim's Algorithm



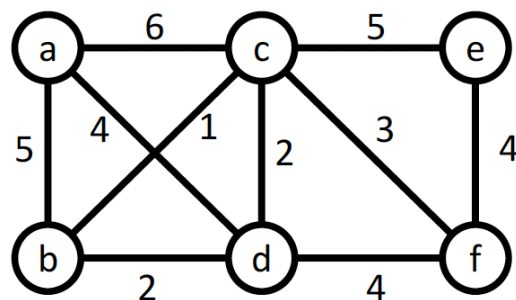
Finding the minimum spanning tree

Dijkstra's Algorithm



Finding the single-source shortest paths

## Review from Lecture 20: Prim's Algorithm



```

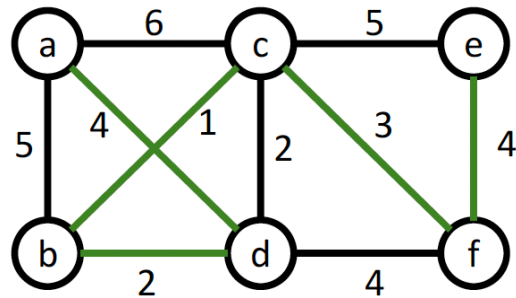
function PRIM( $\langle V, E \rangle$ )
  for each  $v \in V$  do
     $cost[v] \leftarrow \infty$ 
     $prev[v] \leftarrow nil$ 
  pick initial node  $v_0$ 
   $cost[v_0] \leftarrow 0$ 
   $Q \leftarrow \text{INITPRIORITYQUEUE}(V)$   $\triangleright$  priorities are cost values
  while  $Q$  is non-empty do
     $u \leftarrow \text{EJECTMIN}(Q)$ 
    for each  $(u, w) \in E$  do
      if  $w$  in  $Q$  and  $weight(u, w) < cost[w]$  then
         $cost[w] \leftarrow weight(u, w)$ 
         $prev[w] \leftarrow u$ 
         $\text{UPDATE}(Q, w, cost[w])$   $\triangleright$  rearranges priority queue

```

- On the first loop, we only create the table.

[illegible]

# Review from Lecture 20: Prim's Algorithm

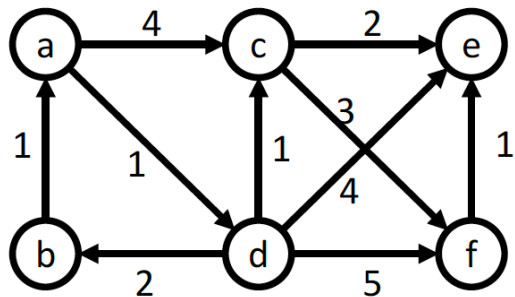


```
function PRIM( $\langle V, E \rangle$ )  
  for each  $v \in V$  do  
     $cost[v] \leftarrow \infty$   
     $prev[v] \leftarrow nil$   
  pick initial node  $v_0$   
   $cost[v_0] \leftarrow 0$   
   $Q \leftarrow INITPRIORITYQUEUE(V)$   $\triangleright$  priorities are cost values  
  while  $Q$  is non-empty do  
     $u \leftarrow EJECTMIN(Q)$   
    for each  $(u, w) \in E$  do  
      if  $w$  in  $Q$  and  $weight(u, w) < cost[w]$  then  
         $cost[w] \leftarrow weight(u, w)$   
         $prev[w] \leftarrow u$   
        UPDATE( $Q, w, cost[w]$ )  $\triangleright$  rearranges priority queue
```

- The resulting tree is  $\{a, d, b, c, f, e\}$ .

Tree $T$		$a$	$b$	$c$	$d$	$e$	$f$
	$cost$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	$prev$	$nil$	$nil$	$nil$	$nil$	$nil$	$nil$
	$cost$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	$prev$	$nil$	$nil$	$nil$	$nil$	$nil$	$nil$
$a$	$cost$		5	6	4	$\infty$	$\infty$
	$prev$		$a$	$a$	$a$	$nil$	$nil$
$a, d$	$cost$		2	2		$\infty$	4
	$prev$		$d$	$d$		$nil$	$d$
$a, d, b$	$cost$			1		$\infty$	4
	$prev$			$b$		$nil$	$d$
$a, d, b, c$	$cost$					5	3
	$prev$					$c$	$c$
$a, d, b, c, f$	$cost$					4	
	$prev$					$f$	
$a, d, b, c, f, e$	$cost$						
	$prev$						

# Review from Lecture 20: Dijkstra's Algorithm



```

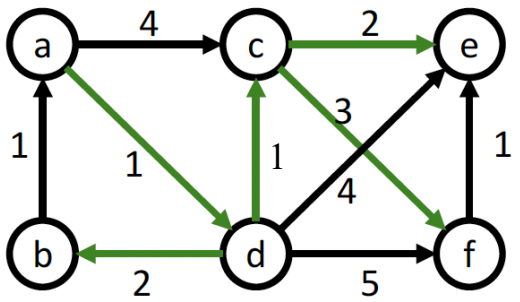
function DIJKSTRA( $\langle V, E \rangle, v_0$ )
  for each  $v \in V$  do
     $dist[v] \leftarrow \infty$ 
     $prev[v] \leftarrow nil$ 
   $dist[v_0] \leftarrow 0$ 
   $Q \leftarrow \text{INITPRIORITYQUEUE}(V)$   $\triangleright$  priorities are distances
  while  $Q$  is non-empty do
     $u \leftarrow \text{EJECTMIN}(Q)$ 
    for each  $(u, w) \in E$  do
      if  $w$  in  $Q$  and  $dist[u] + weight(u, w) < dist[w]$  then
         $dist[w] \leftarrow dist[u] + weight(u, w)$ 
         $prev[w] \leftarrow u$ 
         $\text{UPDATE}(Q, w, dist[w])$   $\triangleright$  rearranges priority queue

```

- On the first loop, we only create the table.

[illegible]

# Review from Lecture 20: Dijkstra’s Algorithm



- The complete tree is  $\{a, d, c, b, e, f\}$ .

```
function DIJKSTRA( $\langle V, E \rangle, v_0$ )  
  for each  $v \in V$  do  
     $dist[v] \leftarrow \infty$   
     $prev[v] \leftarrow nil$   
   $dist[v_0] \leftarrow 0$   
   $Q \leftarrow INITPRIORITYQUEUE(V)$        $\triangleright$  priorities are distances  
  while  $Q$  is non-empty do  
     $u \leftarrow EJECTMIN(Q)$   
    for each  $(u, w) \in E$  do  
      if  $w$  in  $Q$  and  $dist[u] + weight(u, w) < dist[w]$  then  
         $dist[w] \leftarrow dist[u] + weight(u, w)$   
         $prev[w] \leftarrow u$   
        UPDATE( $Q, w, dist[w]$ )   $\triangleright$  rearranges priority queue
```

Tree $T$		$a$	$b$	$c$	$d$	$e$	$f$
	$cost$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	$prev$	$nil$	$nil$	$nil$	$nil$	$nil$	$nil$
	$cost$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
	$prev$	$nil$	$nil$	$nil$	$nil$	$nil$	$nil$
$a$	$cost$		$\infty$	4	1	$\infty$	$\infty$
	$prev$		$nil$	$a$	$a$	$nil$	$nil$
$a, d$	$cost$		3	2		5	6
	$prev$		$d$	$d$		$d$	$d$
$a, d, c$	$cost$		3			4	5
	$prev$		$d$			$c$	$c$
$a, d, c, b$	$cost$					4	5
	$prev$					$c$	$c$
$a, d, c, b, e$	$cost$						5
	$prev$						$c$
$a, d, c, b, e, f$	$cost$						
	$prev$						

# Data Compression

- From an information-theoretic point of view, most computer files contain a lot of redundancy.
- Compression is used to store files in less space.
- For text files, savings up to 50% are common.
- For binary files, savings up to 90% are common.
- Savings in space mean savings in time for files transmission.



# Run-Length Encoding

- For a text that has long runs of repeated characters, we could compress by counting the runs:

*AAAABBBBAABBBBBBCCCCCCCCDABCBAAABBBBBCCCD*

can then be encoded as

*4A3BAA5B8CDABCB3A4B3CD*

- For English text this is not very useful.
- For binary files it can be very good.

[illegible]

# Variable-Length Encoding

- Instead of using, say, 8 bits for characters (as ASCII does), assign character codes in such a way that common characters have shorter codes.
- For example, *E* could have code 0.
- But then no other character code can start with 0.
- In general, no character's code should be a prefix of some other character's code (unless we somehow put separators between characters, which would take up space).

# Variable-Length Encoding

- Suppose we count symbols and find these numbers of occurrences:
- Here are some sensible codes that we may use for symbols:

Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

Symbol	Code
A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

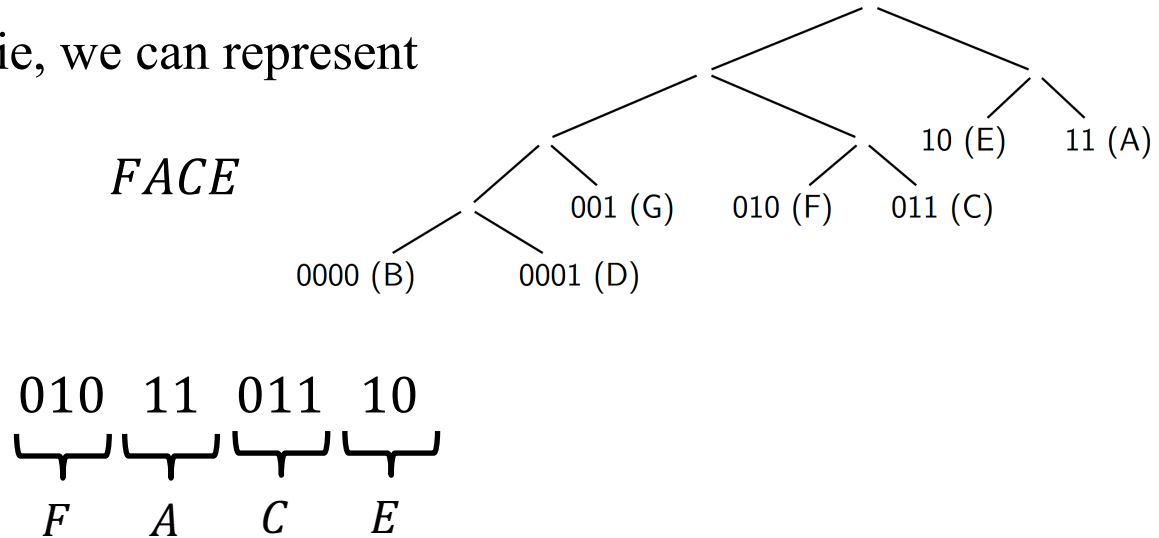
- 
- ```

graph TD
    Root(( )) --- L1L(( ))
    Root --- L1R(( ))
    L1L --- L2L(( ))
    L1L --- L2R(( ))
    L2L --- L3L(( ))
    L2L --- L3R(( ))
    L2R --- L3R2(( ))
    L2R --- L3R3(( ))
    L3L --- L4L(( ))
    L3L --- L4R(( ))
    L3R2 --- L4R2(( ))
    L3R2 --- L4R3(( ))
    L3R3 --- L4R32(( ))
    L3R3 --- L4R33(( ))
    L4L --- L5L(( ))
    L4L --- L5R(( ))
    L4R2 --- L5R2(( ))
    L4R2 --- L5R3(( ))
    L4R32 --- L5R32(( ))
    L4R32 --- L5R33(( ))
    L4R33 --- L5R332(( ))
    L4R33 --- L5R333(( ))
    L5L --- L6L(( ))
    L5L --- L6R(( ))
    L5R2 --- L6R2(( ))
    L5R2 --- L6R3(( ))
    L5R32 --- L6R32(( ))
    L5R32 --- L6R33(( ))
    L5R332 --- L6R332(( ))
    L5R332 --- L6R333(( ))
    L5R333 --- L6R3332(( ))
    L5R333 --- L6R3333(( ))
    L6L --- L7L(( ))
    L6L --- L7R(( ))
    L6R2 --- L7R2(( ))
    L6R2 --- L7R3(( ))
    L6R32 --- L7R32(( ))
    L6R32 --- L7R33(( ))
    L6R332 --- L7R332(( ))
    L6R332 --- L7R333(( ))
    L6R3332 --- L7R3332(( ))
    L6R3332 --- L7R3333(( ))
    L6R3333 --- L7R33332(( ))
    L6R3333 --- L7R33333(( ))
    L7L --- L8L(( ))
    L7L --- L8R(( ))
    L7R2 --- L8R2(( ))
    L7R2 --- L8R3(( ))
    L7R32 --- L8R32(( ))
    L7R32 --- L8R33(( ))
    L7R332 --- L8R332(( ))
    L7R332 --- L8R333(( ))
    L7R3332 --- L8R3332(( ))
    L7R3332 --- L8R3333(( ))
    L7R33332 --- L8R33332(( ))
    L7R33332 --- L8R33333(( ))
    L8L --- L9L(( ))
    L8L --- L9R(( ))
    L8R2 --- L9R2(( ))
    L8R2 --- L9R3(( ))
    L8R32 --- L9R32(( ))
    L8R32 --- L9R33(( ))
    L8R332 --- L9R332(( ))
    L8R332 --- L9R333(( ))
    L8R3332 --- L9R3332(( ))
    L8R3332 --- L9R3333(( ))
    L8R33332 --- L9R33332(( ))
    L8R33332 --- L9R33333(( ))
    L9L --- L10L(( ))
    L9L --- L10R(( ))
    L9R2 --- L10R2(( ))
    L9R2 --- L10R3(( ))
    L9R32 --- L10R32(( ))
    L9R32 --- L10R33(( ))
    L9R332 --- L10R332(( ))
    L9R332 --- L10R333(( ))
    L9R3332 --- L10R3332(( ))
    L9R3332 --- L10R3333(( ))
    L9R33332 --- L10R33332(( ))
    L9R33332 --- L10R33333(( ))
    L10L --- L11L(( ))
    L10L --- L11R(( ))
    L10R2 --- L11R2(( ))
    L10R2 --- L11R3(( ))
    L10R32 --- L11R32(( ))
    L10R32 --- L11R33(( ))
    L10R332 --- L11R332(( ))
    L10R332 --- L11R333(( ))
    L10R3332 --- L11R3332(( ))
    L10R3332 --- L11R3333(( ))
    L10R33332 --- L11R33332(( ))
    L10R33332 --- L11R33333(( ))
    L11L --- L12L(( ))
    L11L --- L12R(( ))
    L11R2 --- L12R2(( ))
    L11R2 --- L12R3(( ))
    L11R32 --- L12R32(( ))
    L11R32 --- L12R33(( ))
    L11R332 --- L12R332(( ))
    L11R332 --- L12R333(( ))
    L11R3332 --- L12R3332(( ))
    L11R3332 --- L12R3333(( ))
    L11R33332 --- L12R33332(( ))
    L11R33332 --- L12R33333(( ))
    L12L --- L13L(( ))
    L12L --- L13R(( ))
    L12R2 --- L13R2(( ))
    L12R2 --- L13R3(( ))
    L12R32 --- L13R32(( ))
    L12R32 --- L13R33(( ))
    L12R332 --- L13R332(( ))
    L12R332 --- L13R333(( ))
    L12R3332 --- L13R3332(( ))
    L12R3332 --- L13R3333(( ))
    L12R33332 --- L13R33332(( ))
    L12R33332 --- L13R33333(( ))
    L13L --- L14L(( ))
    L13L --- L14R(( ))
    L13R2 --- L14R2(( ))
    L13R2 --- L14R3(( ))
    L13R32 --- L14R32(( ))
    L13R32 --- L14R33(( ))
    L13R332 --- L14R332(( ))
    L13R332 --- L14R333(( ))
    L13R3332 --- L14R3332(( ))
    L13R3332 --- L14R3333(( ))
    L13R33332 --- L14R33332(( ))
    L13R33332 --- L14R33333(( ))
    L14L --- L15L(( ))
    L14L --- L15R(( ))
    L14R2 --- L15R2(( ))
    L14R2 --- L15R3(( ))
    L14R32 --- L15R32(( ))
    L14R32 --- L15R33(( ))
    L14R332 --- L15R332(( ))
    L14R332 --- L15R333(( ))
    L14R3332 --- L15R3332(( ))
    L14R3332 --- L15R3333(( ))
    L14R33332 --- L15R33332(( ))
    L14R33332 --- L15R33333(( ))
    L15L --- L16L(( ))
    L15L --- L16R(( ))
    L15R2 --- L16R2(( ))
    L15R2 --- L16R3(( ))
    L15R32 --- L16R32(( ))
    L15R32 --- L16R33(( ))
    L15R332 --- L16R332(( ))
    L15R332 --- L16R333(( ))
    L15R3332 --- L16R3332(( ))
    L15R3332 --- L16R3333(( ))
    L15R33332 --- L16R33332(( ))
    L15R33332 --- L16R33333(( ))
    L16L --- L17L(( ))
    L16L --- L17R(( ))
    L16R2 --- L17R2(( ))
    L16R2 --- L17R3(( ))
    L16R32 --- L17R32(( ))
    L16R32 --- L17R33(( ))
    L16R332 --- L17R332(( ))
    L16R332 --- L17R333(( ))
    L16R3332 --- L17R3332(( ))
    L16R3332 --- L17R3333(( ))
    L16R33332 --- L17R33332(( ))
    L16R33332 --- L17R33333(( ))
    L17L --- L18L(( ))
    L17L --- L18R(( ))
    L17R2 --- L18R2(( ))
    L17R2 --- L18R3(( ))
    L17R32 --- L18R32(( ))
    L17R32 --- L18R33(( ))
    L17R332 --- L18R332(( ))
    L17R332 --- L18R333(( ))
    L17R3332 --- L18R3332(( ))
    L17R3332 --- L18R3333(( ))
    L17R33332 --- L18R33332(( ))
    L17R33332 --- L18R33333(( ))
    L18L --- L19L(( ))
    L18L --- L19R(( ))
    L18R2 --- L19R2(( ))
    L18R2 --- L19R3(( ))
    L18R32 --- L19R32(( ))
    L18R32 --- L19R33(( ))
    L18R332 --- L19R332(( ))
    L18R332 --- L19R333(( ))
    L18R3332 --- L19R3332(( ))
    L18R3332 --- L19R3333(( ))
    L18R33332 --- L19R33332(( ))
    L18R33332 --- L19R33333(( ))
    L19L --- L20L(( ))
    L19L --- L20R(( ))
    L19R2 --- L20R2(( ))
    L19R2 --- L20R3(( ))
    L19R32 --- L20R32(( ))
    L19R32 --- L20R33(( ))
    L19R332 --- L20R332(( ))
    L19R332 --- L20R333(( ))
    L19R3332 --- L20R3332(( ))
    L19R3332 --- L20R3333(( ))
    L19R33332 --- L20R33332(( ))
    L19R33332 --- L20R33333(( ))
    L20L --- L21L(( ))
    L20L --- L21R(( ))
    L20R2 --- L21R2(( ))
    L20R2 --- L21R3(( ))
    L20R32 --- L21R32(( ))
    L20R32 --- L21R33(( ))
    L20R332 --- L21R332(( ))
    L20R332 --- L21R333(( ))
    L20R3332 --- L21R3332(( ))
    L20R3332 --- L21R3333(( ))
    L20R33332 --- L21R33332(( ))
    L20R33332 --- L21R33333(( ))
    L21L --- L22L(( ))
    L21L --- L22R(( ))
    L21R2 --- L22R2(( ))
    L21R2 --- L22R3(( ))
    L21R32 --- L22R32(( ))
    L21R32 --- L22R33(( ))
    L21R332 --- L22R332(( ))
    L21R332 --- L22R333(( ))
    L21R3332 --- L22R3332(( ))
    L21R3332 --- L22R3333(( ))
    L21R33332 --- L22R33332(( ))
    L21R33332 --- L22R33333(( ))
    L22L --- L23L(( ))
    L22L --- L23R(( ))
    L22R2 --- L23R2(( ))
    L22R2 --- L23R3(( ))
    L22R32 --- L23R32(( ))
    L22R32 --- L23R33(( ))
    L22R332 --- L23R332(( ))
    L22R332 --- L23R333(( ))
    L22R3332 --- L23R3332(( ))
    L22R3332 --- L23R3333(( ))
    L22R33332 --- L23R33332
```

# Variable-Length Encoding

- With the codes given by the trie, we can represent

in just 10 bits:



- If we were to assign 3 bits per character, would take 12 bits.

## Encoding a Message

- We shall shortly see how to design the codes for symbols, taking symbol frequencies into account.
- Once we have a table of codes, encoding is straightforward.
- For example, to encode '*BAGGED*' simply concatenate the codes for *B*, *A*, *G*, *G*, *E* and *D*:

000011001001100001

|   |      |
|---|------|
| A | 11   |
| B | 0000 |
| C | 011  |
| D | 0001 |
| E | 10   |
| F | 010  |
| G | 001  |

# Huffman Encoding: Choosing the Codes

- Sometimes (for example for common English text) we may know the frequencies of letters fairly well.
- If we don't know about frequencies then we can still count all characters in the given text as a first step.
- But how do we assign codes to the characters once we know their frequencies?
- By repeatedly selecting the two smallest weights and fusing them.
- This is **Huffman's algorithm**—another example of a **greedy method**.
- The resulting tree is a **Huffman tree**.



# Huffman Trees: Example

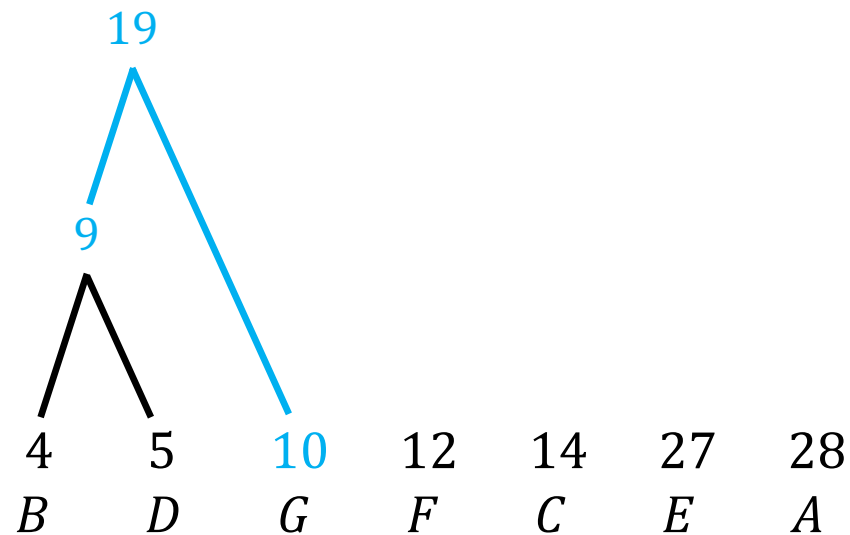
|          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|
| 4        | 5        | 10       | 12       | 14       | 27       | 28       |
| <i>B</i> | <i>D</i> | <i>G</i> | <i>F</i> | <i>C</i> | <i>E</i> | <i>A</i> |



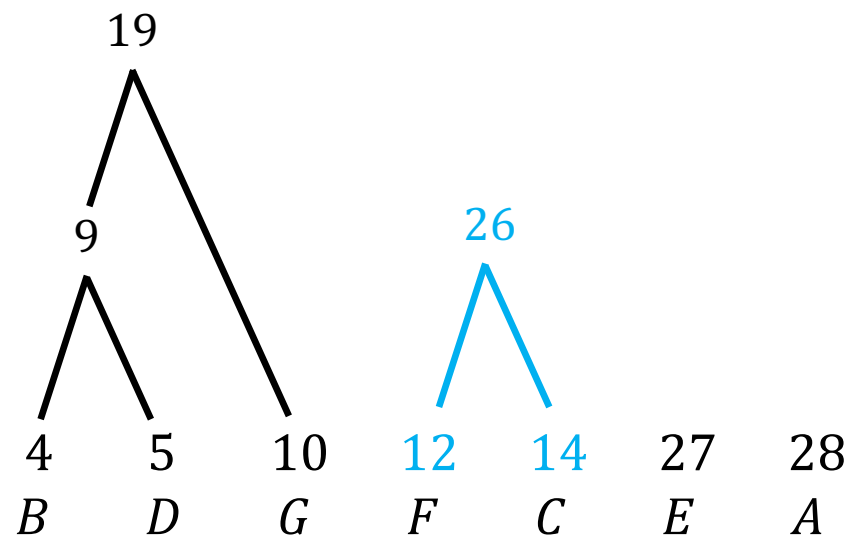
# Huffman Trees: Example



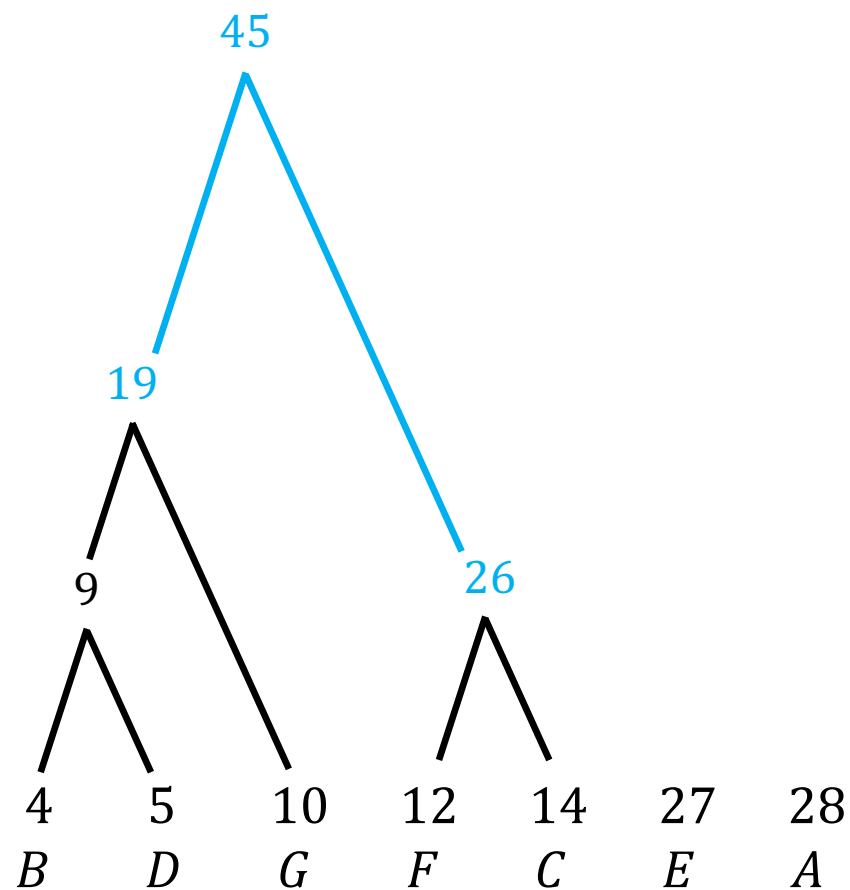
# Huffman Trees: Example



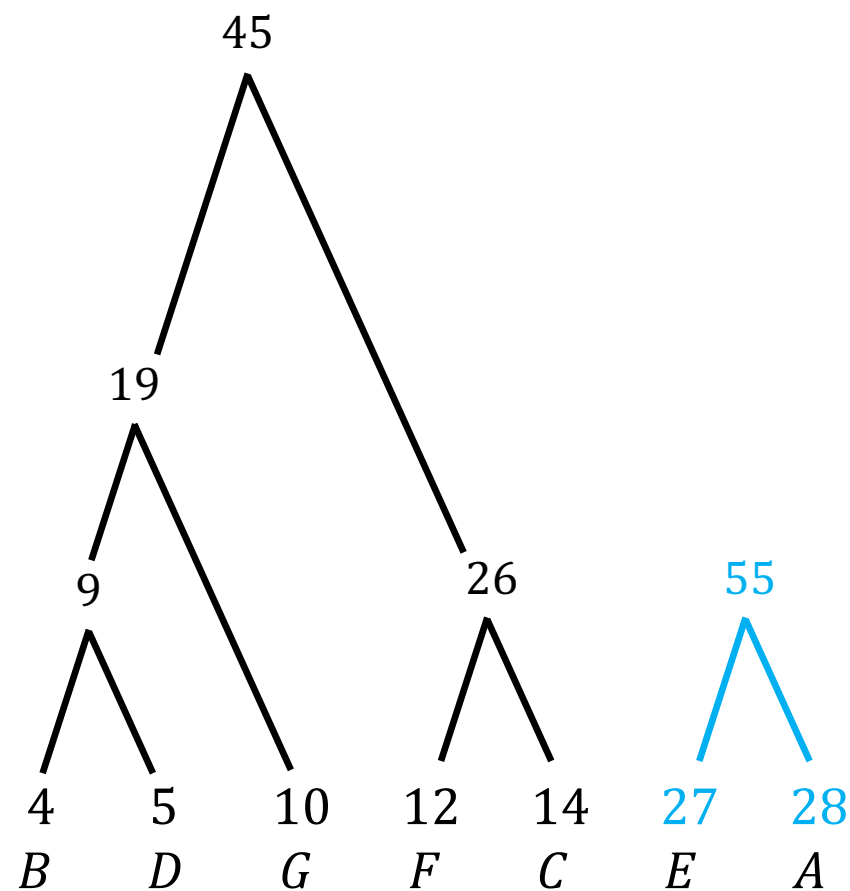
# Huffman Trees: Example



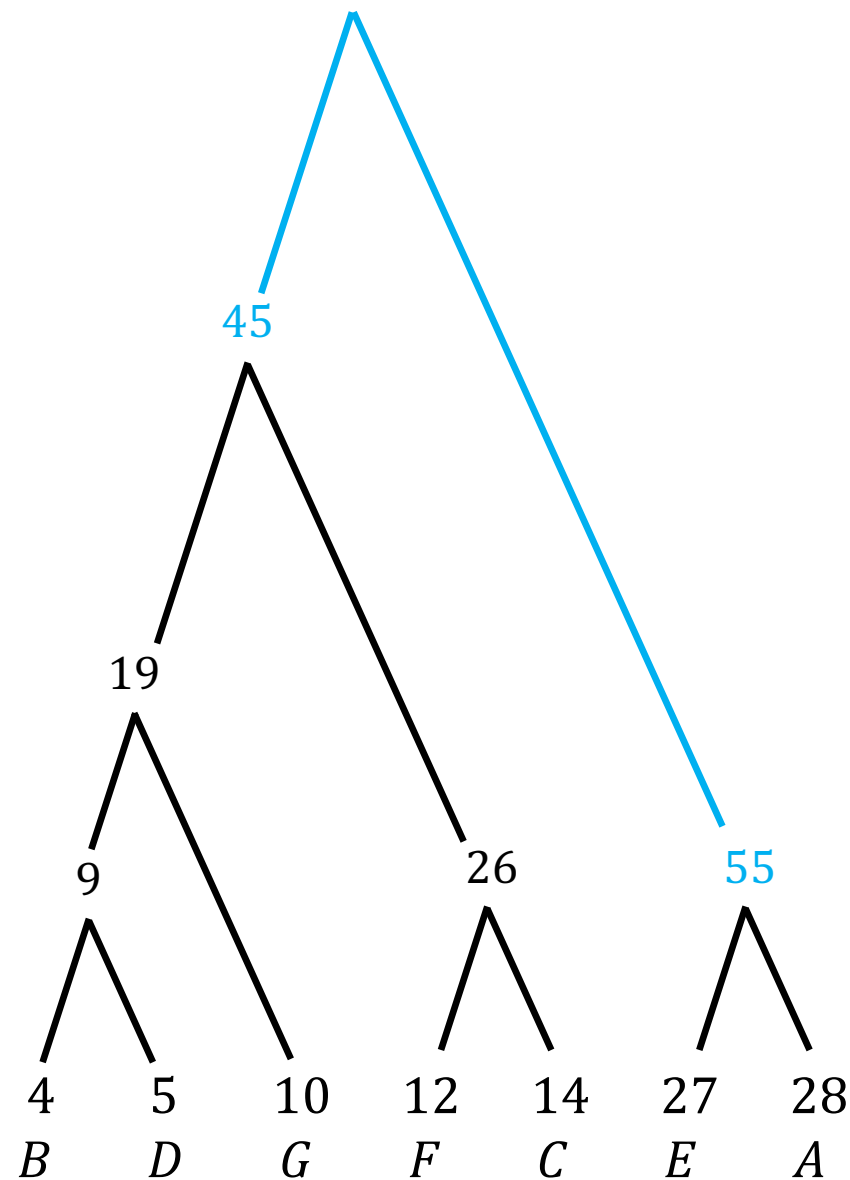
# Huffman Trees: Example



# Huffman Trees: Example

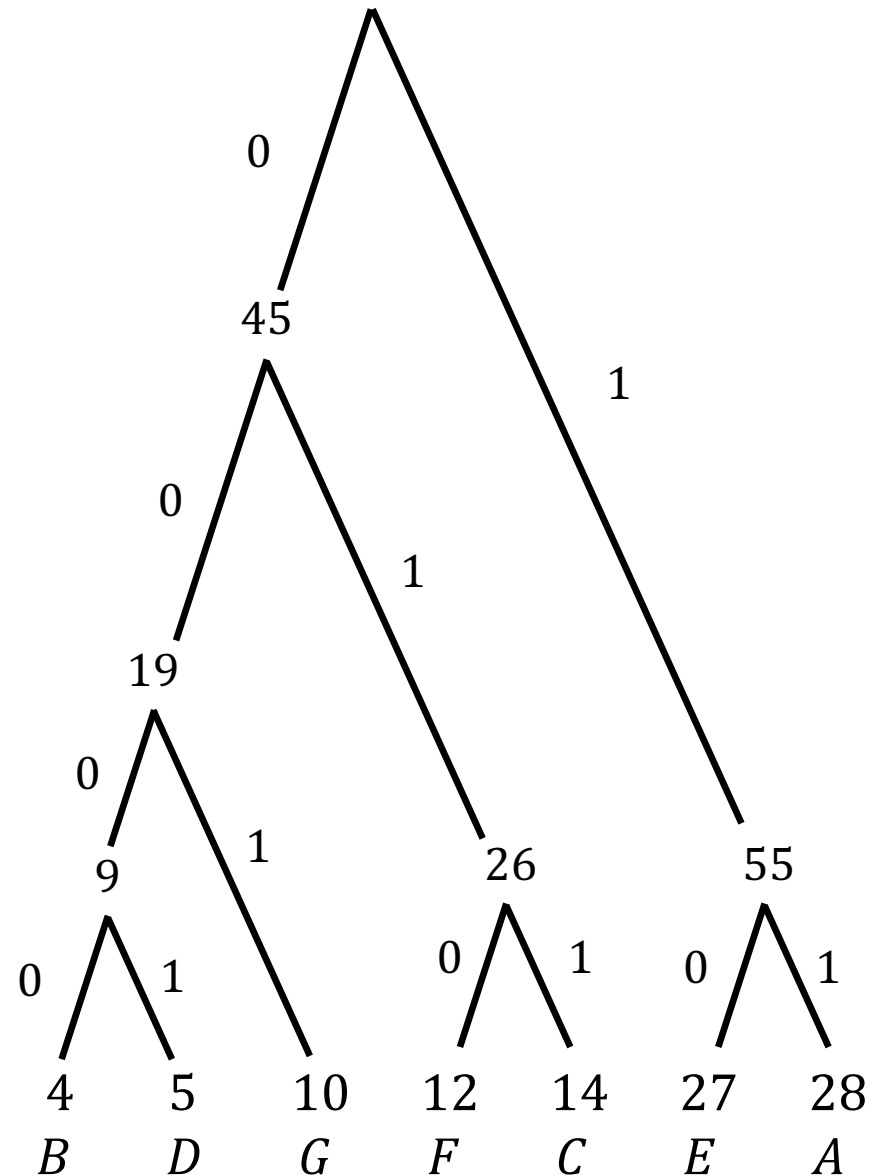


# Huffman Trees: Example



## Huffman Trees: Example

- We end up with the trie from before!
- One can show that this gives the **best** encoding.







# Huffman Trees: Example

|           |          |          |          |          |      |
|-----------|----------|----------|----------|----------|------|
| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | —    |
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



# Huffman Trees: Example

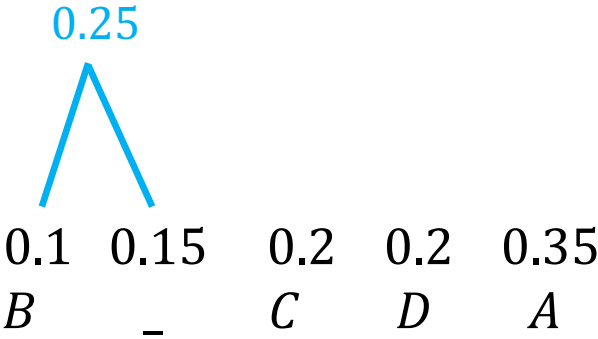
|           |          |          |          |          |      |
|-----------|----------|----------|----------|----------|------|
| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |

0.1   0.15   0.2   0.2   0.35  
*B*   –   *C*   *D*   *A*



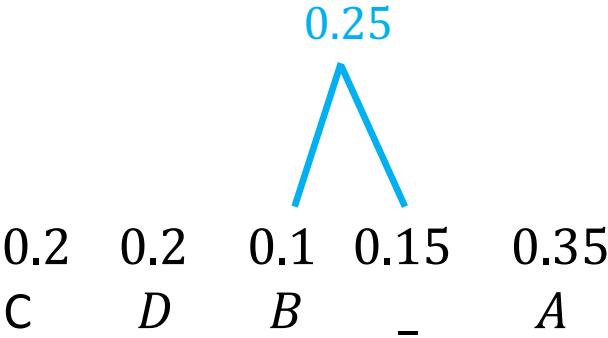
# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



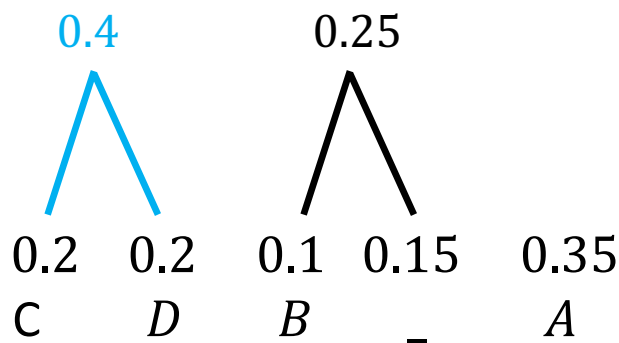
# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



# Huffman Trees: Example

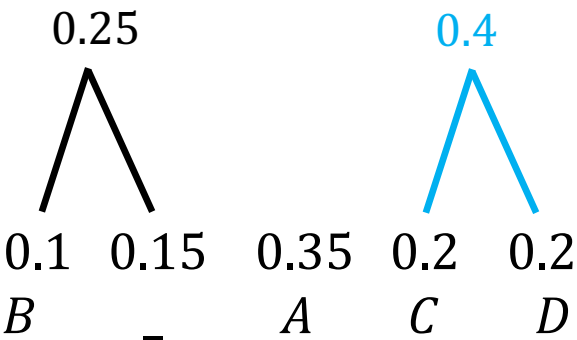
| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |





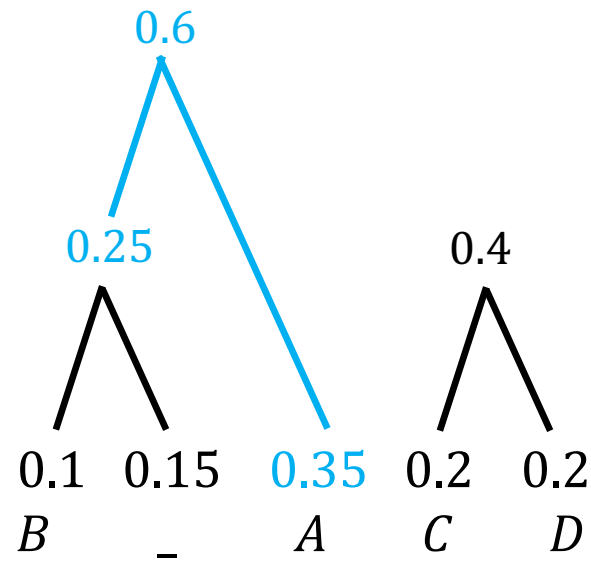
# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



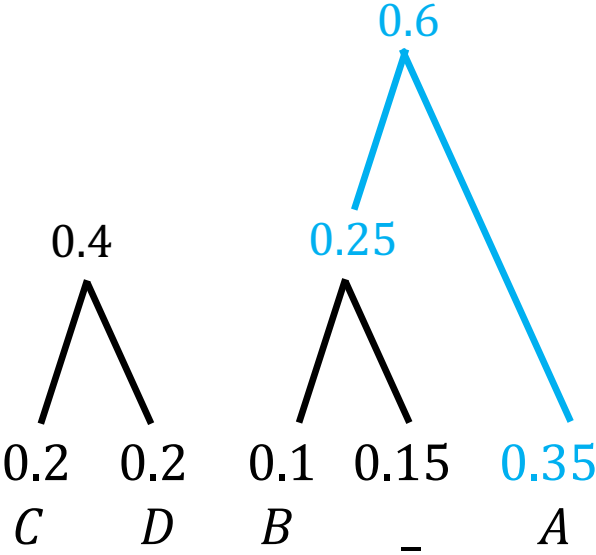
# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



# Huffman Trees: Example

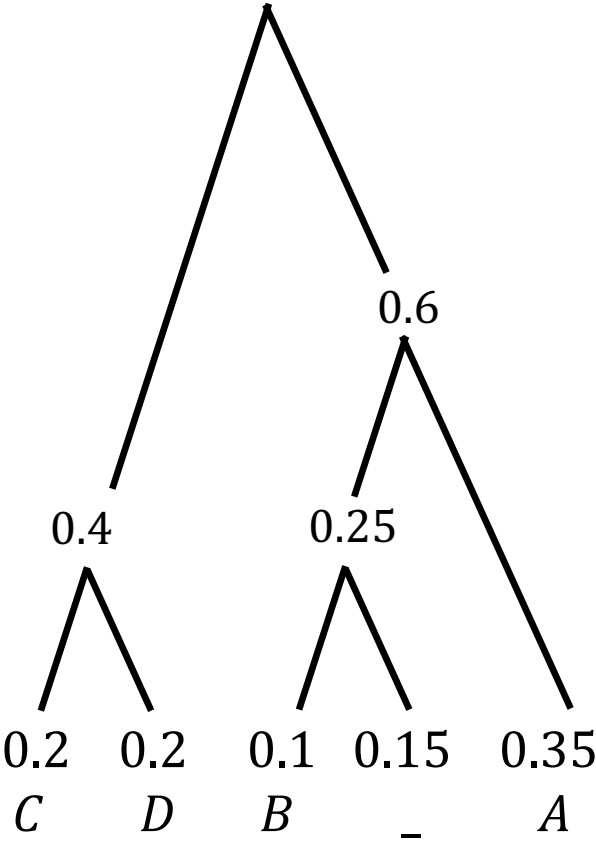
| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |





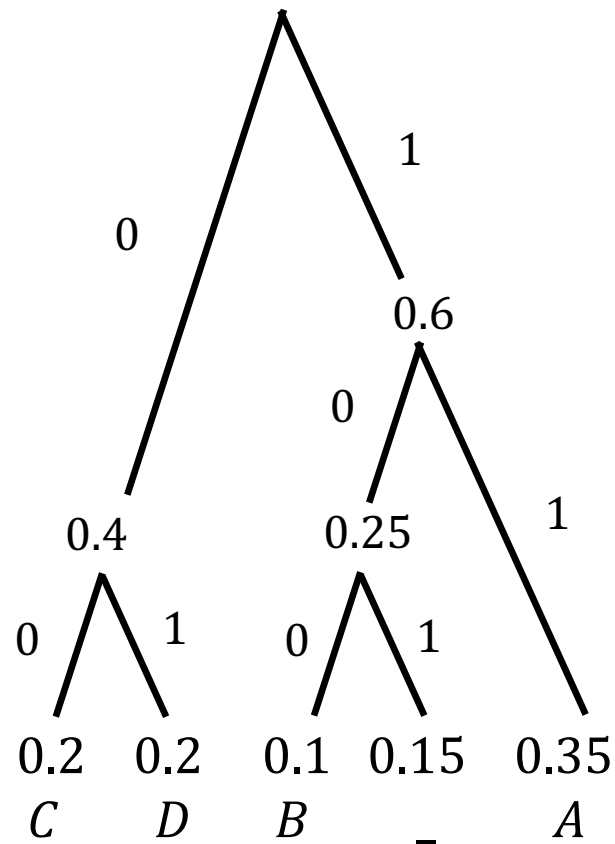
# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



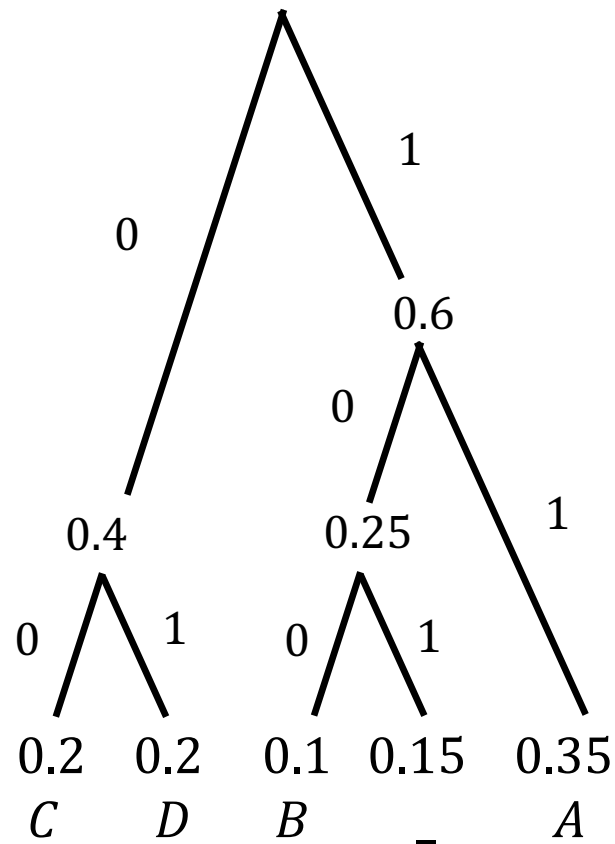
# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |



# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.35     | 0.1      | 0.2      | 0.2      | 0.15 |
| Codeword  | 11       | 100      | 00       | 01       | 101  |





# Huffman Trees: Example

|           |          |          |          |          |      |
|-----------|----------|----------|----------|----------|------|
| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | —    |
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

# Huffman Trees: Example

| Symbol    | $A$ | $B$ | $C$ | $D$  | —    |
|-----------|-----|-----|-----|------|------|
| Frequency | 0.4 | 0.1 | 0.2 | 0.15 | 0.15 |

Encode *ABACABAD*

Decode 100010111001010



# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

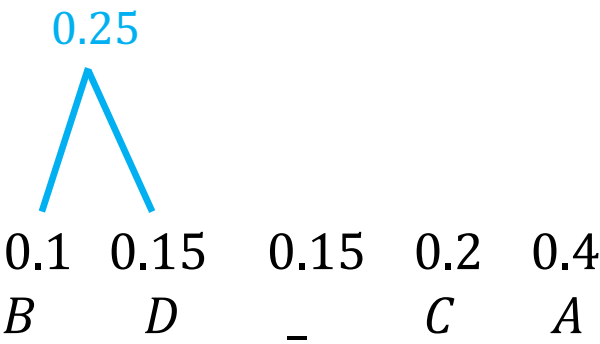
0.1   0.15   0.15   0.2   0.4  
*B*   *D*   –   *C*   *A*

# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010



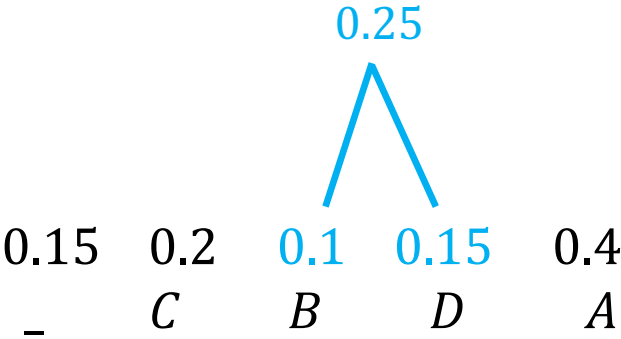


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010





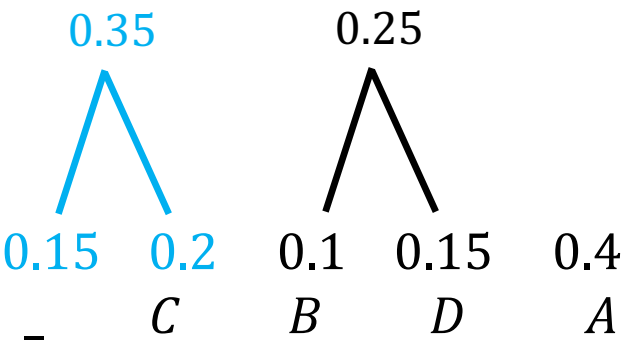


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

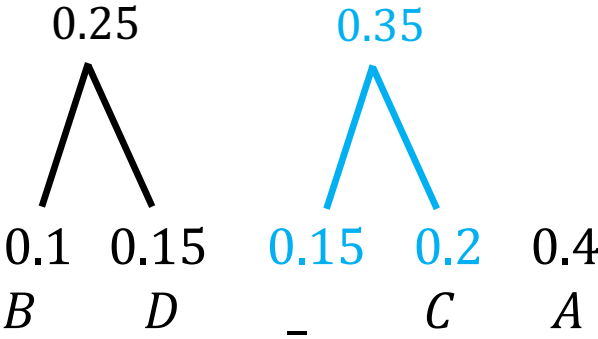


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | —    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

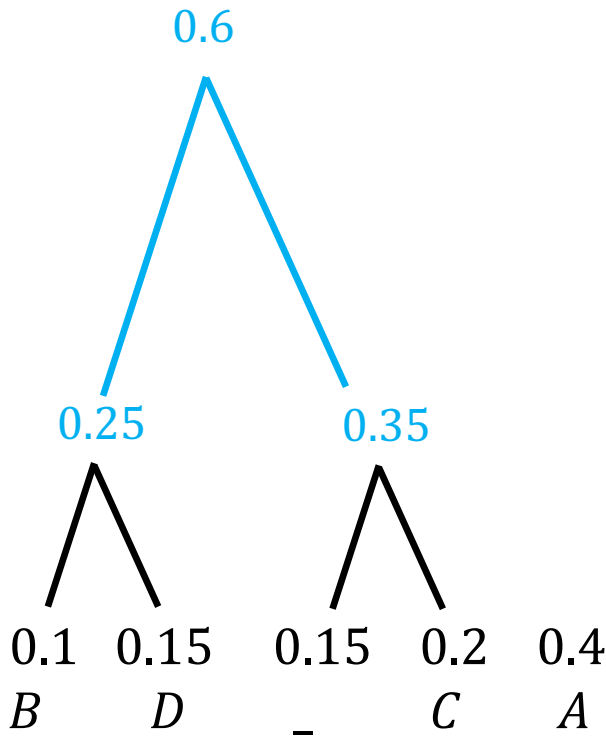


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

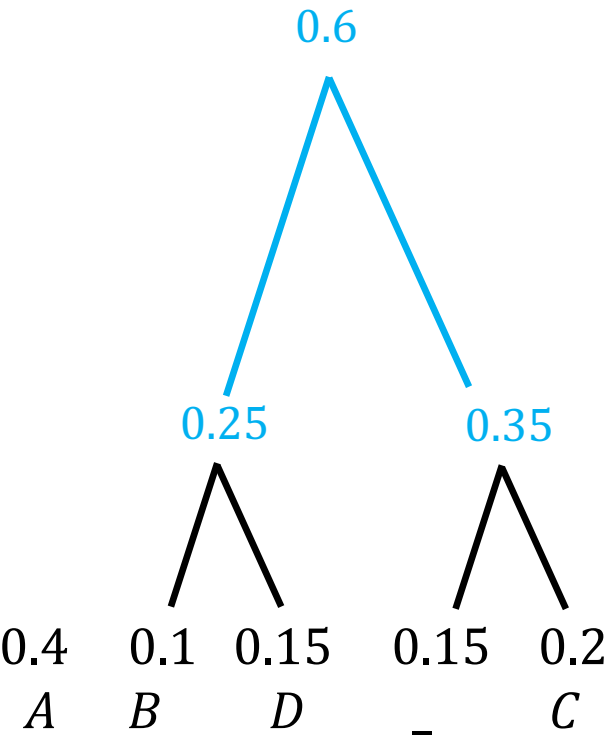


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

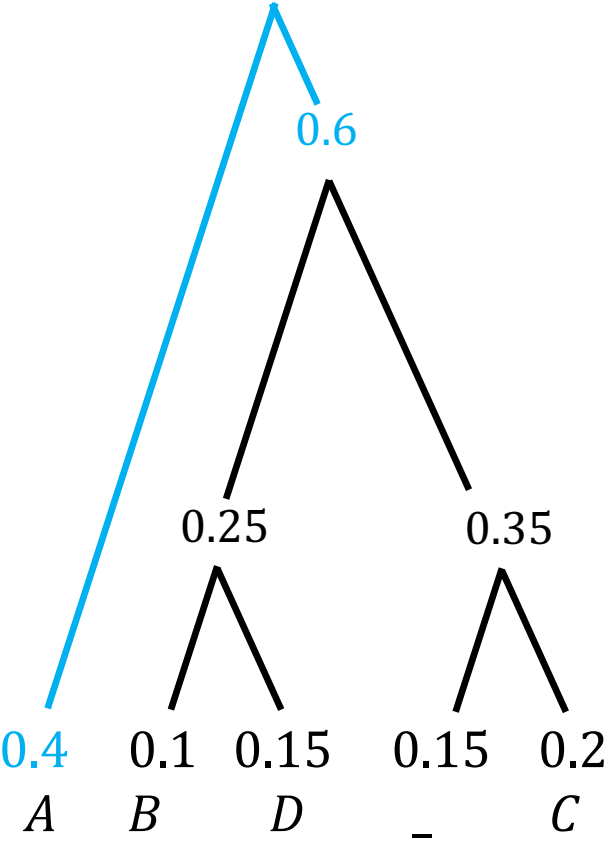


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

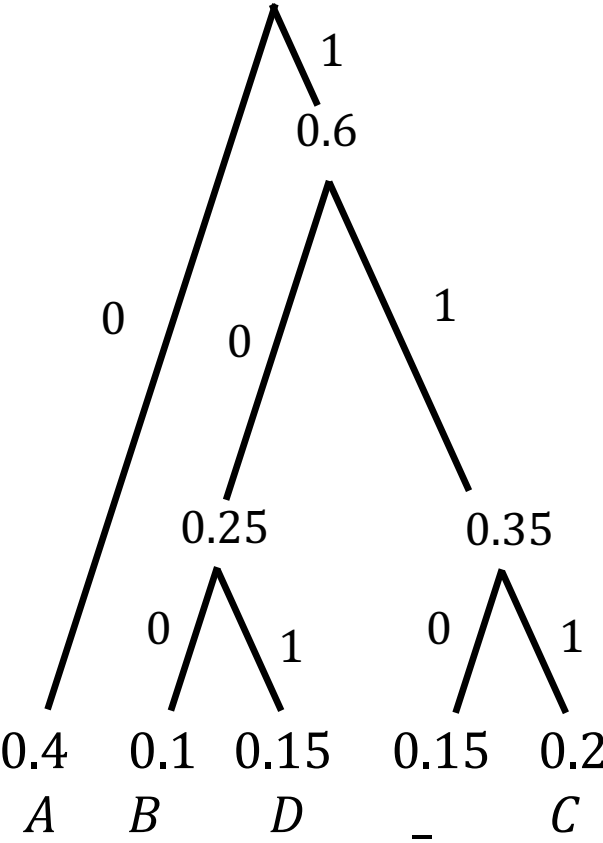


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |

Encode *ABACABAD*

Decode 100010111001010

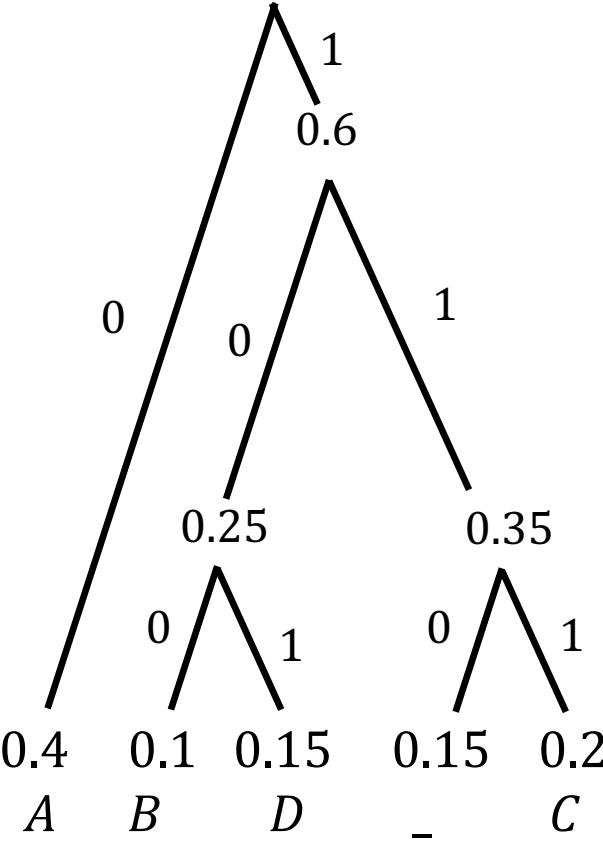


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |
| Codeword  | 0        | 100      | 111      | 101      | 110  |

Encode *ABACABAD*

Decode 100010111001010

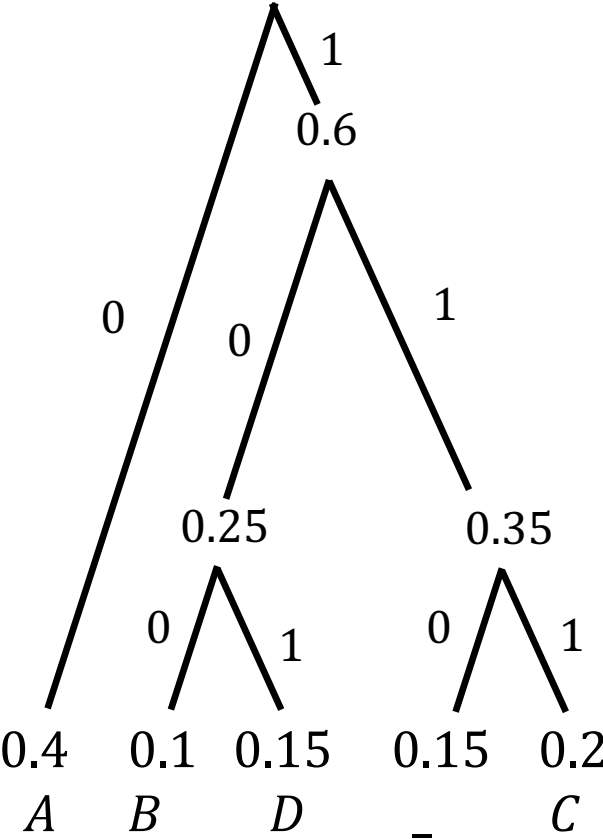


# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |
| Codeword  | 0        | 100      | 111      | 101      | 110  |

Encode *ABACABAD*:  
0100011101000101

Decode 100010111001010

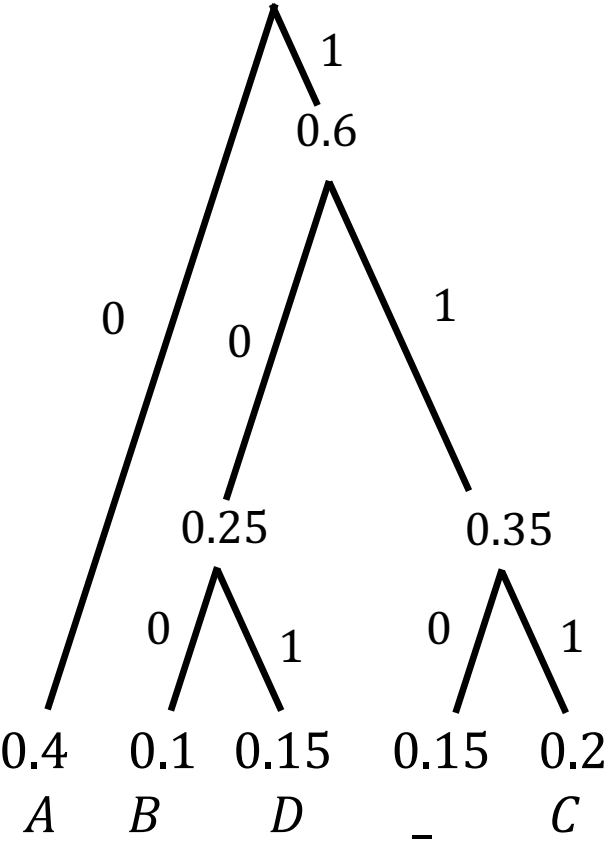




# Huffman Trees: Example

| Symbol    | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> | –    |
|-----------|----------|----------|----------|----------|------|
| Frequency | 0.4      | 0.1      | 0.2      | 0.15     | 0.15 |
| Codeword  | 0        | 100      | 111      | 101      | 110  |

Encode *ABACABAD*:  
0100011101000101



Decode 100010111001010:

|          |          |          |     |          |          |          |
|----------|----------|----------|-----|----------|----------|----------|
| 100      | 0        | 101      | 110 | 0        | 101      | 0        |
| <i>B</i> | <i>A</i> | <i>D</i> | –   | <i>A</i> | <i>D</i> | <i>A</i> |

# Compressed Transmission

- If the compressed file is being sent from one party to another, the parties must agree about the codes used.
- Alternatively, the trie can be sent along with the message.
- For long files this extra cost is negligible.
- Modern variant of Huffman encoding, like **Lempel-Ziv compression**, assign codes not to individual symbols, but to sequences of symbols.

# Coming Up Next



- NP-completeness.