Chapter 3: Compressing

Panos Louridas

Athens University of Economics and Business

Real World Algorithms
A Beginners Guide
The MIT Press

Outline

1 Encodings and Number Systems

2 Compression

3 Priority Queues

4 Huffman Coding

5 Lempel-Ziv-Welch Compression

The ASCII Code

- The most common way to represent latin characters on a computer is the ASCII encoding (American Standard Code for Information Interchange).
- ASCII uses 7 bits to represent each character.
- ASCII can represent $2^7 = 128$ characters in total.

Number of Characters Represented with 7 bits

$$\boxed{0/1}\quad\boxed{0/1}\quad\boxed{0/1}\quad\boxed{0/1}\quad\boxed{0/1}\quad\boxed{0/1}\quad\boxed{0/1}$$

$$\underbrace{2 \quad \underbrace{2 \quad \underbrace{2 \quad \underbrace{2 \quad \underbrace{2 \quad \underbrace{2 \quad 2}_{2 \times 2}}_{2 \times 2 \times 2}}_{2 \times 2 \times 2 \times 2}}_{2 \times 2 \times 2 \times 2 \times 2}}_{2 \times 2 \times 2 \times 2 \times 2 \times 2}}_{2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^7}$$

ASCII Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Decimal Number System

- In the decimal number system, number 53 has as its value 53 because $5 \times 10 + 3$.
- The general rule is that a number written as:

$$D_n D_{n-1} \dots D_1 D_0$$

has as its value:

$$D_n \times 10^n + D_{n-1} \times 10^{n-1} + \cdots + D_1 \times 10^1 + D_0 \times 10^0$$

.

Hexadecimal Number System

- In the *hexadecimal* number system (or hex, for short), the logic is the same, but we use as *base* the number 16.
- The number:

  $$H_n H_{n-1} \ldots H_1 H_0$$

  in hexadecimal has as its value:

  $$H_n \times 16^n + H_{n-1} \times 16^{n-1} + \cdots + D_1 \times 16^1 + D_0 \times 16^0$$

  For example, the value of 20 in hexadecimal is $2 \times 16 + 0 = 32$ and number 1B in hexadecimal is equal to $1 \times 16 + 11 = 27$.

Hexadecimal Numbers

- We usually write 0x before a number to show that it is written in hex.
- So we write 0x20 to indicate that the number's value is 32 and not 20.
- We also write 0x1B, even though the number 1B cannot be a decimal number, in order to be consistent.

Binary Number System

- The *binary* number system uses 2 as its base, so that a number written as:

$$B_n B_{n-1} \ldots B_1 B_0$$

has as its value:

$$B_n \times 2^n + B_{n-1} \times 2^{n-1} + \cdots + B_1 \times 2^1 + B_0 \times 2^0$$

.

Other Number Systems

- We can have other number systems with different bases b.
- The rule in *positional number systems* is that the number:

$$X_n X_{n-1} \ldots X_1 X_0$$

in a number system with base b has as its value:

$$X_n \times b^n + X_{n-1} \times b^{n-1} + \cdots + X_1 \times b^1 + X_0 \times b^0$$

.

ASCII Encoding Example

Say you have the sentence “I am seated in an office”.

I		a	m		s	e	a
0x49	0x20	0x61	0x6D	0x20	0x73	0x65	0x61
1001001	100000	1100001	1101101	100000	1110011	1100101	1100001
t	e	d		i	n		a
0x74	0x65	0x64	0x20	0x69	0x6E	0x20	0x61
1110100	1100101	1100100	100000	1101001	1101110	100000	1100001
n		o	f	f	i	c	e
0x6E	0x20	0x6F	0x66	0x66	0x69	0x63	0x65
1101110	100000	1101111	1100110	1100110	1101001	1100011	1100101

ASCII Encoding Example (Contd.)

- Say you have the sentence “I am seated in an office”.
- In ASCII this is:
  49 20 61 6D 20 73 65 61 74 65 64 20 69 6E 20 61 6E 20 6F 66 66 69 63 65
- Inside a computer it is:
  1001001 100000 1100001 1101101 100000 1110011 1100101 1100001
  1110100 1100101 1100100 100000 1101001 1101110 100000 1100001
  1101110 100000 1101111 1100110 1100110 1101001 1100011 1100101
- As the sentence contains 24 characters, it needs $24 \times 7 = 168$ bits.

Outline

1 Encodings and Number Systems

2 Compression

3 Priority Queues

4 Huffman Coding

5 Lempel-Ziv-Welch Compression

Compression

Definition

Compression is the process by which we encode some amount of information, representing it using fewer bits than its original representation.

Lossless Compression

- If the reduction comes as a result of detecting and eliminating redundant information, then we talk about *lossless compression*.
- An example of lossless compression is *run-length encoding*.

Example of Run-Length Encoding

- A black and wide image consists of lines that are sequences of black and white pixels:

  □□□□■□□□□□□□□□■■□□□□□□□□□□□□□□□■■□□□□□□□□□□□□□

- Run-length encoding uses runs of data, that is, sequences of the same value that are represented as the value and its count.
- The previous line would be represented as:

  □4■1□9■2□15■2□13

- Note that we can reconstruct the original line exactly as it was, without any loss of information.

Lossy Compression

- In *lossy compression* reduction comes from detecting information that we deem is not really necessary so that it can be removed without discernible loss.
- Examples: JPEG, MPEG-4, MP3.

Morse Code

A	.-	8.04%	J	.---	0.16%	S	...	6.51%	2	..---
B	-...	1.48%	K	-.-	0.54%	T	-	9.28%	3	...--
C	-.-.	3.34%	L	.-..	4.07%	U	..-	2.73%	4	....-
D	-..	3.82%	M	--	2.51%	V	...-	1.05%	5	.....
E	.	12.49%	N	-.	7.23%	W	.--	1.68%	6	-....
F	..-.	2.4%	O	---	7.64%	X	-..-	0.23%	7	--...
G	--.	1.87%	P	.--.	2.14%	Y	-.--	1.66%	8	---..
H	....	5.05%	Q	--.-	0.12%	Z	--..	0.09%	9	----.
I	..	7.57%	R	.-.	6.28%	1	.----		0	-----

Representing “effervescence”

- To represent “effervescence” in ASCII we need $13 \times 7 = 91$ bits.
- However, the word has only 7 different characters.
- Therefore we could use only 3 bits per character, as $2^3 = 8 > 7$.

Representing “effervescence” with 3 bits per Character

E:	000	F:	001	R:	010	V:	011	S:	100	C:	101	N:	110

We need $13 \times 3 = 39$ to represent the word.

Fixed Length and Variable Length Encodings

Definition

An encoding that uses the same number of bits for each symbol is called *fixed length encoding*. An encoding that uses different number of bits for different symbols is called *variable length encoding*.

Basic Idea Behind Variable Length Encoding

Use Frequencies

In variable length encodings we want to use shorter encodings for the characters that appear more frequently.

Character Frequencies in “effervescence”

E: 5	F: 2	R: 1	V: 1	S: 1	C: 2	N: 1

Variable Length Encoding

E:	0	F:	1	R:	11	V:	100	S:	101	C:	10	N:	110

This encoding is wrong!

Wrong Encoding

- The encoding of the word “effervescence” starts with:
  011011
- Suppose we start reading an encoded word that begins with:
  011011
- Does it start with an E (0) followed by two F (1 and 1), or an R (11)?

Prefix-Free Encoding

Definition

An encoding in which no character's code is a prefix of another character's code is called a *prefix-free encoding*.

Prefix-Free Encoding (Contd.)

E:	0	F:	100	R:	1100	V:	1110	S:	1111	C:	101	N:	1101

Correct encoding.

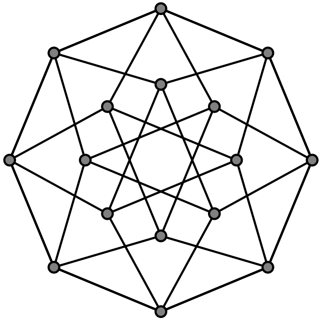
- The word “effervescence” is represented as: 01001000110011100111101011011010
- We use 33 bits in total.

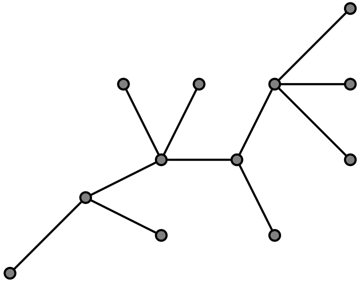
Graphs and Trees

Definition

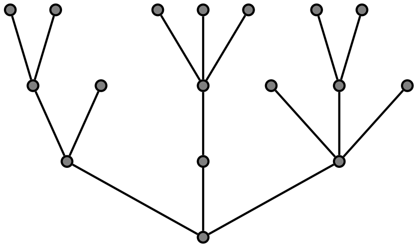
A *tree* is an undirected graph without cycles. Alternatively, we can say that a tree is a data structure with a root node and a set of nodes connected to the root, where each node is a root of another tree.

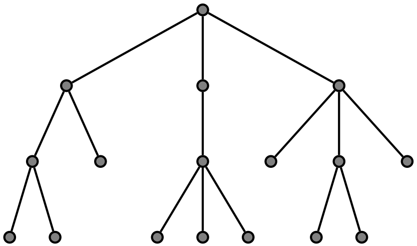
Example of Graph and Tree





Tree Growth





Binary Trees

The number of children of a tree node is called the *degree* of the node.

Definition

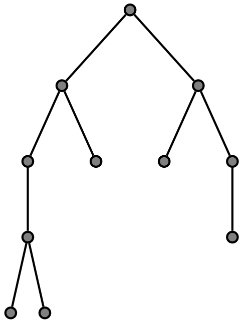
A tree in which each node has no more than two children is called a *binary tree.*

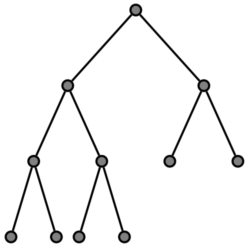
Complete Binary Tree

Definition

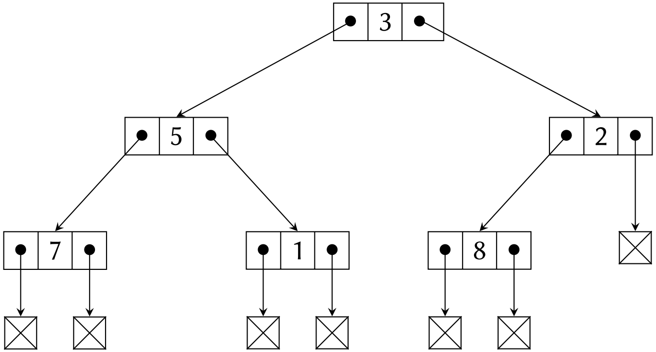
A binary tree that has the smallest possible number of levels is called a *complete binary tree.*

Binary Tree Examples


Binary Tree.


Complete Binary Tree.

Representing Trees with Links



Binary Tree Creation

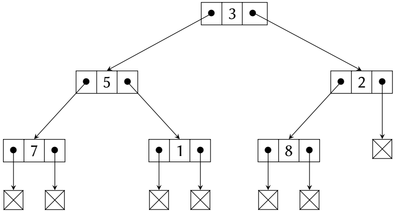
To create binary trees we need the following function:

- `CreateTree`(d, x, y), takes two nodes x and y and creates a new node with data d having x, y as its children, left and right. It returns the newly created node. x and y may be NULL, so that we can create a node with zero, one, or two children.

Binary Tree Creation

To create the left part of the binary tree of our example we would do:

```
$n_1 \leftarrow$ CreateTree(7, NULL, NULL)
$n_2 \leftarrow$ CreateTree(1, NULL, NULL)
$n_3 \leftarrow$ CreateTree(5, $n_1$, $n_2$)
```



Outline

1 Encodings and Number Systems

2 Compression

3 Priority Queues

4 Huffman Coding

5 Lempel-Ziv-Welch Compression

Priority Queues

Definition

A priority queue is a data structure in which when we remove items we remove the item with the largest value, or the smallest value, depending on whether we have a maximum priority queue (max-priority) or a minimum priority queue (min-priority).

Priority Queue Functions

- `CreatePQ()` creates a new, empty priority queue.
- `InsertInPQ(`pq, i`)` inserts i in pq.
- `FindMinInPQ(`pq`)` or `FindMaxInPQ(`pq`)` returns the minimum (for minimum priority queues) or the maximum (for maximum priority queues) element of the queue.
- `ExtractMinFromPQ(`pq`)` or `ExtractMaxFromPQ(`pq`)` removes and returns from the queue the minimum (for minimum priority queues) or the maximum (for maximum priority queues) of the elements of the queue.
- `SizePQ(`pq`)` returns the number of elements in pq.

Minimum Priority Queue Example

1. InsertInPQ(pq, 3)
2. InsertInPQ(pq, 1)
3. InsertInPQ(pq, 4)
4. ExtractMinFromPQ(pq) $\longrightarrow$ 1
5. ExtractMinFromPQ(pq) $\longrightarrow$ 3
6. ExtractMinFromPQ(pq) $\longrightarrow$ 4

Outline

1 Encodings and Number Systems

2 Compression

3 Priority Queues

4 Huffman Coding

5 Lempel-Ziv-Welch Compression

Huffman Coding

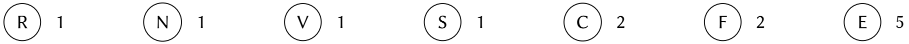
- The Huffman code is an optimal variable length encoding method, that is, it uses the smallest amount of bits to represent the symbols we want to encode.
- The encoding is derived from a tree whose leaves are the symbols we want to encode.

Basic Idea

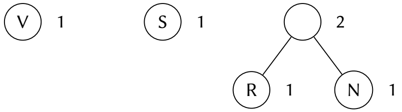
- We put in a minimum priority queue trees, each containing a single node. Each node contains a character along with its frequency.
- We pick the two trees with the smallest frequencies, we take them out of the queue, and we insert into the queue a new tree with the two trees as its children and the sum of their frequency as its frequency.
- We repeat this process until we arrive at a single tree, containing all characters we want to encode.

Huffman Code Construction (1)

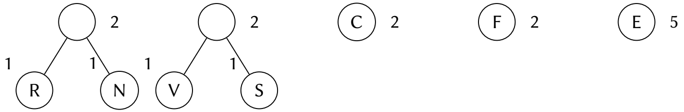
E: 5	F: 2	R: 1	V: 1	S: 1	C: 2	N: 1



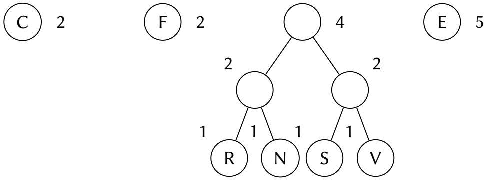
Huffman Code Construction (2)



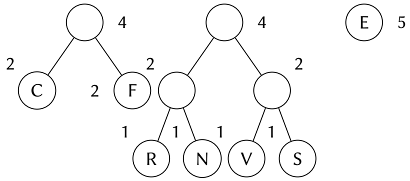
Huffman Code Construction (3)



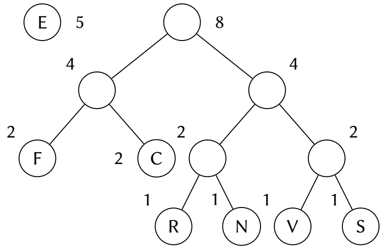
Huffman Code Construction (4)



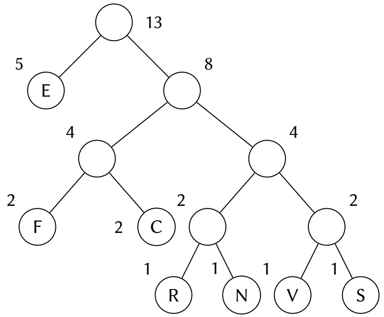
Huffman Code Construction (5)



Huffman Code Construction (6)



Huffman Code Construction (7)

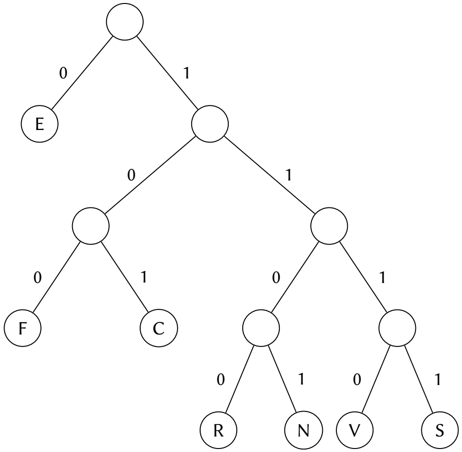


Huffman Code Construction Algorithm

Algorithm: Huffman Code Construction

```
CreateHuffmanCode($pq$) $\longrightarrow$ $hc$
    Input: $pq$, a priority queue
    Output: $hc$, a binary tree representing a Huffman code
1   while SizePQ($pq$) $> 1$ do
2       $x \leftarrow$ ExtractMinFromPQ($pq$)
3       $y \leftarrow$ ExtractMinFromPQ($pq$)
4       $sum \leftarrow$ GetData($x$) + GetData($y$)
5       $z \leftarrow$ CreateTree($sum, x, y$)
6       InsertInPQ($pq, z$)
7   return ExtractMinFromPQ($pq$)
```

Huffman Code for “effervescence”



E: 0	F: 100	R: 1100	V: 1110	S: 1111	C: 101	N: 1101

Encoding Using Huffman Codes

To encode a text using Huffman coding we must read it twice, therefore it is a *two-pass* method.

1. The first time we count character frequencies so that we can create the Huffman coding
2. The second time we go through the text and we substitute each character with its Huffman encoding.

If the text we want to compress has length n characters, the first step, that is, the creation of the Huffman encoding requires $O(n \lg n)$ time.

Huffman Coding Complexity

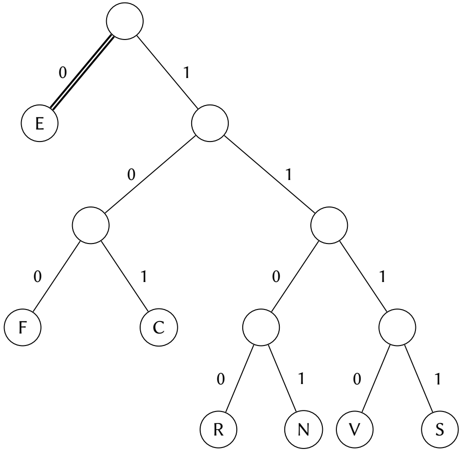
- Why do we need $O(n \lg n)$ time?
- Each time we process the priority queue we remove two elements and add one.
- We do that as long as the queue has more than one element.
- Each time the size of the queue is reduced by one.
- So, if we have n elements initially, we will repeat the steps $n - 1$ times.
- Inserting items in the priority queue requires $O(\lg n)$ time, as does extracting elements from it.
- Therefore we have in total $O((n-1)3\lg n) = O(n \lg n)$.

Decoding Using Huffman Codes

To decode a text that has been encoding using a Huffman code we must have the encoding tree at our disposal.

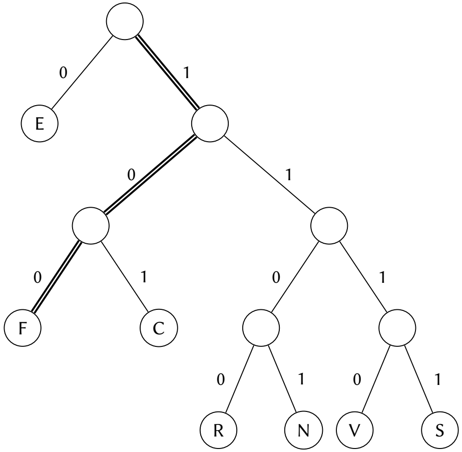
1. We start at the root of the tree.
2. We read a bit from the encoded text.
3. We follow the corresponding branch of the tree.
4. We repeat steps 2–3 until we arrive at a leaf, where we find the encoded character.
5. We go back to step 1 until we have read the whole text.

Decoding Using Huffman Codes (1)



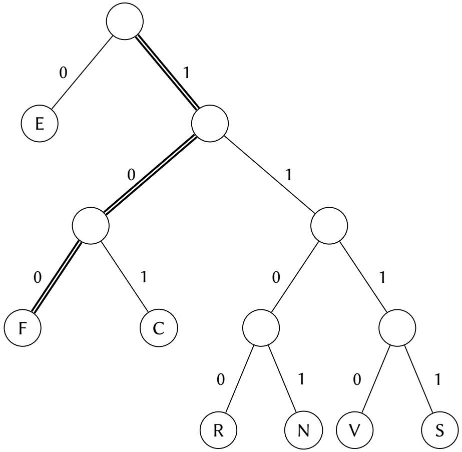
0	1	0	0	1	0	0	0	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0
E																																

Decoding Using Huffman Codes (2)



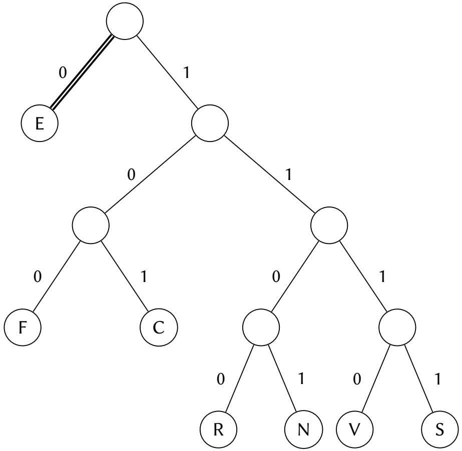
1	0	0	1	0	0	0	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0
	F																														

Decoding Using Huffman Codes (3)



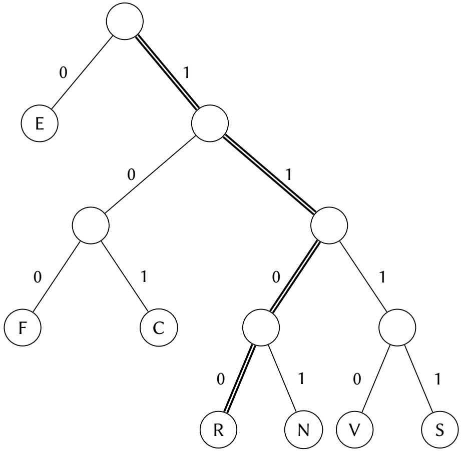
1	0	0	0	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0
	F																											

Decoding Using Huffman Codes (4)



0	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0
E																									

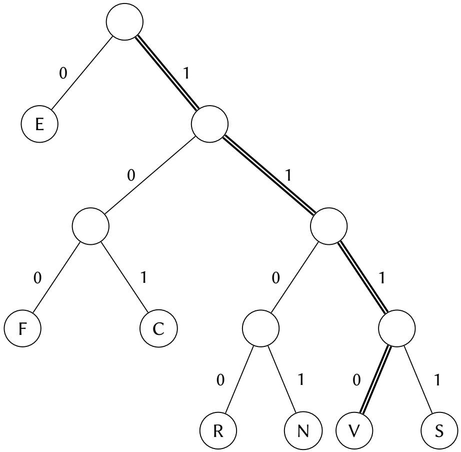
Decoding Using Huffman Codes (5)



1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0

R

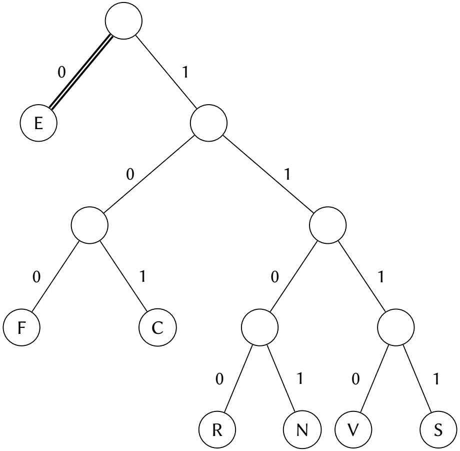
Decoding Using Huffman Codes (6)



1 1 1 0	0 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 0

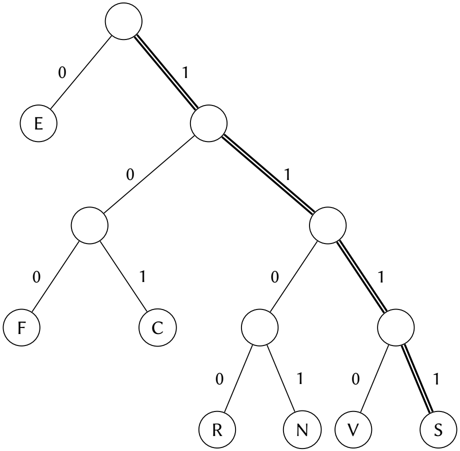
V

Decoding Using Huffman Codes (7)



0	1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0
E																

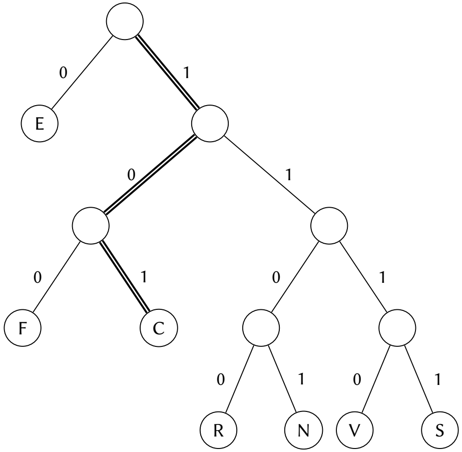
Decoding Using Huffman Codes (8)



1	1	1	1	1	0	1	0	1	1	0	1	1	0	1	0

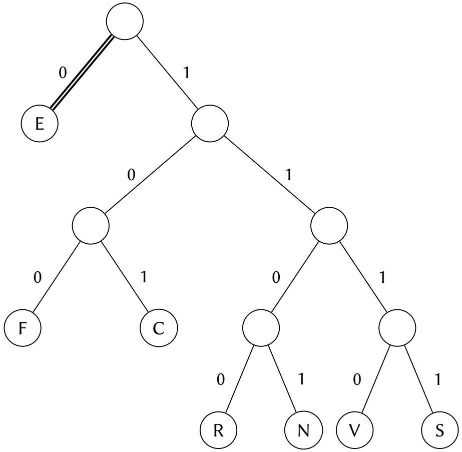
S

Decoding Using Huffman Codes (9)



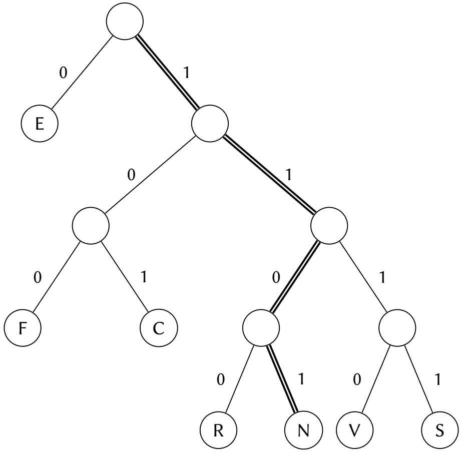
1	0	1	0	1	1	0	1	1	0	1	0
	C										

Decoding Using Huffman Codes (10)



0	1	1	0	1	1	0	1	0
E								

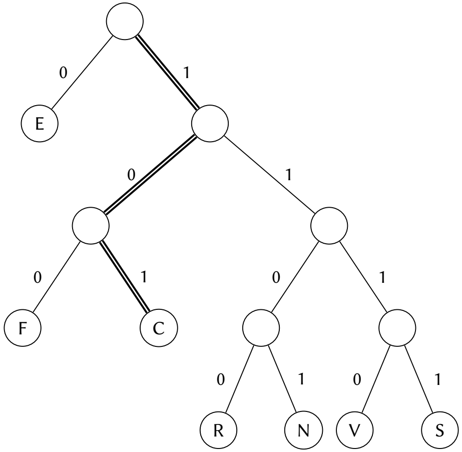
Decoding Using Huffman Codes (11)

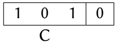


1	1	0	1	1	0	1	0

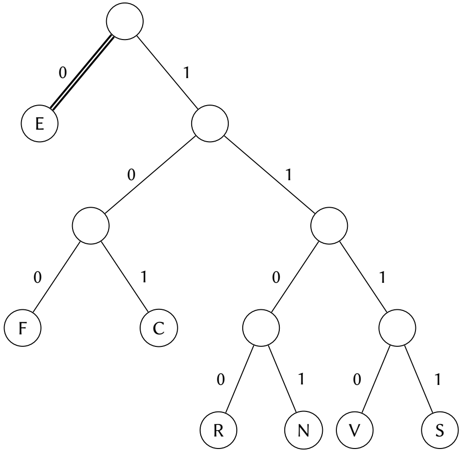
N

Decoding Using Huffman Codes (12)





Decoding Using Huffman Codes (13)



$\boxed{0}$
E

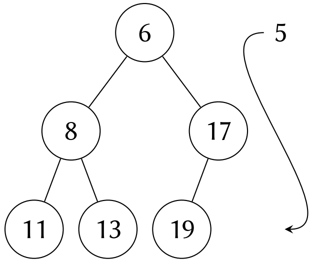
Implementing Priority Queues

Priority queues are implemented with *heaps*.

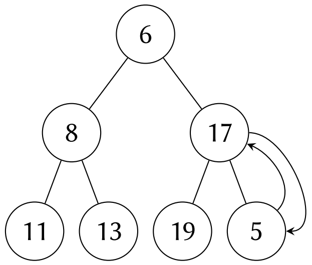
Definition

A heap is a complete binary tree, in which each node is greater than or equal, or less than or equal, to its children.
If each node is greater than or equal to its children, we have a *maximum heap* (max-heap).
If each node is less than or equal to its children, we have a *minimum heap* (min-heap).

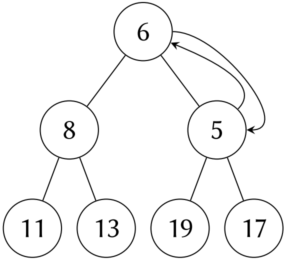
Insertion in a Priority Queue (1)



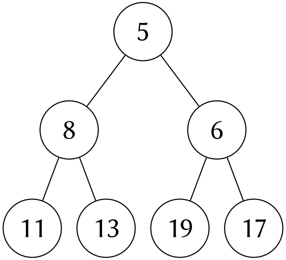
Insertion in a Priority Queue (2)



Insertion in a Priority Queue (3)



Insertion in a Priority Queue (4)



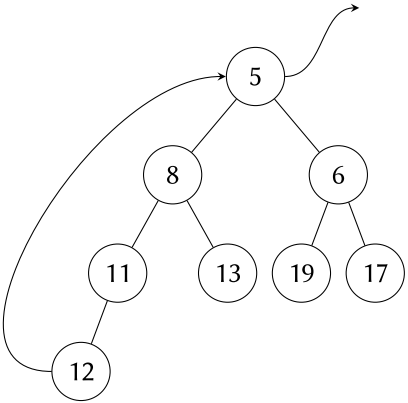
Priority Queue Insertion Algorithm (min-heap)

Algorithm: Priority queue, minimum heap insert.

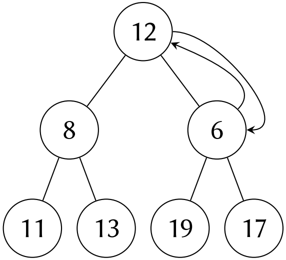
```
InsertInPQ($pq$, $c$)
    Input: $pq$, a priority queue
           $c$, an item to insert to the queue

1   AddLast($pq$, $c$)
2   while $c \neq$ GetRoot($pq$) and GetData($c$) $<$ GetData(Parent($c$)) do
3       $p \leftarrow$ Parent($c$)
4       Exchange($pq$, $c$, $p$)
5       $c \leftarrow p$
```

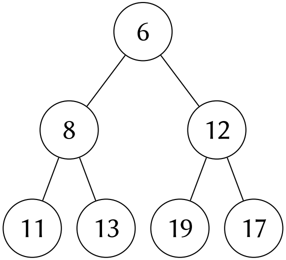
Extraction from a Priority Queue (1)



Extraction from a Priority Queue (2)



Extraction from a Priority Queue (3)



Priority Queue Extraction Algorithm (min-heap)

Algorithm: Priority queue, minimum heap extract minimum.

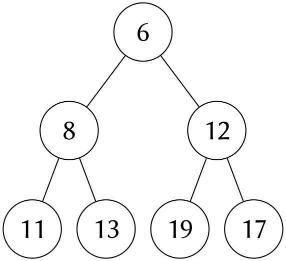
```
ExtractMinFromPQ($pq$) $\rightarrow c$
    Input: $pq$, a priority queue
    Output: $c$, the minimum element of the queue

1  $c \leftarrow$ GetRoot($pq$)
2  SetRoot($pq$, ExtractLastFromPQ($pq$))
3  $i \leftarrow$ GetRoot($pq$)
4  while HasChildren($i$) do
5      $j \leftarrow$ Min(Children($i$))
6      if GetData($i$) $<$ GetData($j$) then
7          return $c$
8      Exchange($pq, i, j$)
9      $i \leftarrow j$
10 return $c$
```

Representing Priority Queues

- Although we usually represents trees with links, it is convenient to represent a priority queue using an array, without wasting space.
- Position 0 of the array contains the root.
- For each node i, its left child is at position $2i + 1$ of the array and its right child is at position $2i + 2$.
- Conversely, if we are at node j, its parent is at position $\lfloor (j - 1)/2 \rfloor$.

Priority Queue Representation Example



6	8	12	11	13	19	17

Outline

1. Encodings and Number Systems
2. Compression
3. Priority Queues
4. Huffman Coding
5. Lempel-Ziv-Welch Compression

Lempel-Ziv-Welch Compression Idea

- Lempel-Ziv-Welch (LZW) was invented by Abraham Lempel, Jacob Ziv, and Terry Welch.
- The basic idea is that, instead of varying the length of the encodings for our items, we vary the length of the *items we want to encode.*

Variable Item Length Encoding

- Suppose we have a text encoded in ASCII.
- With eight bits we can represent $2^8 = 256$ different items.
- We use the numbers from 0x00 (0) to 0x7F (127) to represent the ASCII characters.
- Then we have the numbers from 0x80 (128) to 0xFF (255) available to represent *sequences of characters.*

N-Grams

- Sequences of two letters are called *bigrams.*
- sequences of three letters are called *trigrams.*
- Longer sequences are called by their value plus the “gram” suffix, like “four-gram.”
- In general we have *n-grams.*
- An n-gram with a single item is called a *unigram.*

LZW N-Grams

- In LZW, we will be encoding single characters with their ASCII values and n-grams with the remaining values from our encoding.
- We will be determining these n-grams and their encodings as we go.
- In essence, we will be assigning a new value for each new n-gram we will encounter (as long as we have available values).
- We will use a table to associate n-grams with their encodings.
- We start with the encoding table containing the encodings for unigrams.

LZW Compression Example

t = {…, ␣: 32, …,
A: 65, B: 66, C: 67, D: 68, E: 69, F: 70, G: 71, H: 72, I: 73
J: 74, K: 75, L: 76, M:77, N: 78, O: 79, P: 80, Q: 81, R: 82
S: 83, T: 84, U: 85, V: 86, W: 87, X: 88, Y: 89, Z: 90, …}

	$\boxed{\text{M}}$	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	O	W	
M ◁ 77	**M**	$\boxed{\text{E}}$	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	O	W	{ ME: 128 } → t
E ◁ 69	M	**E**	$\boxed{\text{L}}$	L	O	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	O	W	{ EL: 129 } → t
L ◁ 76	M	E	**L**	$\boxed{\text{L}}$	O	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	O	W	{ LL: 130 } → t
L ◁ 76	M	E	L	**L**	$\boxed{\text{O}}$	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	O	W	{ LO: 131 } → t
O ◁ 79	M	E	L	L	**O**	$\boxed{\text{W}}$	␣	Y	E	L	L	O	W	␣	F	E	L	L	O	W	{ OW: 132 } → t
W ◁ 87	M	E	L	L	O	**W**	$\boxed{␣}$	Y	E	L	L	O	W	␣	F	E	L	L	O	W	{ W␣ 133 } → t
␣ ◁ 32	M	E	L	L	O	W	**␣**	$\boxed{\text{Y}}$	E	L	L	O	W	␣	F	E	L	L	O	W	{ ␣Y: 134 } → t
Y ◁ 89	M	E	L	L	O	W	␣	**Y**	$\boxed{\text{E}}$	L	L	O	W	␣	F	E	L	L	O	W	{ YE : 135 } → t
	M	E	L	L	O	W	␣	Y	**E**	$\boxed{\text{L}}$	L	O	W	␣	F	E	L	L	O	W	
EL ◁ 129	M	E	L	L	O	W	␣	Y	**E**	**L**	$\boxed{\text{L}}$	O	W	␣	F	E	L	L	O	W	{ ELL: 136 } → t
	M	E	L	L	O	W	␣	Y	E	L	**L**	$\boxed{\text{O}}$	W	␣	F	E	L	L	O	W	
LO ◁ 131	M	E	L	L	O	W	␣	Y	E	L	**L**	**O**	$\boxed{\text{W}}$	␣	F	E	L	L	O	W	{ LOW: 137 } → t
	M	E	L	L	O	W	␣	Y	E	L	L	O	**W**	$\boxed{␣}$	F	E	L	L	O	W	
W␣ ◁ 133	M	E	L	L	O	W	␣	Y	E	L	L	O	**W**	**␣**	$\boxed{\text{F}}$	E	L	L	O	W	{ W␣F: 138 } → t
F ◁ 70	M	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	**F**	$\boxed{\text{E}}$	L	L	O	W	{ FE: 139 } → t
	M	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	**E**	$\boxed{\text{L}}$	L	O	W	
	M	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	**E**	**L**	$\boxed{\text{L}}$	O	W	
ELL ◁ 136	M	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	**E**	**L**	**L**	$\boxed{\text{O}}$	W	{ ELLO: 140 } → t
	M	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	**O**	$\boxed{\text{W}}$	
OW ◁ 132	M	E	L	L	O	W	␣	Y	E	L	L	O	W	␣	F	E	L	L	**O**	**W**	

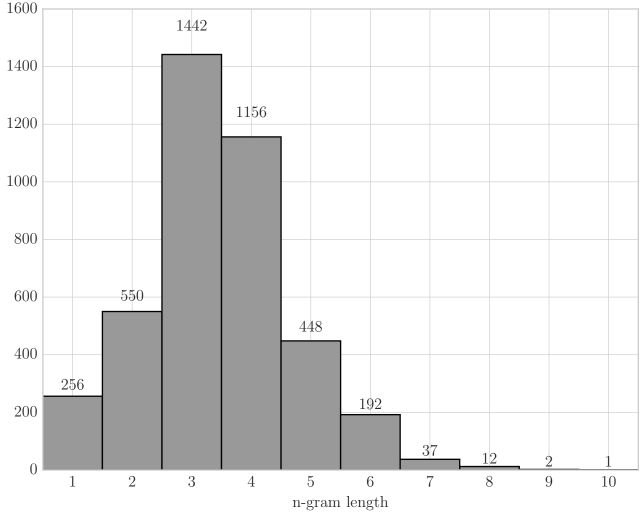
LZW Overall Logic

- Read a character.
- Use it to extend the current n-gram.
- If the resulting n-gram is in the table, repeat by reading the next character.
- Otherwise insert the new n-gram into the table, output the code for the previous n-gram, start a new n-gram with the character just read, and repeat by reading the next character.

LZW Real Encoding Example

- We can use 16 bits for encoding.
- Then the codes from 0 up to and including 255 represent the individual characters.
- The values from 256 to 4095 are used to represent n-grams.
- If we run LZW with these parameters on James Joyce’s *Ulysses*, we reduce the text to about 41% of its original size.

Distribution of LZW N-Grams in *Ulysses*



Maps, Dictionaries, Associative Arrays

- To implement LZW we need a way to go from n-grams to their encodings.
- We will use a data structure called a *map*, or *associative array*, or *dictionary*.
- It is called a map because it maps items, called *keys* to their values.
- Map lookup and insertion can be performed in $O(1)$ time.

Map Functions

- `CreateMap()` creates a new empty map.
- `InsertInMap(`t, k, v`)` inserts item k into the map t with value v.
- `Lookup(`t, k`)` performs a lookup for item k in the map t; it returns the value associated with v, if it exists, or NULL if the map t does not contain k.

LZW Compression Algorithm

Algorithm: LZW compression.

```
LZWCompress($s, nb, n$) $\rightarrow$ $compressed$
    Input: $s$, a string to compress
           $nb$, the number of bits used to represent an item
           $n$, the number of items in the alphabet
    Output: $compressed$, a list containing the numbers representing $s$
            according to LZW compression

    $compressed \leftarrow$ CreateList()
    $max\_code \leftarrow 2^{nb} - 1$
    $t \leftarrow$ CreateMap()
    for $i \leftarrow 0$ to $n$ do
        InsertInMap($t$, Char($i$), $i$)
    $code \leftarrow n$

    $w \leftarrow$ CreateString()
    $p \leftarrow$ NULL
    foreach $c$ in $s$ do
        $wc \leftarrow w + c$
        $v \leftarrow$ Lookup($t, wc$)
        if $v \neq$ NULL then
            $w \leftarrow wc$
        else
            $v \leftarrow$ Lookup($t, w$)
            $p \leftarrow$ InsertInList($compressed, p, v$)
            $w \leftarrow c$
            if $code \leq max\_code$ then
                InsertInMap($t, wc, code$)
                $code \leftarrow code + 1$

    if $|w| > 0$ then
        InsertInList($compressed, p, v$)
    return $compressed$
```

LZW Decompression

- To decompress a text encoded using LZW, we need to reconstruct the map with the encodings and use it in reverse (from the encoded values to the n-grams).
- To do that, we create a decoding table using the n-grams we encounter.
- We start with the decoding table containing the unigrams.
- Then, for every value we encounter, we look it up in the decoding table.
- Then, we enter into the decoding table the n-gram consisting of the previous output and the first character of the current output.

LZW Decompression Example

dt = {…, 32: ␣, …,
65: A, 66: B, 67: C, 68: D, 69: E, 70: F, 71: G, 72: H, 73: I
74: J, 75: K, 76: L, 77: M, 78: N, 79: O, 80: P, 81: Q, 82: R
83: S, 84: T, 85: U, 86: V, 87: W, 88: X, 89: Y, 90: Z, …}

77 ◁ M	$\boxed{77}$	69	76	76	79	87	32	89	129	131	133	70	136	132	
69 ◁ E	77	$\boxed{69}$	76	76	79	87	32	89	129	131	133	70	136	132	{ 128: ME } → dt
76 ◁ L	77	69	$\boxed{76}$	76	79	87	32	89	129	131	133	70	136	132	{ 129: EL } → dt
76 ◁ L	77	69	76	$\boxed{76}$	79	87	32	89	129	131	133	70	136	132	{ 130: LL } → dt
79 ◁ O	77	69	76	76	$\boxed{79}$	87	32	89	129	131	133	70	136	132	{ 131: LO } → dt
87 ◁ W	77	69	76	76	79	$\boxed{87}$	32	89	129	131	133	70	136	132	{ 132: OW } → dt
32 ◁ ␣	77	69	76	76	79	87	$\boxed{32}$	89	129	131	133	70	136	132	{ 133: W␣ } → dt
89 ◁ Y	77	69	76	76	79	87	32	$\boxed{89}$	129	131	133	70	136	132	{ 134: ␣Y } → dt
129 ◁ EL	77	69	76	76	79	87	32	89	$\boxed{129}$	131	133	70	136	132	{ 135: YE } → dt
131 ◁ LO	77	69	76	76	79	87	32	89	129	$\boxed{131}$	133	70	136	132	{ 136: ELL } → dt
133 ◁ W␣	77	69	76	76	79	87	32	89	129	131	$\boxed{133}$	70	136	132	{ 137: LOW } → dt
70 ◁ F	77	69	76	76	79	87	32	89	129	131	133	$\boxed{70}$	136	132	{ 138: W␣F } → dt
136 ◁ ELL	77	69	76	76	79	87	32	89	129	131	133	70	$\boxed{136}$	132	{ 139: LOW } → dt
132 ◁ OW	77	69	76	76	79	87	32	89	129	131	133	70	136	$\boxed{132}$	{ 140: ELLO } → dt

Attention: Corner Case

- This process does not work always.
- It will not work when, during compression, we create an n-gram and output its encoding immediately.
- Then, during decompression, we do not have enough time to create the decoding and enter it into the decoding table.
- This is an example of a *corner case*. A corner case is an extreme situation handled by an algorithm or a computer program.

LZW Corner Case Compression

$$\begin{aligned} t = \{&\ldots, ␣: 32, \ldots,\\ &\text{A: 65, B: 66, C: 67, D: 68, E: 69, F: 70, G: 71, H: 72, I: 73}\\ &\text{J: 74, K: 75, L: 76, M:77, N: 78, O: 79, P: 80, Q: 81, R: 82}\\ &\text{S: 83, T: 84, U: 85, V: 86, W: 87, X: 88, Y: 89, Z: 90, } \ldots\} \end{aligned}$$

	$\boxed{A}$	B	A	B	A	B	A	
A $\lhd$ 65	**A**	$\boxed{B}$	A	B	A	B	A	$\{$ AB: 128 $\}$: $\longrightarrow t$
B $\lhd$ 66	A	**B**	$\boxed{A}$	B	A	B	A	$\{$ BA: 129 $\} \longrightarrow t$
	A	B	**A**	$\boxed{B}$	A	B	A	
AB $\lhd$ 128	A	B	**A**	**B**	$\boxed{A}$	B	A	$\{$ ABA: 130 $\} \longrightarrow t$
	A	B	A	B	**A**	$\boxed{B}$	A	
ABA $\lhd$ 130	A	B	A	B	**A**	**B**	$\boxed{A}$	

LZW Corner Case Decompression

$dt = \{\ldots, 32\colon \text{␣}, \ldots,$
$65\colon \text{A}, 66\colon \text{B}, 67\colon \text{C}, 68\colon \text{D}, 69\colon \text{E}, 70\colon \text{F}, 71\colon \text{G}, 72\colon \text{H}, 73\colon \text{I}$
$74\colon \text{J}, 75\colon \text{K}, 76\colon \text{L}, 77\colon \text{M}, 78\colon \text{N}, 79\colon \text{O}, 80\colon \text{P}, 81\colon \text{Q}, 82\colon \text{R}$
$83\colon \text{S}, 84\colon \text{T}, 85\colon \text{U}, 86\colon \text{V}, 87\colon \text{W}, 88\colon \text{X}, 89\colon \text{Y}, 90\colon \text{Z}, \ldots\}$

65 ◁ A	$\boxed{65}$	66	128	130	
66 ◁ B	65	$\boxed{66}$	128	130	{ AB: 128 } $\longrightarrow dt$
128 ◁ AB	65	66	$\boxed{128}$	130	{ BA: 129 } $\longrightarrow dt$
130 ◁ ABA	65	66	128	$\boxed{130}$	{ ABA: 130 } $\longrightarrow dt$

Corner Case Decompression Solution

- Suppose we encounter an encoded value that we have not entered into the decoding table yet.
- We can enter into the decoding table a new entry whose key will be the last n-gram we entered with its first character appended to it.
- Then we can output that newly created n-gram.

LZW Decompression Algorithm

Algorithm: LZW decompression.

```
LZWDecompress($compressed, nb, n$) → $decompressed$
    Input: $compressed$, a list representing a compressed string
           $nb$, the number of bits used to represent an item
           $n$, the number of items in the alphabet
    Output: $decompressed$, the original string
    $max\_code \leftarrow 2^{nb} - 1$
    $dt \leftarrow$ CreateMap()
    for $i \leftarrow 0$ to $n$ do
        InsertInMap($dt, i$, Char($i$))
    $code \leftarrow n$
    $decompressed \leftarrow$ CreateString()
    $c \leftarrow$ GetNextListNode($compressed$, NULL)
    RemoveListNode($compressed$, NULL, $c$)
    $v \leftarrow$ Lookup($dt$, GetData($c$))
    $decompressed \leftarrow decompressed + v$
    $pv \leftarrow v$
    foreach $c$ in $compressed$ do
        $v \leftarrow$ Lookup($dt, c$)
        if $v =$ NULL then
            $v \leftarrow pv + pv[0]$
        $decompressed \leftarrow decompressed + v$
        if $code \leq max\_code$ then
            InsertInMap($dt, code, pv + v[0]$)
            $code \leftarrow code + 1$
        $pv \leftarrow v$
    return $decompressed$
```

LZW Complexity

- The LZW algorithm needs only a single pass through its input: it is a *single-pass* method.
- The speed of the algorithm depends on the speed with which we can manipulate the encoding and decoding tables.
- A map can be implemented so that insertion and lookup can be performed in constant, $O(1)$ time.
- Therefore, compression as well as decompression need linear time, $O(n)$, where n is the length of the input.