COMP90038
Algorithms and Complexity

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

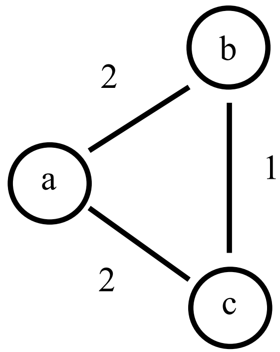
Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

Review from Lecture 20: Greedy Algorithms

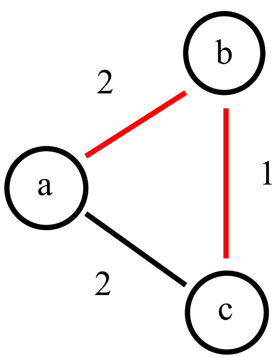
- In general we cannot expect locally best choices to yield globally best outcomes.
- However, there are some well-known algorithms that rely on the greedy approach, being both correct and fast.
- In other cases, for hard problems, a greedy algorithm can sometimes serve as an acceptable approximation 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.

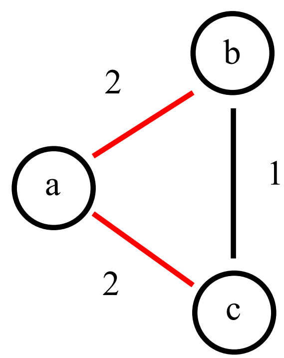


Prim's Algorithm



Finding the minimum spanning tree

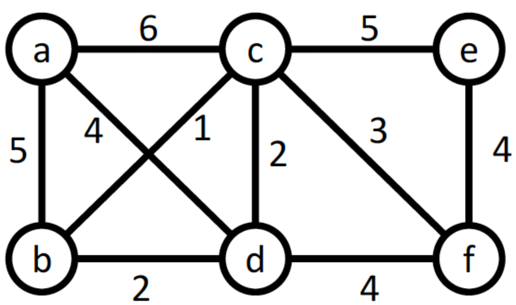
Dijkstra's Algorithm



Finding the single-source shortest paths

Review from Lecture 20: Prim's Algorithm

THE UNIVERSITY OF MELBOURNE

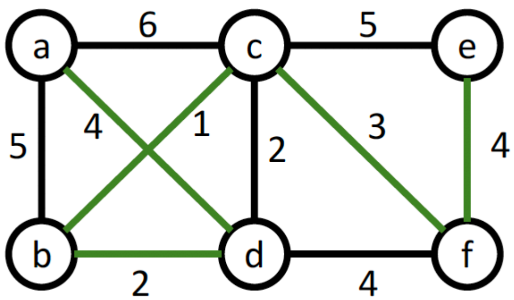


- On the first loop, we only create the table.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil

```
function PRIM(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v0
    cost[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)          ▷ priorities are cost values
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, cost[w])   ▷ rearranges priority queue
```

Review from Lecture 20: Prim's Algorithm

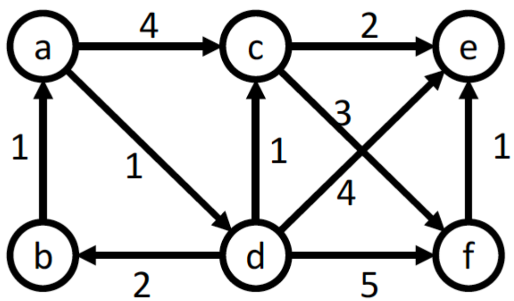



```
function Prim(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v_0
    cost[v_0] ← 0
    Q ← InitPriorityQueue(V)        ▷ priorities are cost values
    while Q is non-empty do
        u ← EjectMin(Q)
        for each (u, w) ∈ E do
            if w in Q and weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                Update(Q, w, cost[w])    ▷ rearranges priority queue
```

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

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

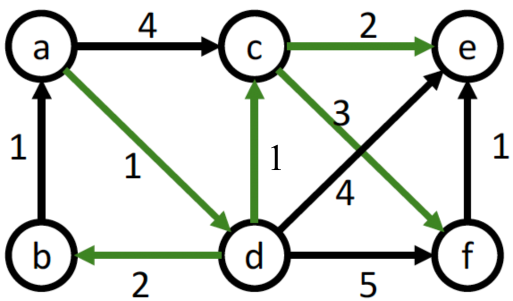


- On the first loop, we only create the table.

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil

```
function DIJKSTRA(⟨V, E⟩, v0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v0] ← 0
    Q ← INITPRIORITYQUEUE(V)            ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if w in Q and dist[u] + weight(u, w) < dist[w] then
                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, dist[w])   ▷ rearranges priority queue
```

Review from Lecture 20: Dijkstra's Algorithm



function DIJKSTRA($\langle V, E\rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ 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($Q, w, dist[w]$) ▷ rearranges priority queue

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

Tree T		a	b	c	d	e	f
	$cost$	∞	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
	$cost$	0	∞	∞	∞	∞	∞
	$prev$	nil	nil	nil	nil	nil	nil
a	$cost$		∞	4	1	∞	∞
	$prev$		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:

$$AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD$$

can then be encoded as

$$4A3BAA5B8CDABCB3A4B3CD$$

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

Run-Length Encoding

```
0000000000000000000000000000011111111111111000000000
0000000000000000000000000001111111111111111110000000
0000000000000000000000001111111111111111111111110000
0000000000000000000000011111111111111111111111111000
0000000000000000000001111111111111111111111111111110
0000000000000000000011111100000000000000000001111111
0000000000000000000011110000000000000000000000011111
0000000000000000000011100000000000000000000000000111
0000000000000000000011100000000000000000000000000111
0000000000000000000011100000000000000000000000000111
0000000000000000000011100000000000000000000000000111
0000000000000000000001110000000000000000000000001110
0000000000000000000000011100000000000000000000111000
0111111111111111111111111111111111111111111111111111
0111111111111111111111111111111111111111111111111111
0111111111111111111111111111111111111111111111111111
0111111111111111111111111111111111111111111111111111
0110000000000000000000000000000000000000000000000011
```

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:

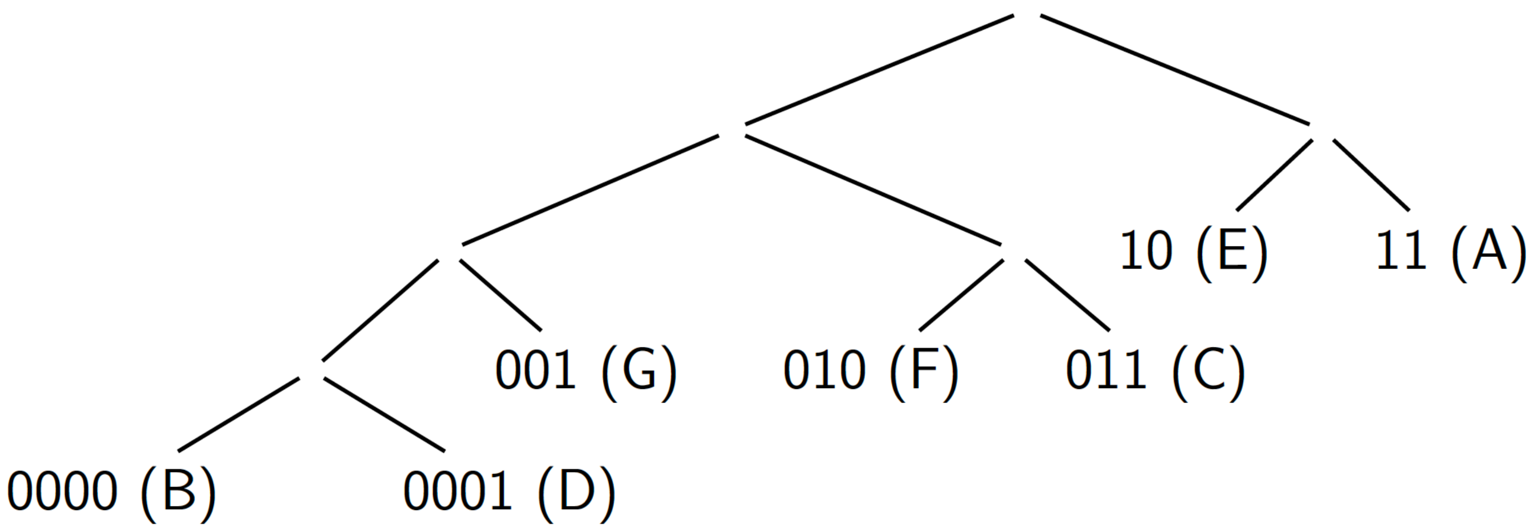
Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

- Here are some sensible codes that we may use for symbols:

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

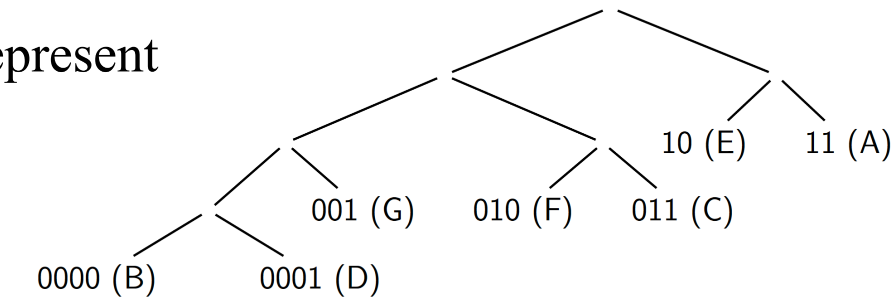
Tries for Variable-Length Encoding

- A trie is a binary tree for search applications.
- To search for a key we look at individual bits of a key and descend to the left whenever a bit is 0, to the right whenever it is 1.
- Using a trie to determine codes means that no code will be the prefix of another!



Variable-Length Encoding

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



$$FACE$$

in just 10 bits:

$$\underbrace{010}_{F}\ \underbrace{11}_{A}\ \underbrace{011}_{C}\ \underbrace{10}_{E}$$

- 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 ‘$BAGGED$’ simple 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

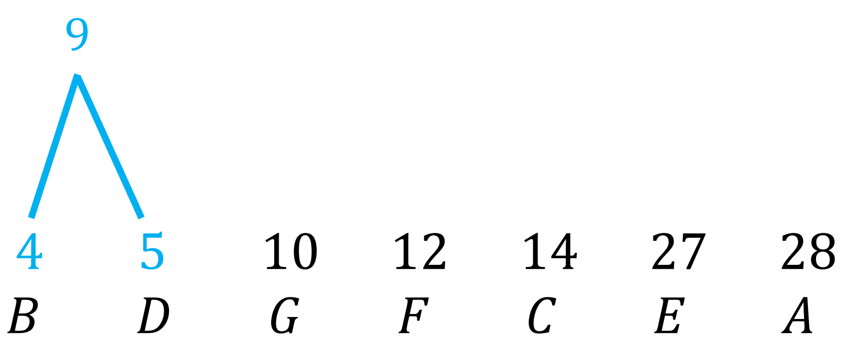
Huffman Encoding: Choosing the Codes

- 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.

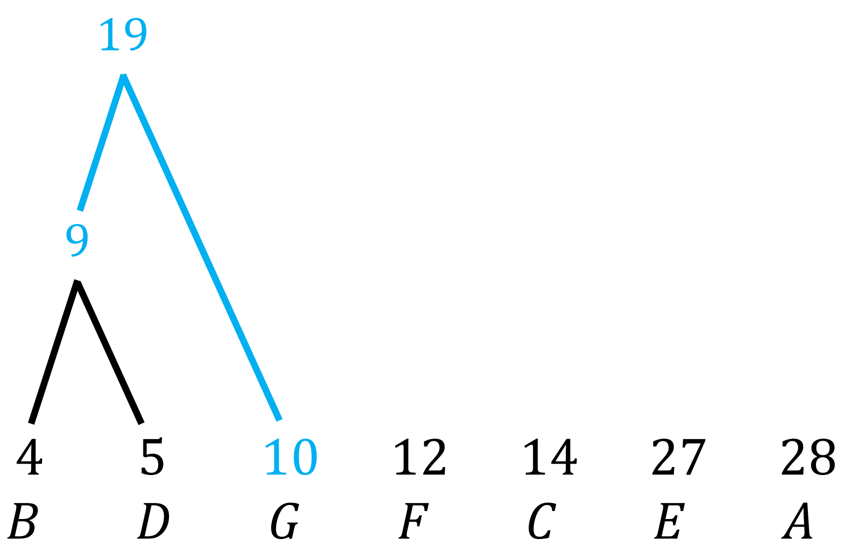
Huffman Trees: Example

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

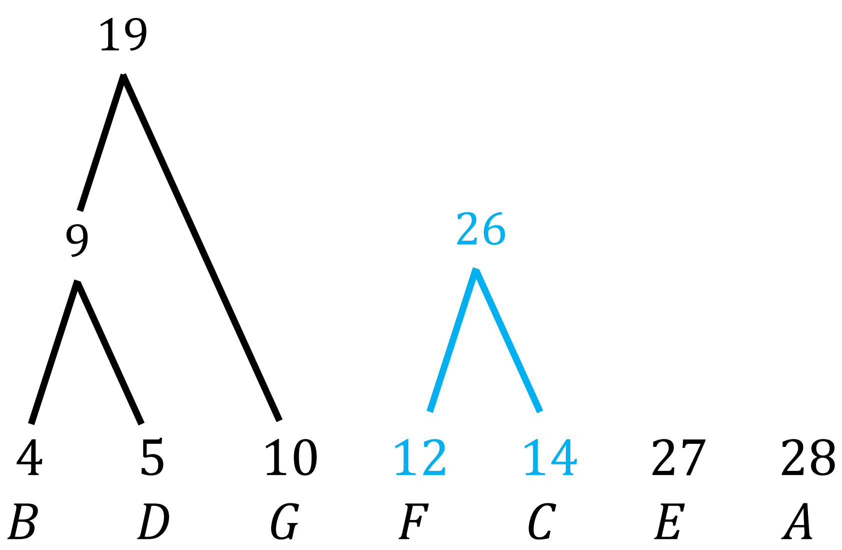
Huffman Trees: Example



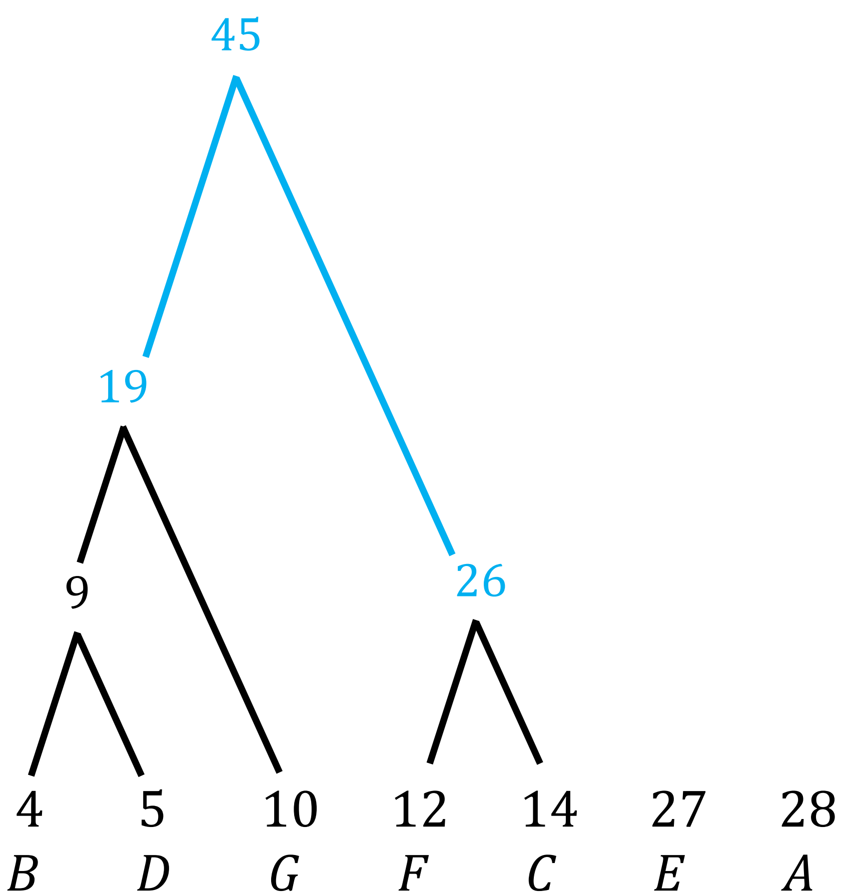
Huffman Trees: Example



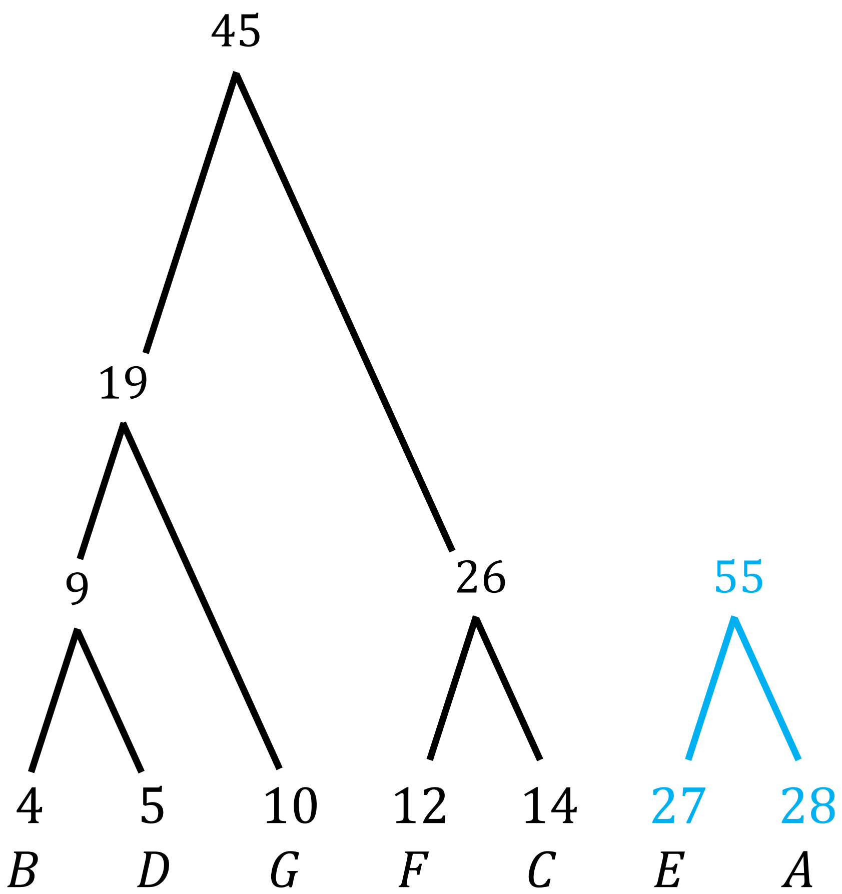
Huffman Trees: Example



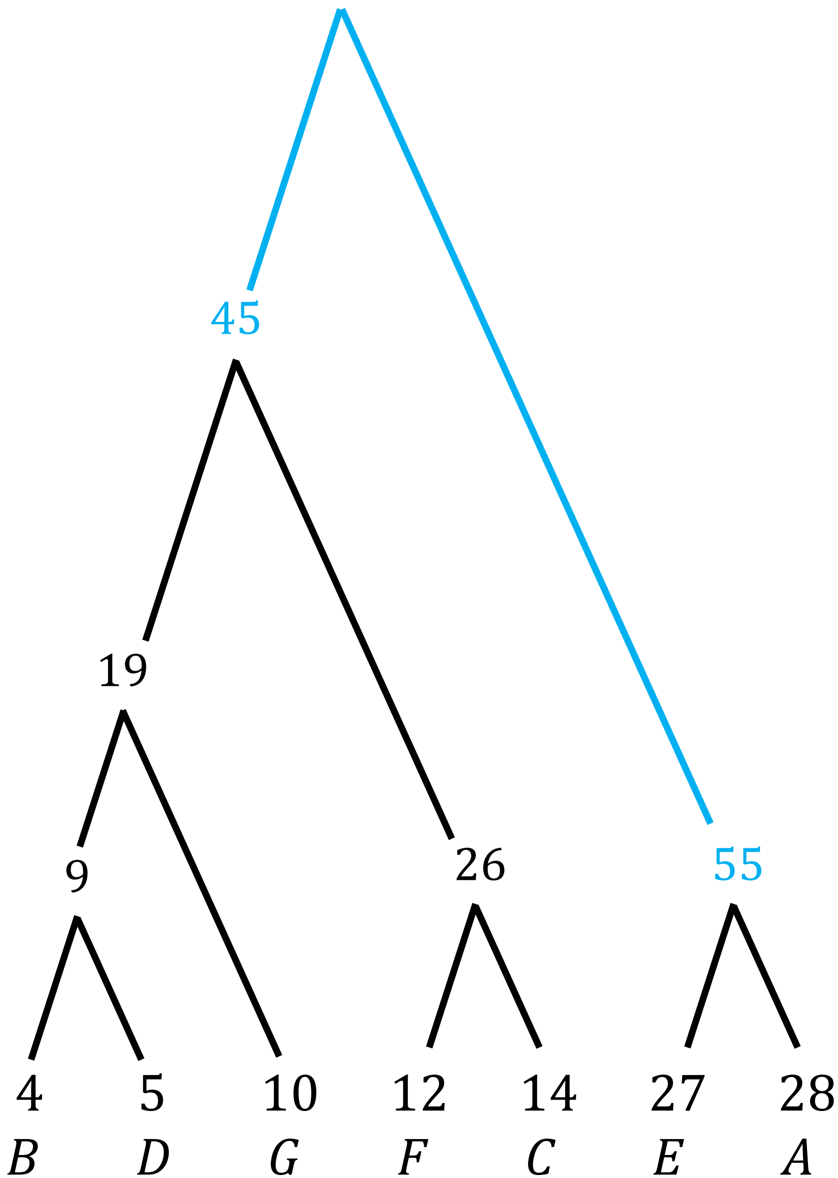
Huffman Trees: Example



Huffman Trees: Example

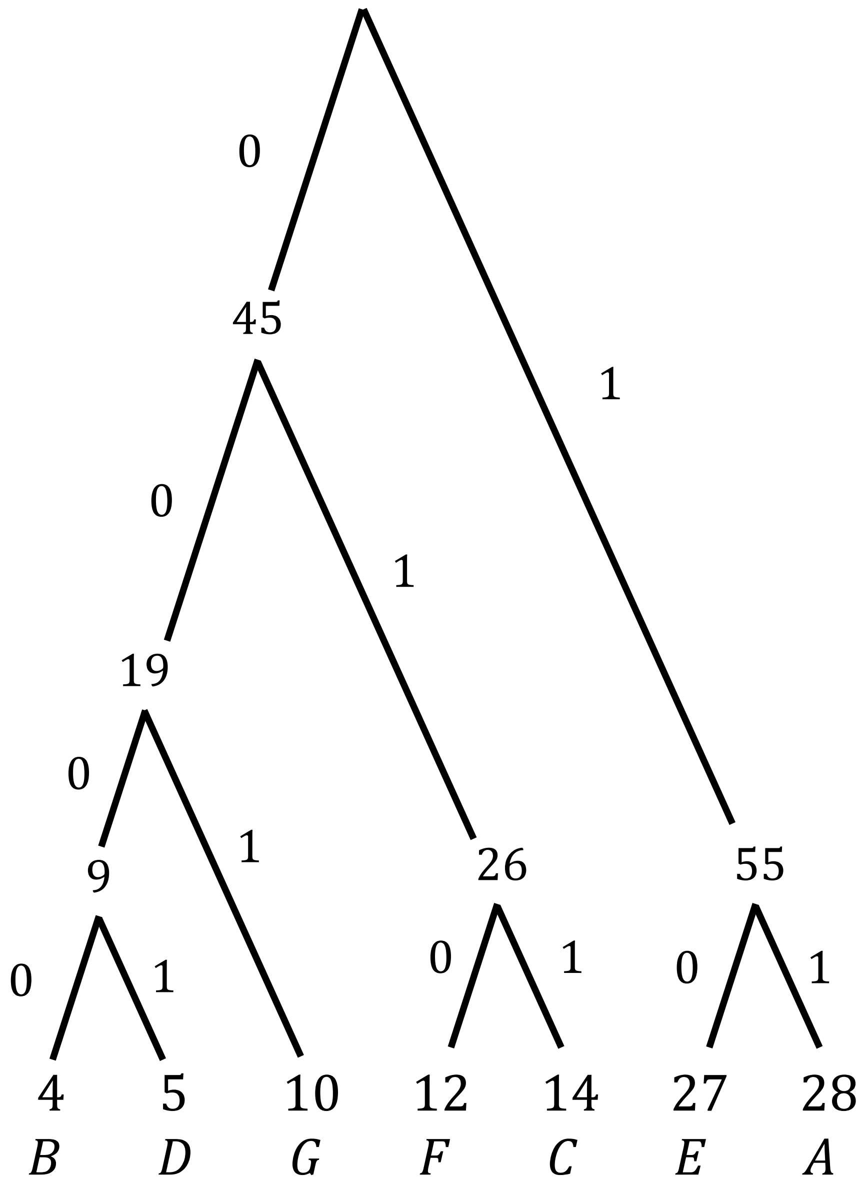


Huffman Trees: Example



Huffman Trees: Example

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



Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15

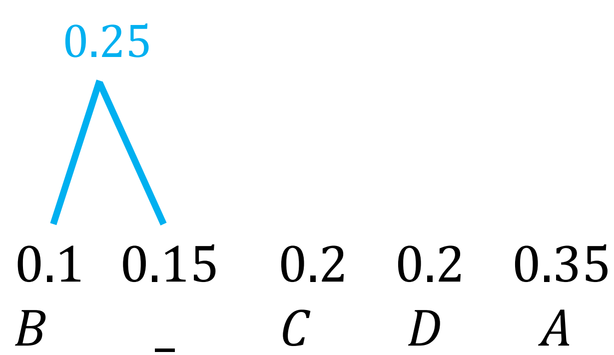
Huffman Trees: Example

Symbol	A	B	C	D	$\_$
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

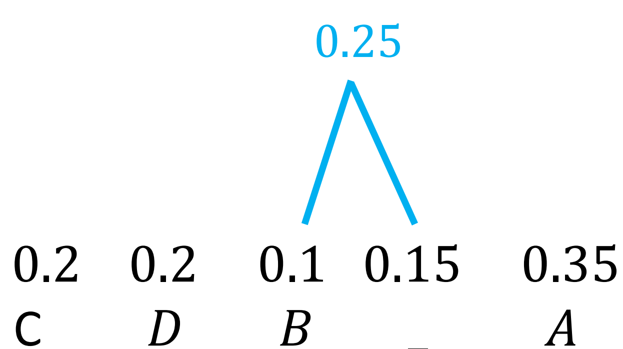
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



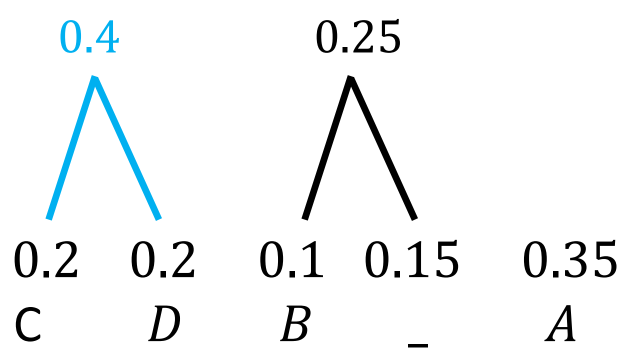
Huffman Trees: Example

Symbol	*A*	*B*	*C*	*D*	_
Frequency	0.35	0.1	0.2	0.2	0.15



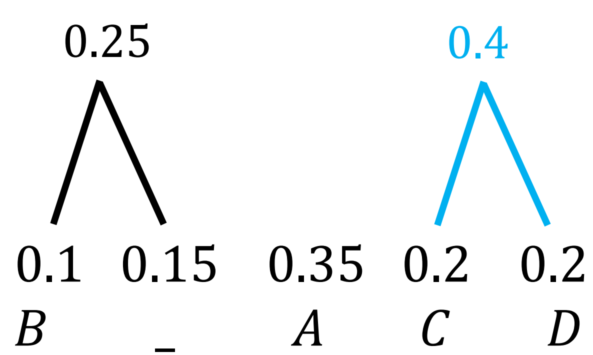
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



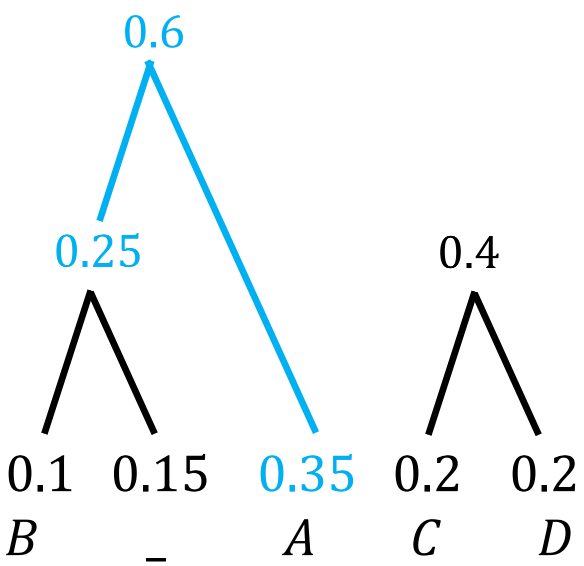
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



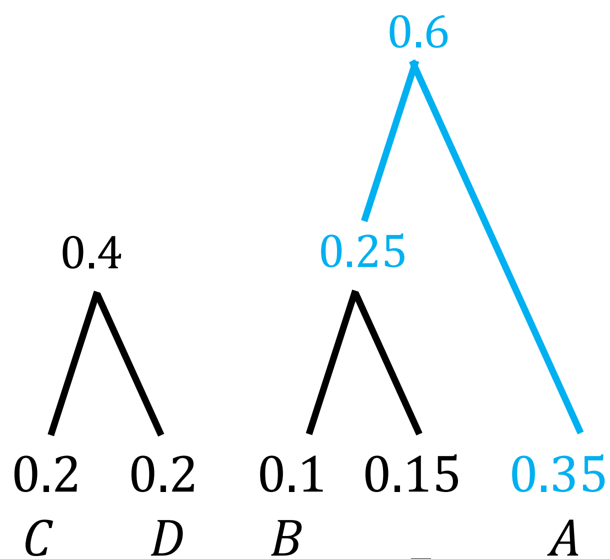
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



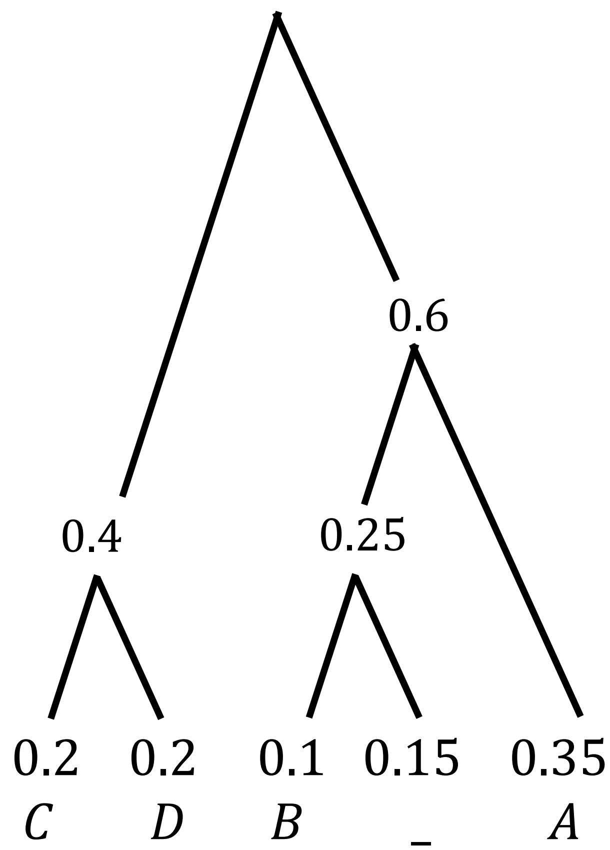
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



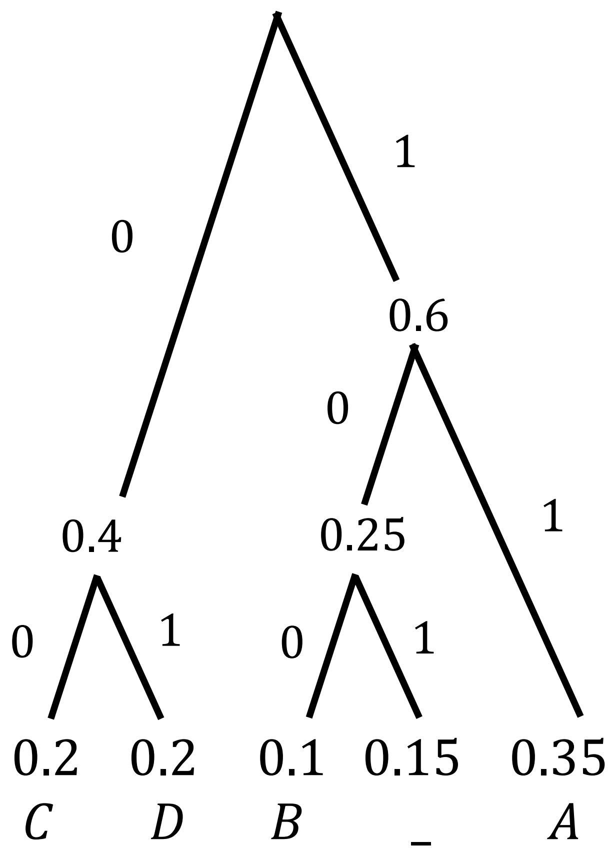
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



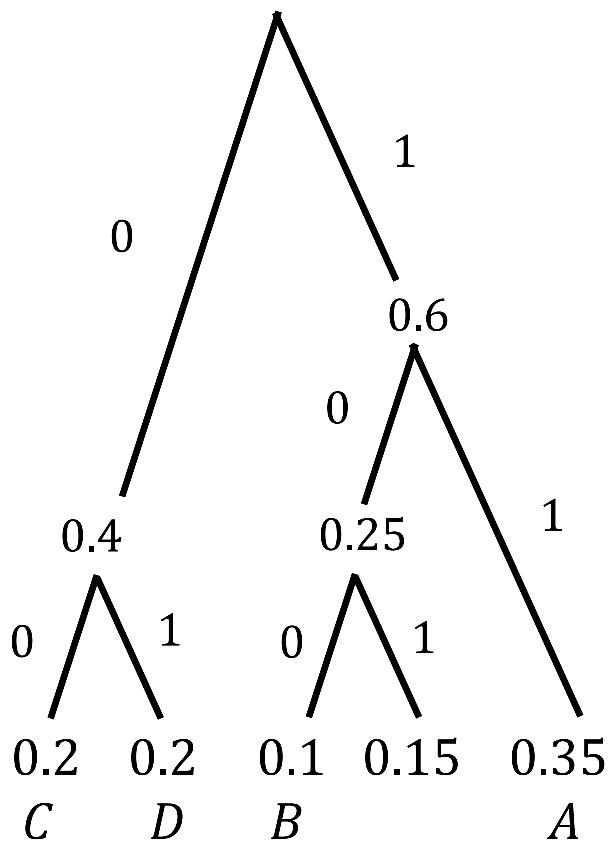
Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15



Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101



Huffman Trees: Example

Symbol	A	B	C	D	_
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	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

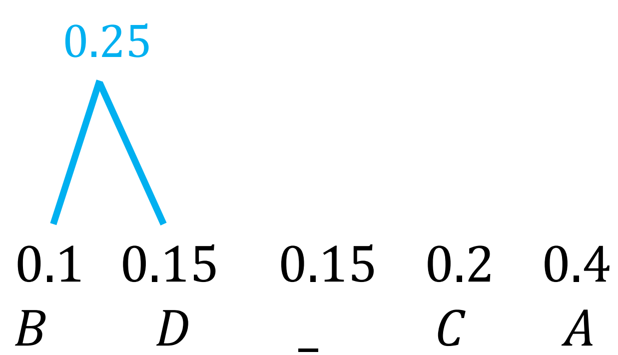
0.1	0.15	0.15	0.2	0.4
B	D	_	C	A

Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

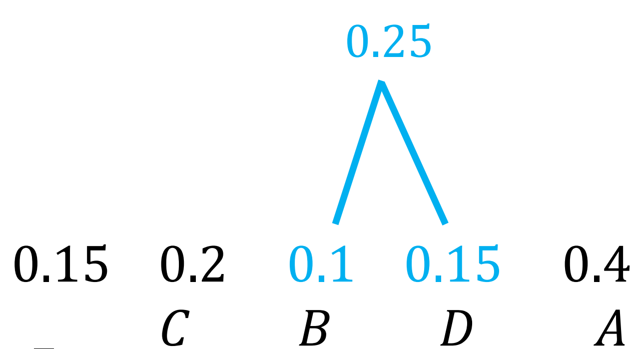


Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

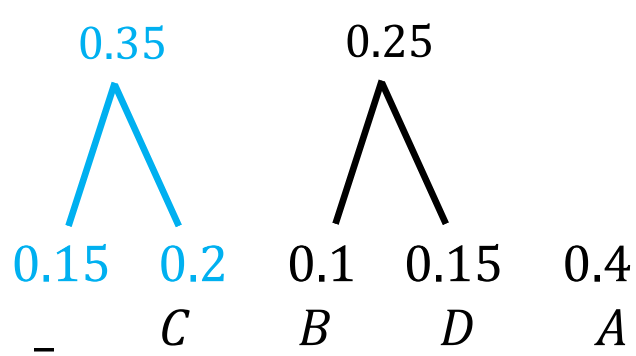


Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

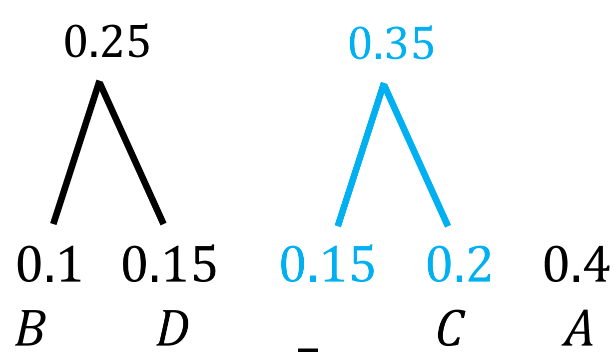


Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

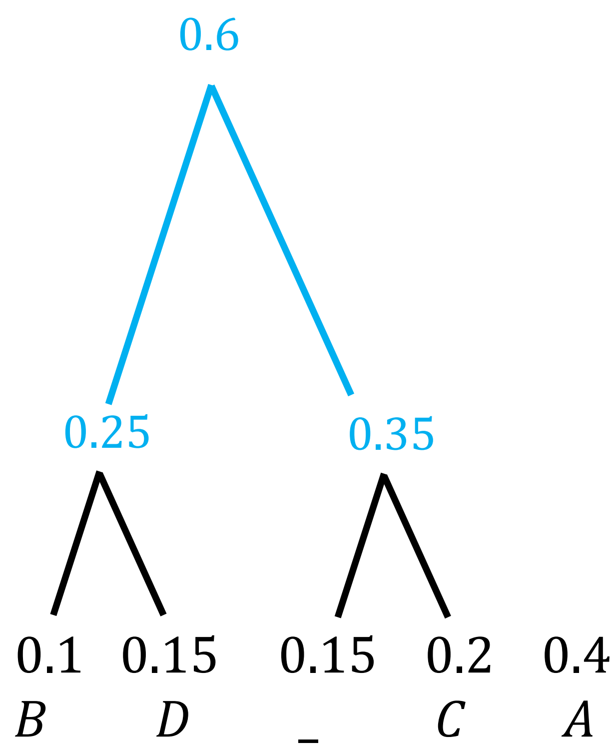


Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

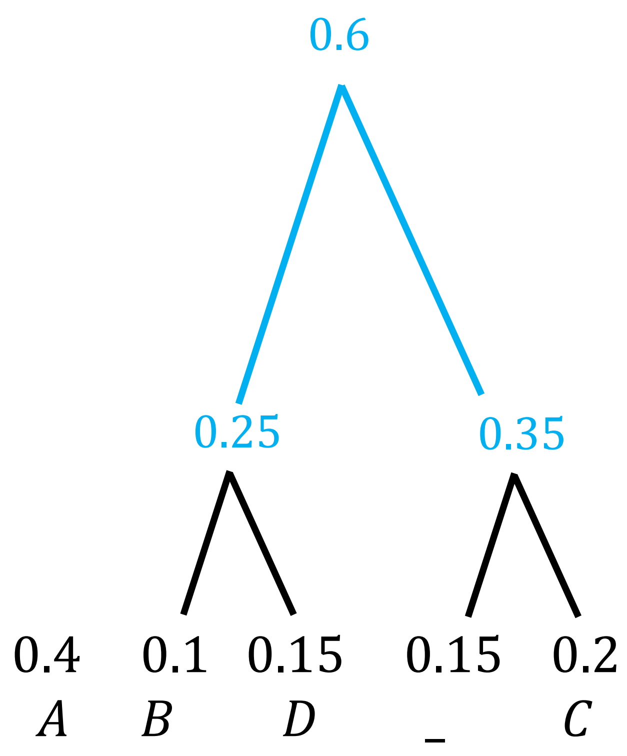


Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

Decode 100010111001010

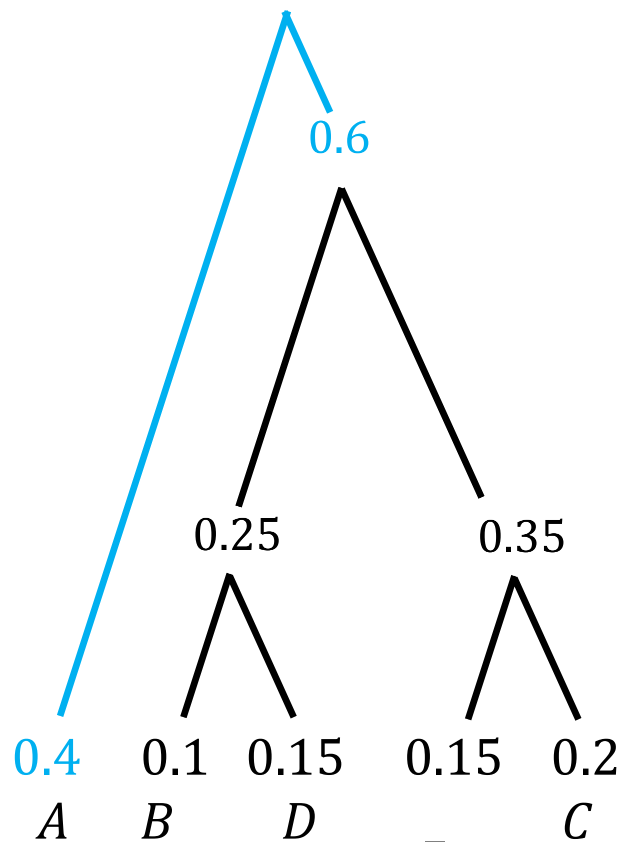


Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

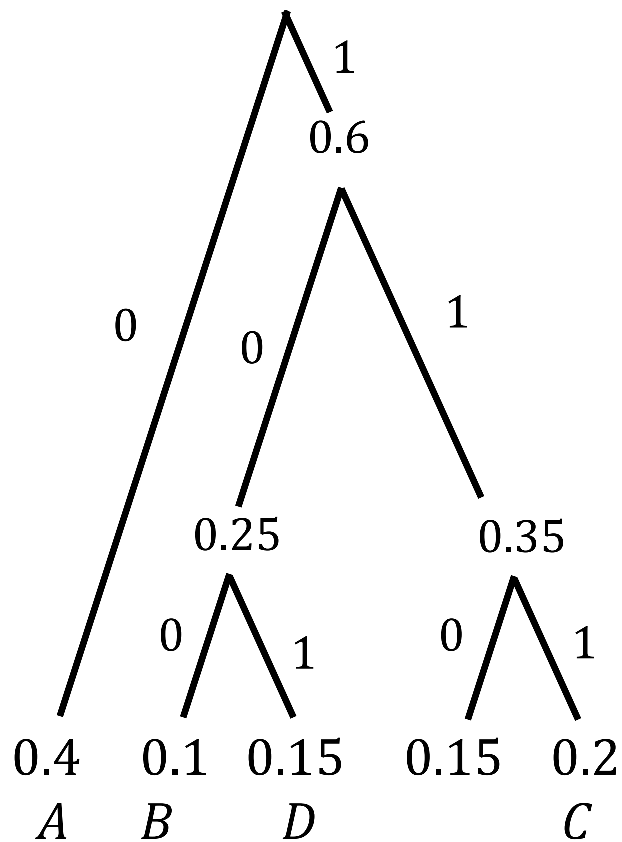
Decode 100010111001010



Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15

Encode $ABACABAD$

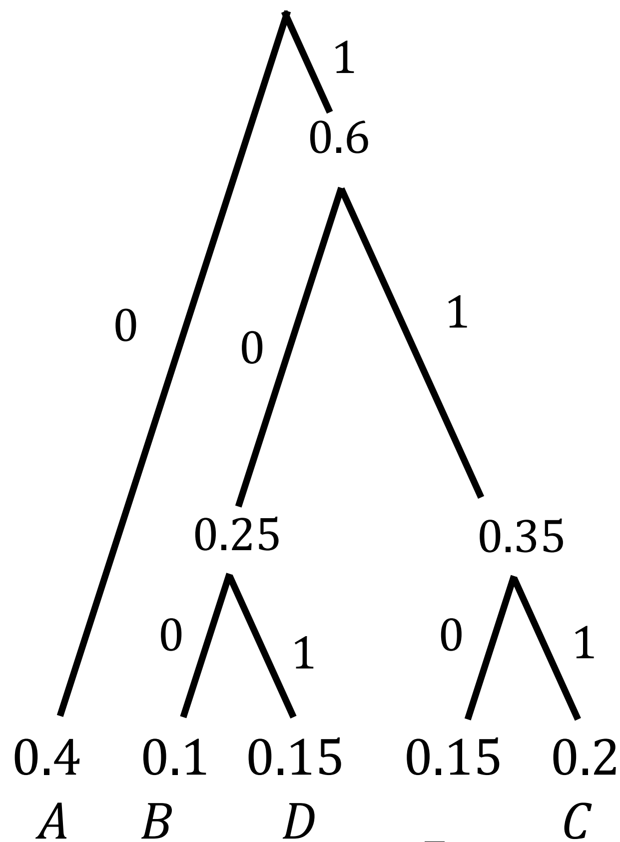


Decode 100010111001010

Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15
Codeword	0	100	111	101	110

Encode $ABACABAD$

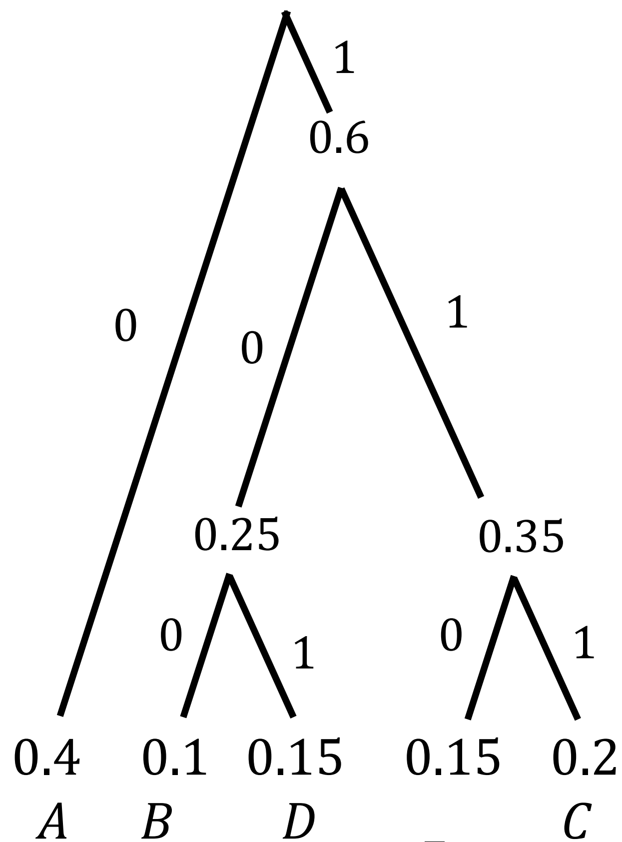


Decode 100010111001010

Huffman Trees: Example

Symbol	A	B	C	D	_
Frequency	0.4	0.1	0.2	0.15	0.15
Codeword	0	100	111	101	110

Encode $ABACABAD$:
0100011101000101

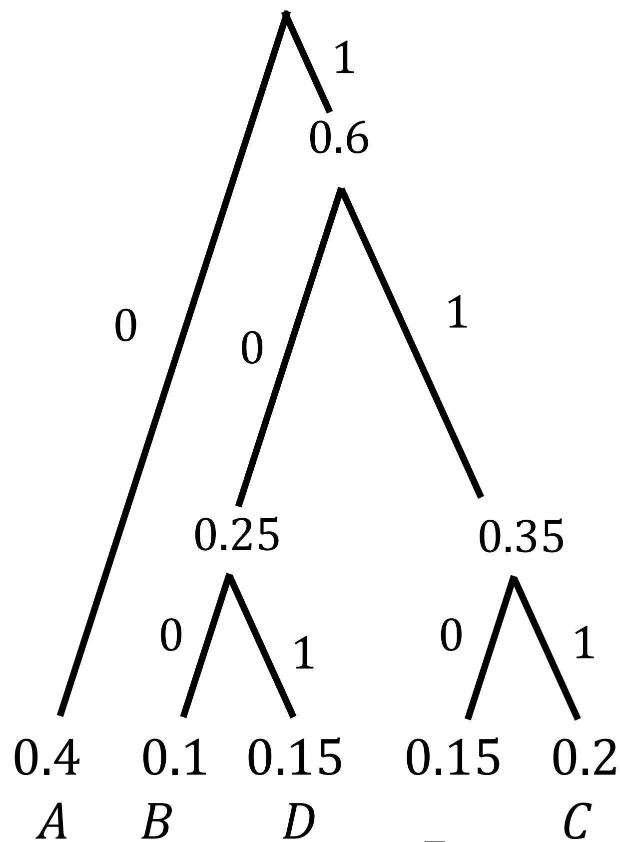


Decode 100010111001010

Huffman Trees: Example

Symbol	A	B	C	D	_
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
B	A	D	_	A	D	A

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.