

Lecture 18: Huffman Coding, ADT and Stacks

Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

CODING ON A MODERN COMPUTER

- All data in a computer is maintained in memory cells that can hold one “bit” of data:
 - a 0 or a 1.
- Alphabet symbols (e.g., letters, numbers, etc.) are usually encoded with 8 bits (for 256 combinations).
- You could get by with less—say 7, but 8 is convenient for other reasons.

Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k

Example: How would we write “hello” in ASCII binary?

- h = 01101000
- e = 01100101
- l = 01101100
- o = 01101111

hello = 01101000 01100101 01101100 01101100 01101111

How much data is required? Well every 1 or 0 is a bit.

Each letter requires 8 bits

Hello has 5 letters

So total is 40 bits

Now consider a sequence of words...

“Hello Kaleel everything edible?”



- You may say this sentence if we are having lunch together. What do you notice about the frequency of the letters?
- 27 letters and “e” appears 7 times so roughly 26% of the letters are e.

AN APPLICATION OF BINARY TREES: HUFFMAN CODING

- Consider the problem of digitally representing documents in English.
 - One straightforward approach: each letter is represented by an 8-bit sequence.
 - (This gives 256 “letters,” enough for basic text.)
- Now consider the natural data compression problem:
 - We’d like to store documents using the least number of bits.
- If the documents are just random sequences of these 256 “letters” there’s nothing you can do.
- However, in English text, for example, the letter “a” is far more common than the letter “z”.
 - Perhaps we can exploit this by coding “a” using a shorter bit string, and “make up for this” by using a longer bit string for “z”?

VARIABLE-LENGTH ENCODING

- This suggests a variable-length encoding.
 - “Frequent” letters should get short encodings;
 - “Infrequent” letters will be stuck with the longer encodings
 - ...turns out this is a good idea!
- Problem: Some care is necessary...how do we “decode”?
 - If
 - 0 represents a
 - 1 represents e
 - 01 represents c

we don't know how to decode “01”!

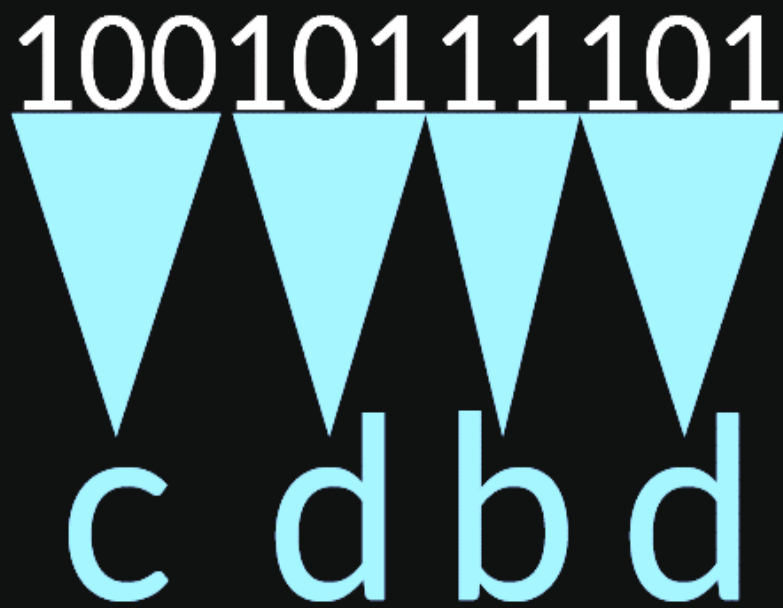
PREFIX-FREE CODES

- Solution: *Prefix-free encoding*--no “codeword” is a prefix of any other codeword.
- Scanning an encoded string left-to-right, one can decode in a unique fashion.
 - Why?
- Once a codeword is observed in the sequence, it cannot be the prefix of any other codeword; then you can decode with confidence!

A PREFIX FREE CODE IN ACTION

- 0 represents a
- 11 represents b
- 100 represents c
- 101 represents d

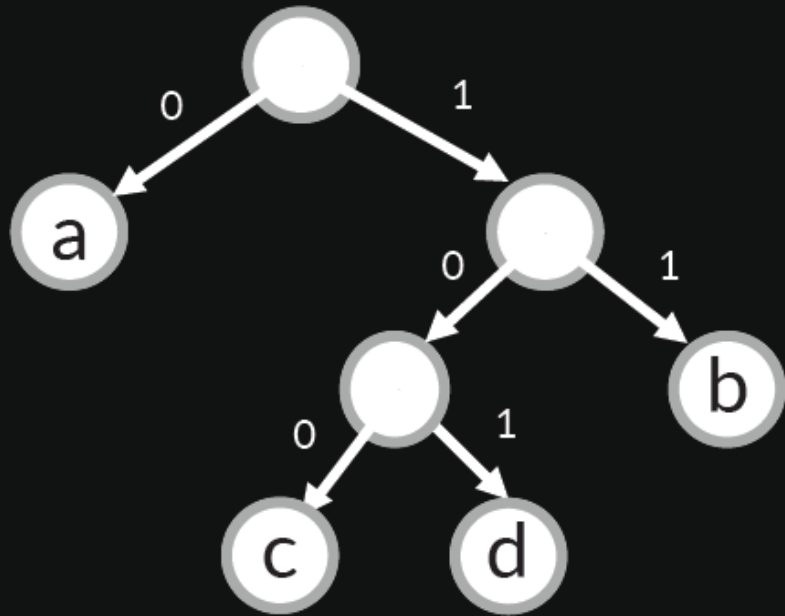
1001011101



c d b d

PREFIX-FREE CODES CAN BE REPRESENTED AS BINARY TREES

- One natural way to represent a prefix-free code is as a binary tree where leaves are labelled with alphabet symbols.

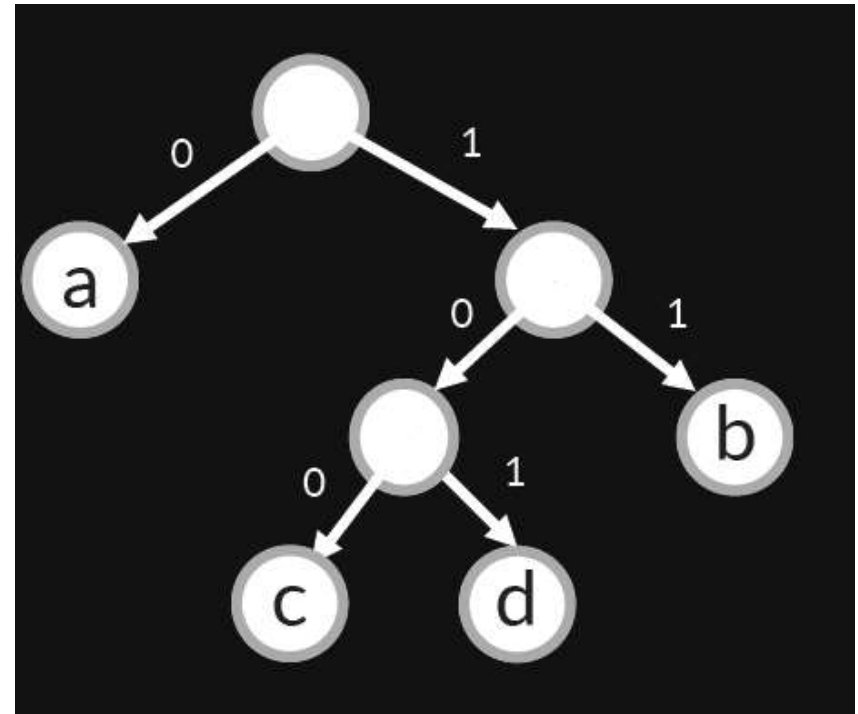


Moving left in the tree corresponds to a 0; moving right a 1.

- 0 represents a
- 11 represents b
- 100 represents c
- 101 represents d

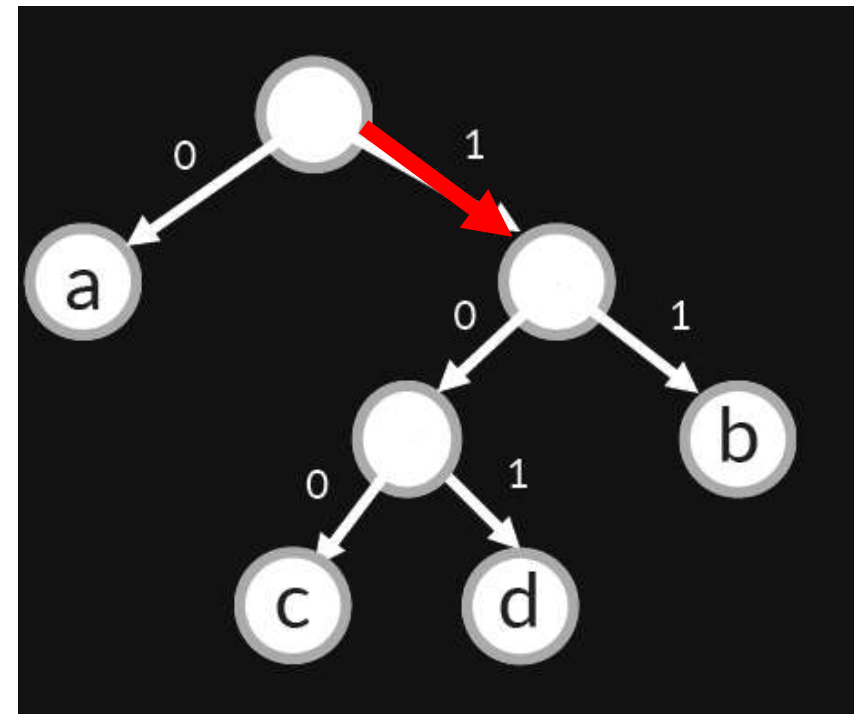
Example Decoding

Decode: 110101



Example Decoding

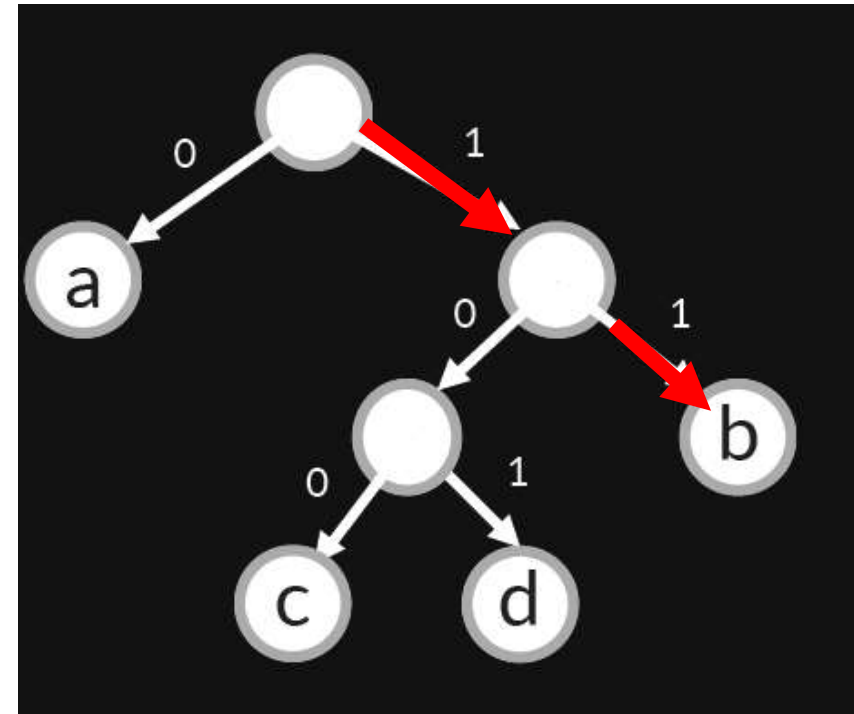
Decode: **1**10101



Example Decoding

Decode: **11**0101

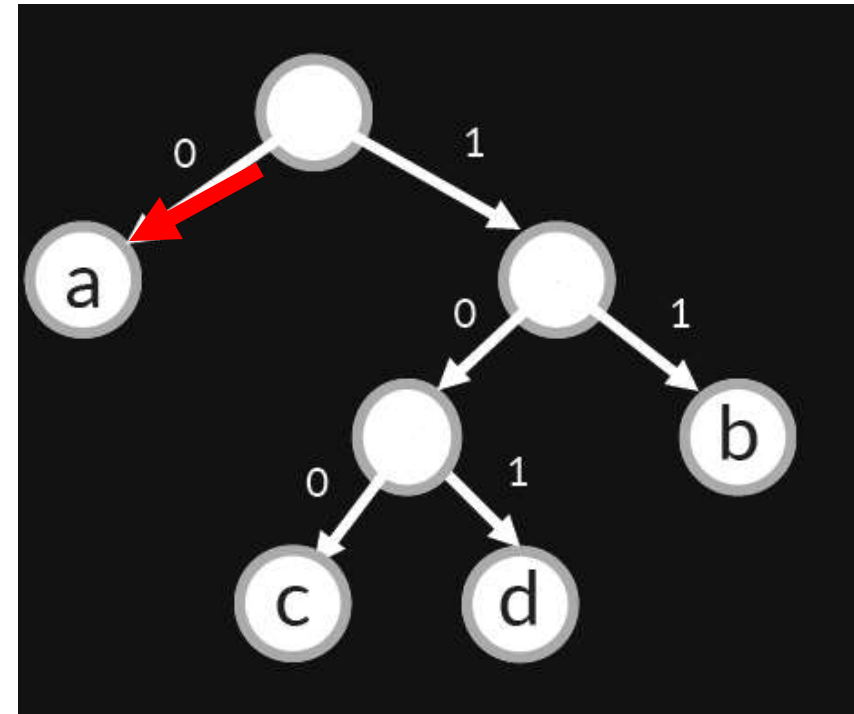
Letters: B



Example Decoding

Decode: 110101

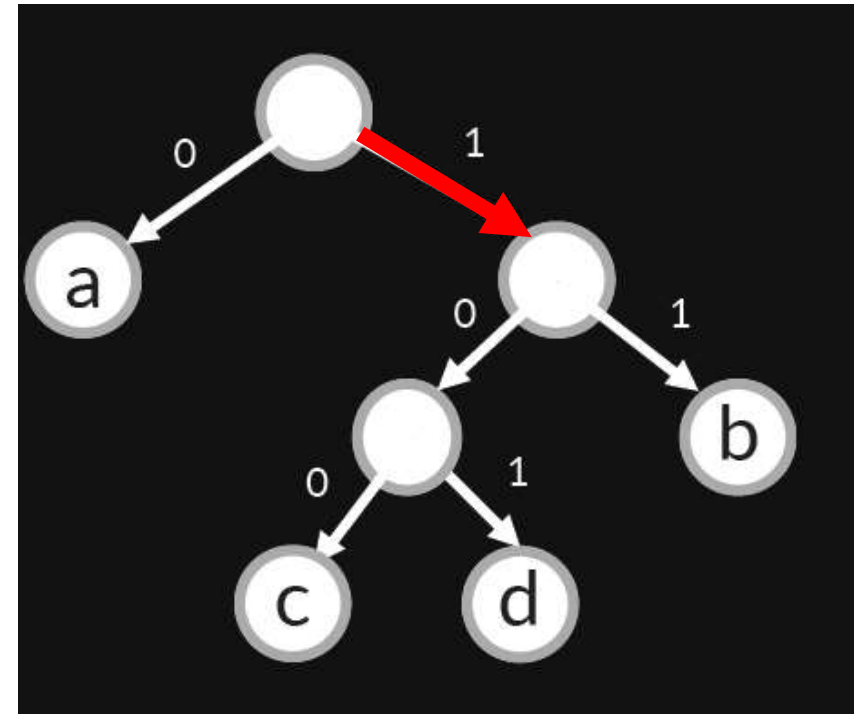
Letters: BA



Example Decoding

Decode: 110**1**01

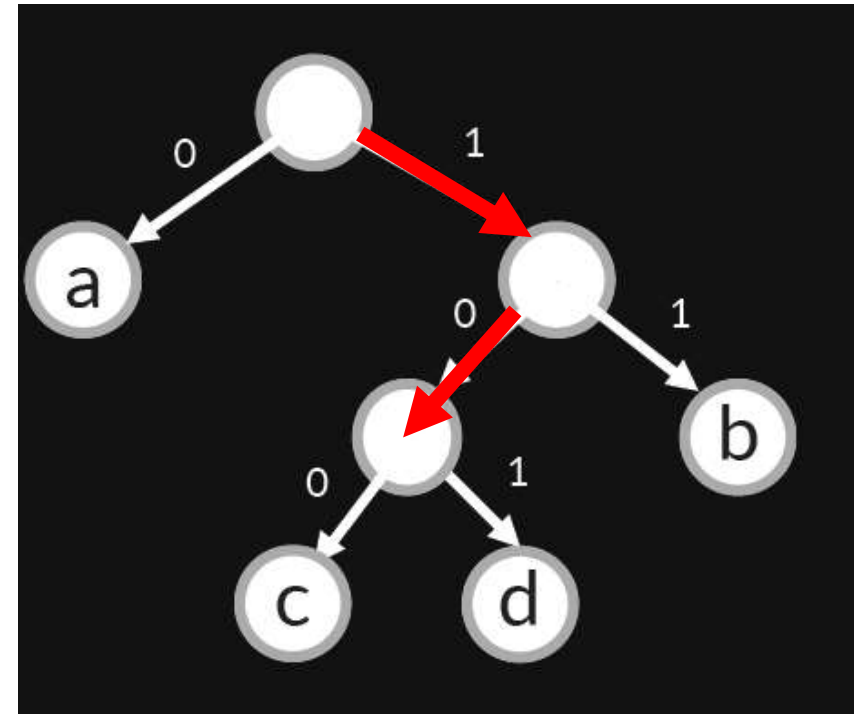
Letters: BA



Example Decoding

Decode: 110**1**01

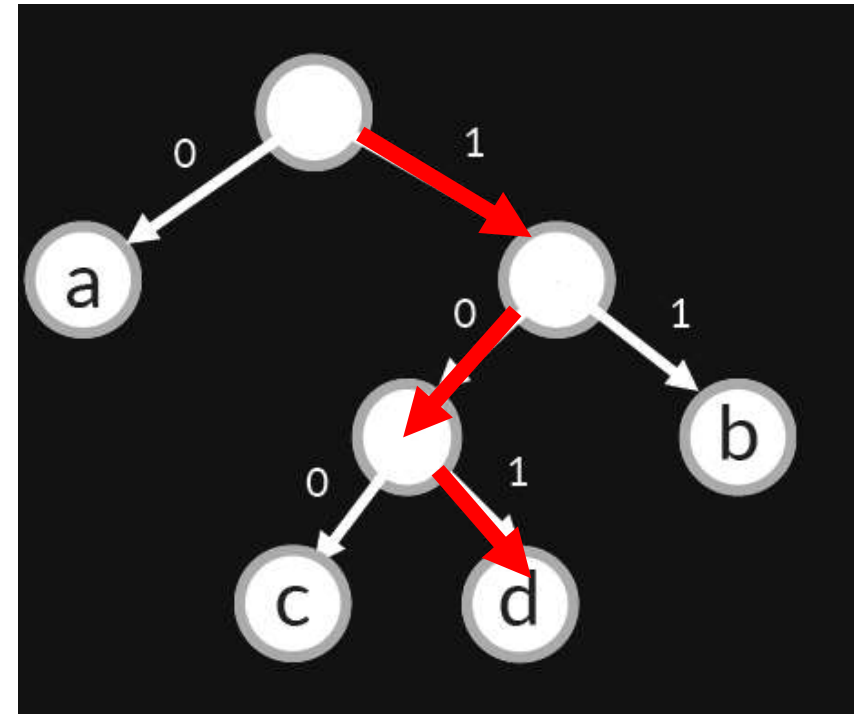
Letters: BA



Example Decoding

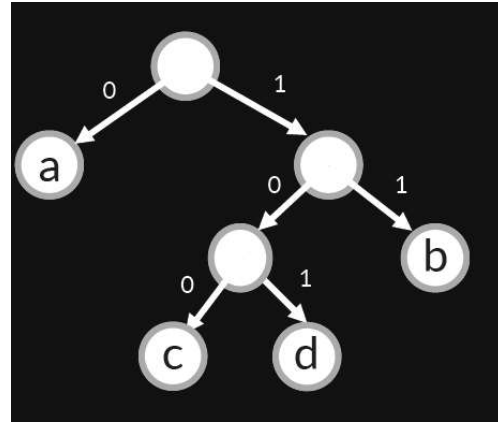
Decode: 110**101**

Letters: BAD



- And we've reached the end of the sequence so we are done decoding.

Why is having different length decode paths for different letters important?



???

- Recall that each time we traverse the tree, the time taken is proportional to the time it takes to reach a leaf node.
- If we make the leaf nodes (letters) that appear more frequently have shorter paths, it will take less time to decode.
- Basic concept: If you have to travel somewhere frequently, having it closer means quicker.

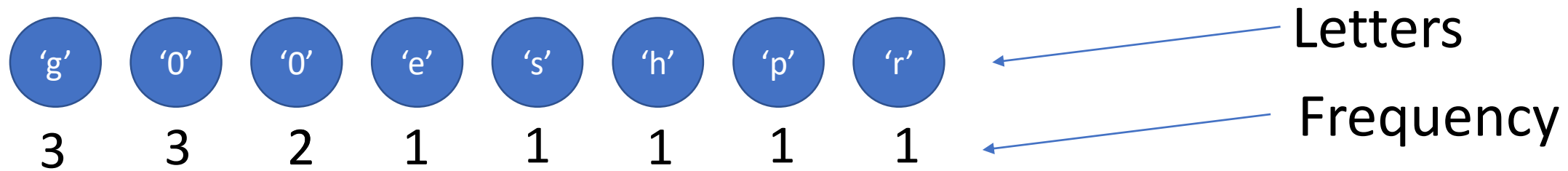
GIVEN FREQUENCIES... WE WANT THE BEST TREE

- Consider frequencies for English:
 - “a” appears 8.1% of the time,
 - “b” 1.4%, ...
- Consider a long document D that exactly matches this distribution (8.1% of the letters are “a”s, etc.)
 - We want a code that minimizes the total length of the encoded document.
 - If we let $\text{frequency}(L)$ denote the number of times the letter L appears, what is our goal?
- **Goal:** Find the code that minimizes

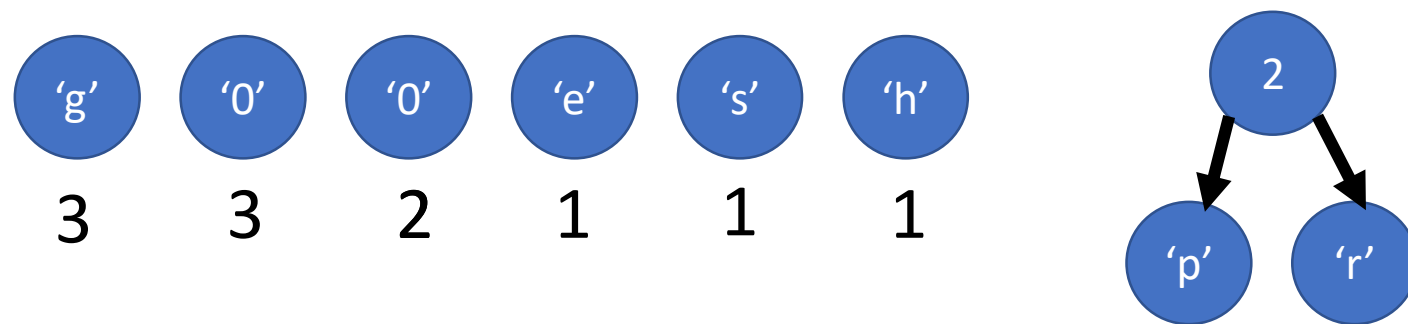
$$\text{Total Length} = \sum_{\text{letters } \ell} \text{codelength}(\ell) \cdot \text{frequency}(\ell)$$

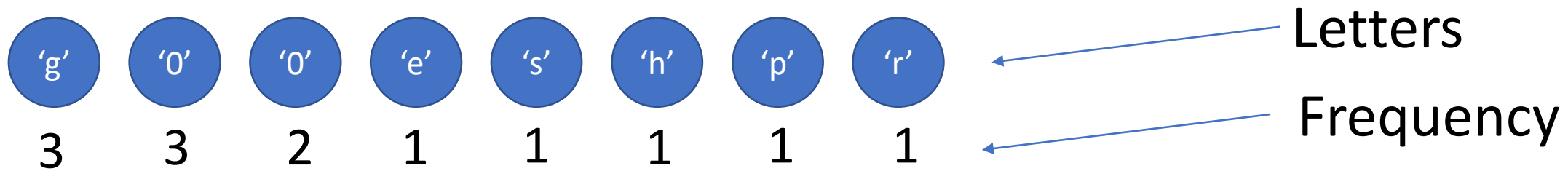
DETERMINING THE OPTIMAL CODE... HUFFMAN'S IDEA

- Begin with each letter in its own tree.
 - The *weight* of the tree is the frequency of the letter.
- Find the two lightest trees.
 - **Join** them with a root above;
 - new weight is their sum.
- Repeat until a single tree emerges.
- Huffman proved that this tree is **optimal**:
 - it encodes any such document with the least number of bits.
 - Equivalently, it minimizes the *average codelength*, when symbols are drawn from the document at random.
 - (Note that it depends on the distribution of the input symbol.)

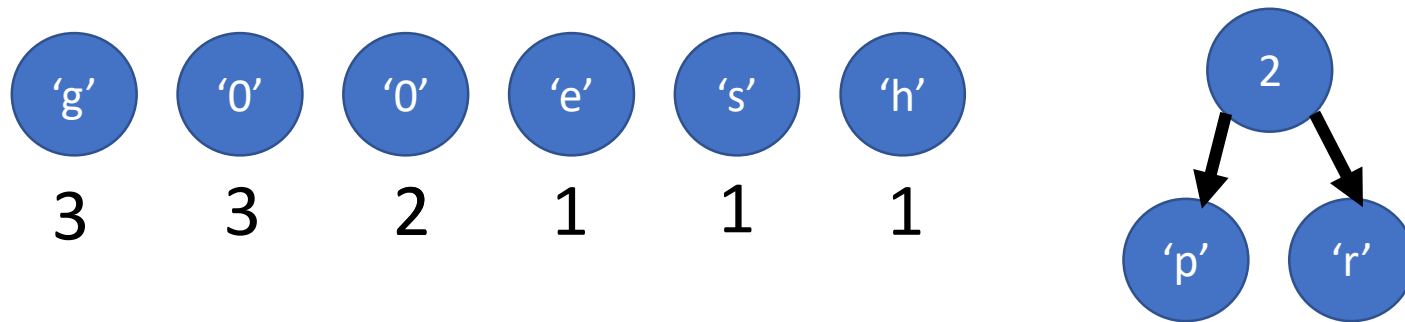


Step 1: Pick the smallest weighted tree and combine:





Step 1: Pick the smallest weighted tree and combine:



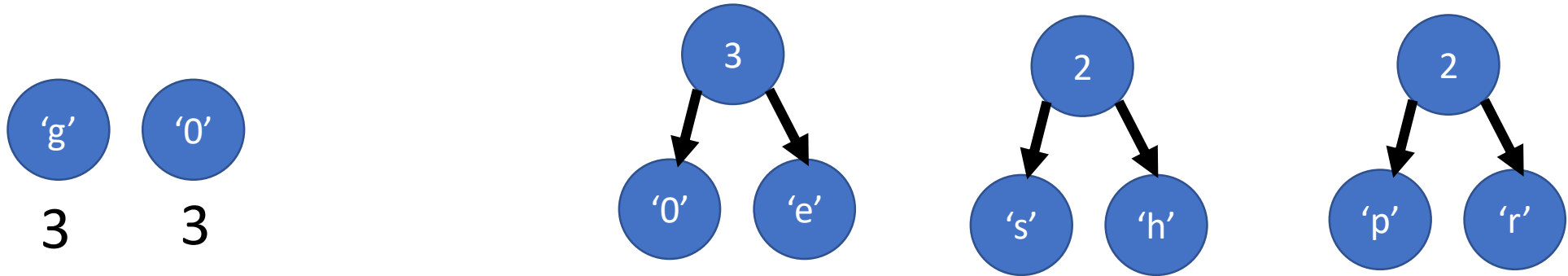
Step 2: Pick the smallest weighted tree and combine:



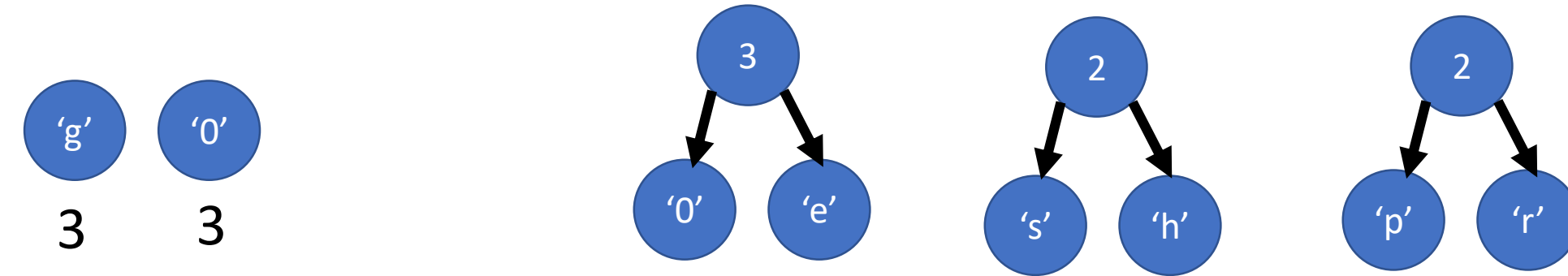
Step 2: Pick the smallest weighted tree and combine:



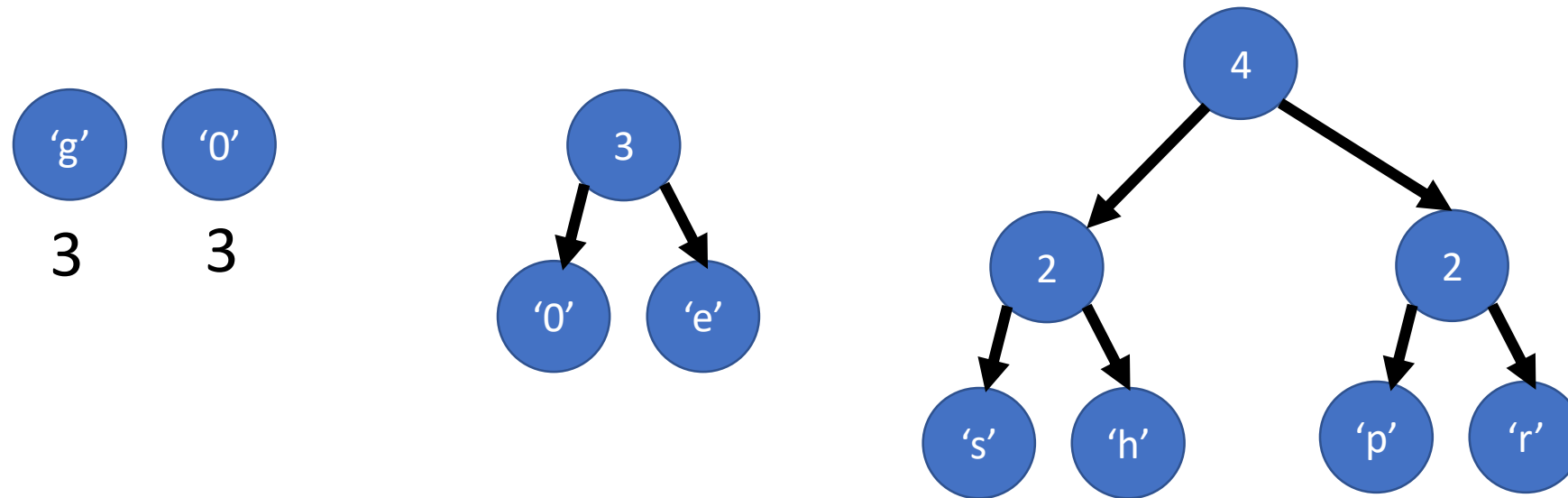
Step 3: Pick the smallest weighted tree and combine:



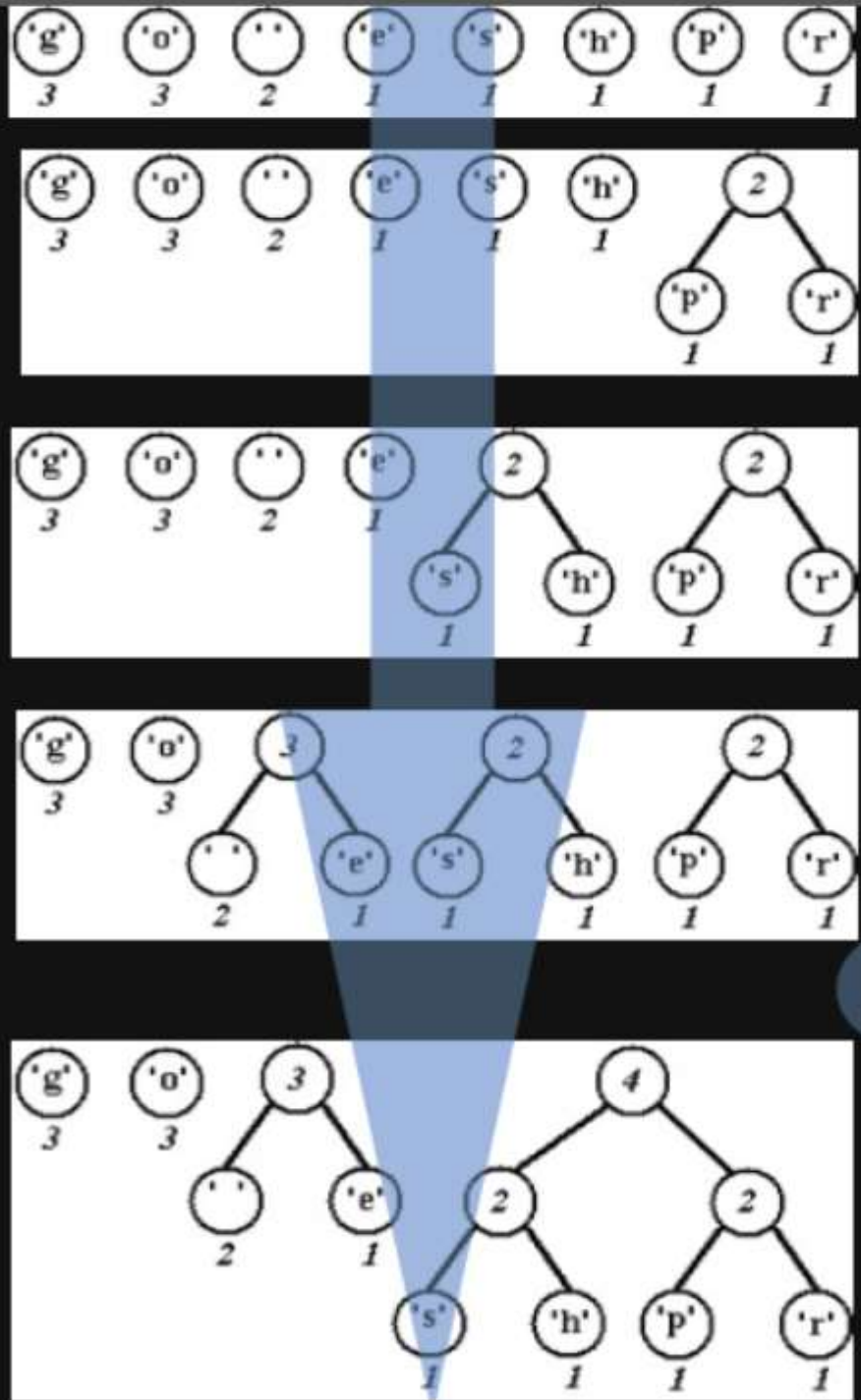
Step 3: Pick the smallest weighted tree and combine:



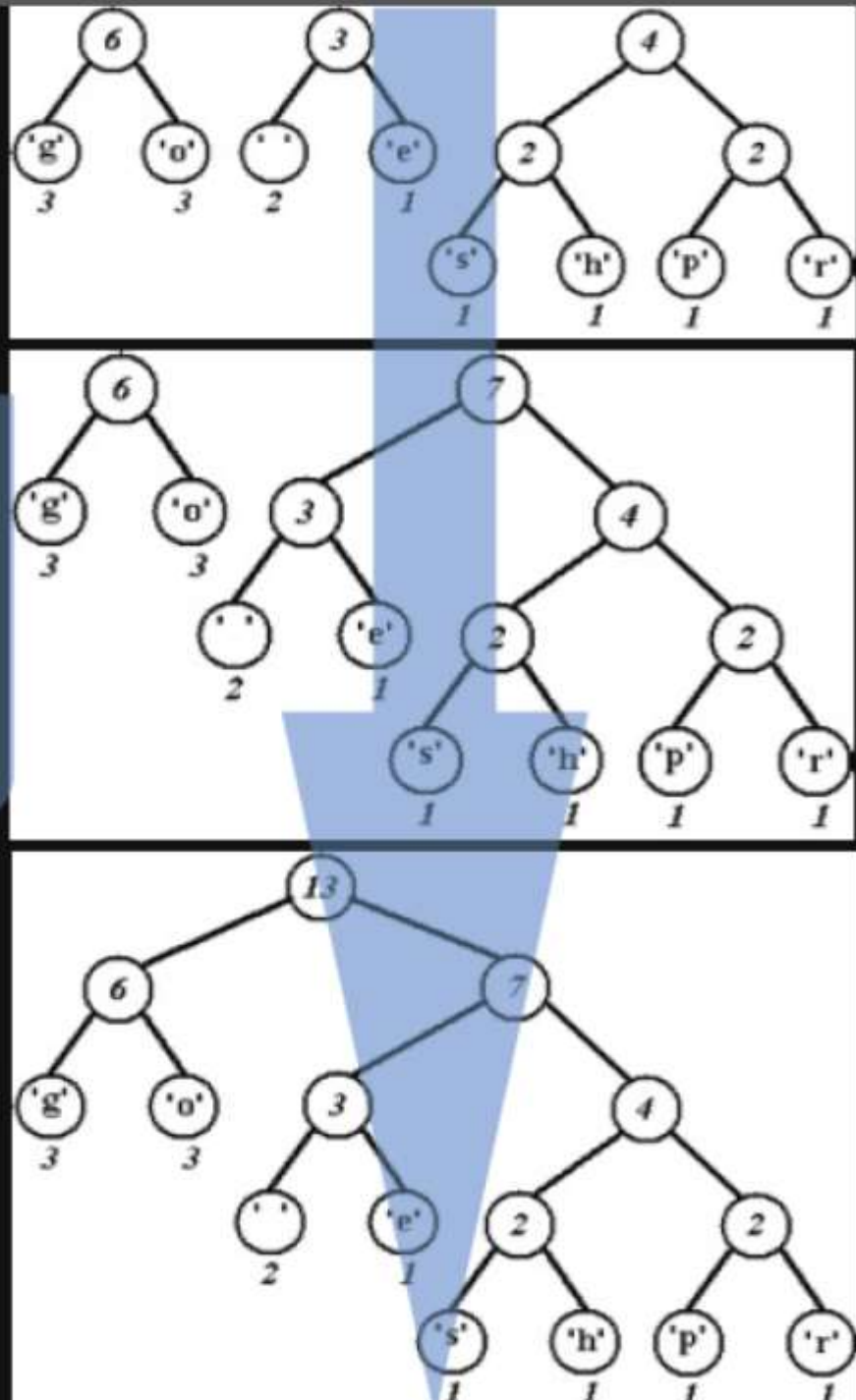
Step 4: Pick the smallest weighted tree and combine:



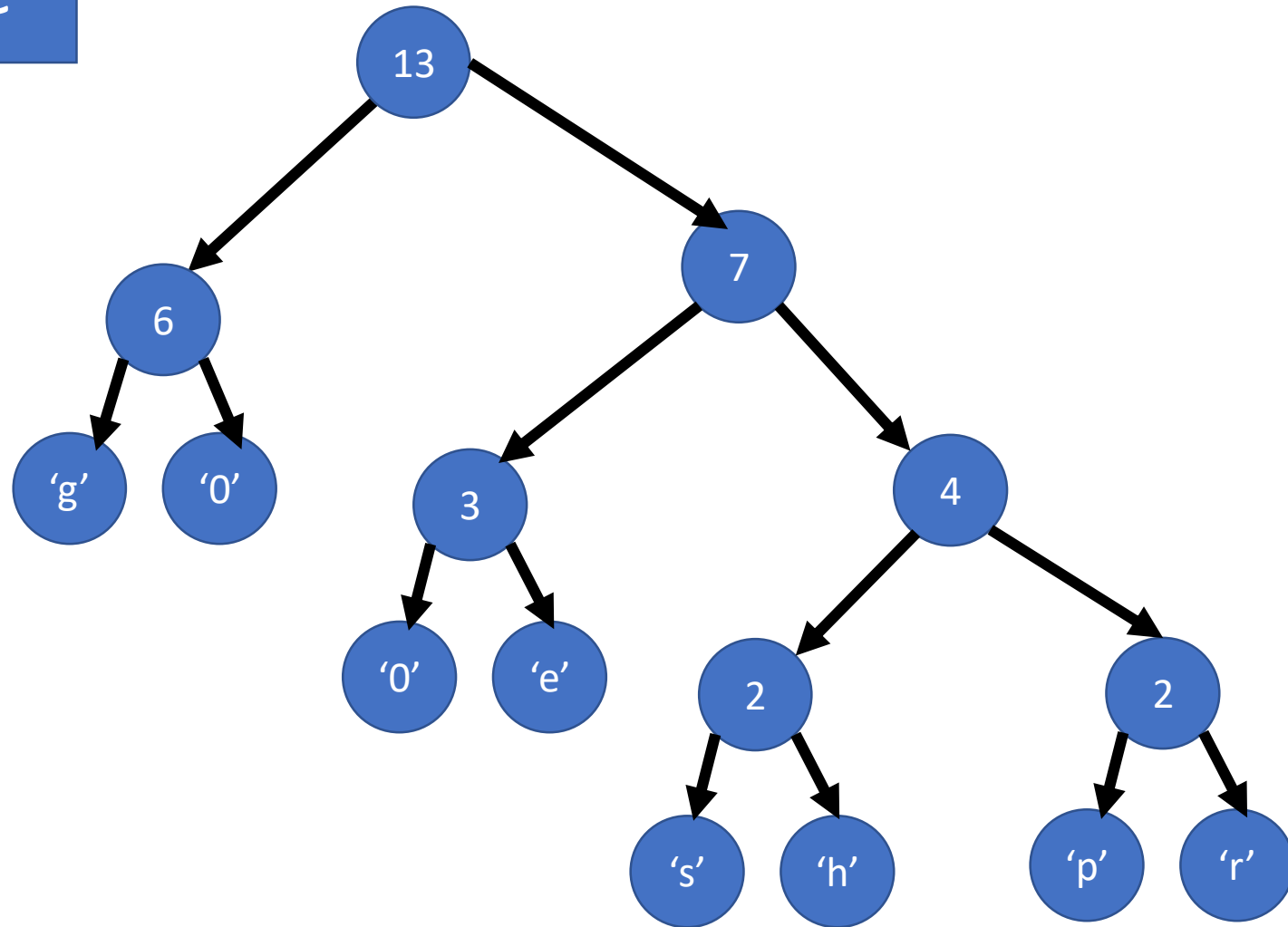
We'll then skip ahead to the final tree...



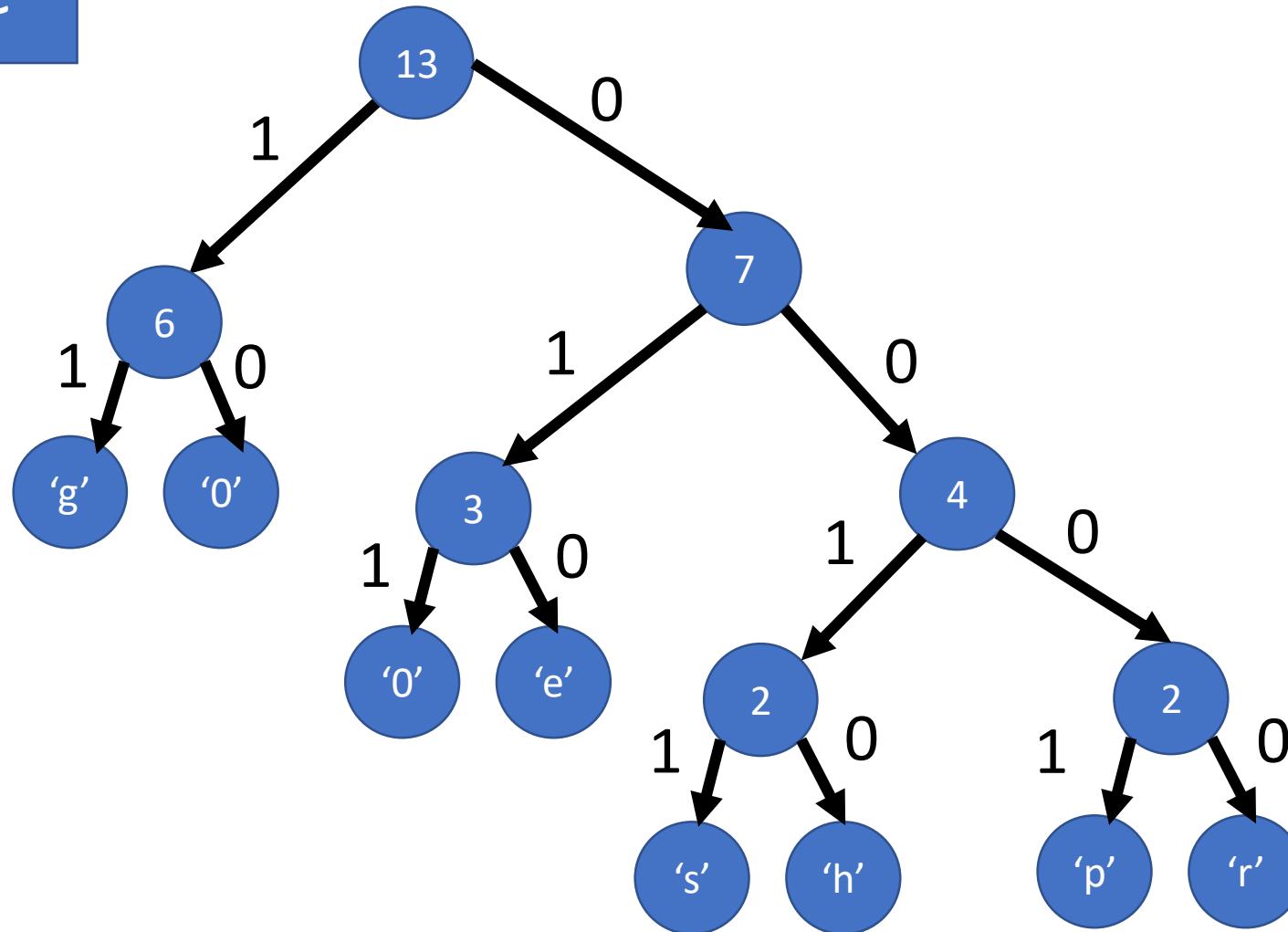
Intermediate Steps



Final Tree



Final Tree



Final Step: Assign 1 and 0 to each branch.

TRIES (OR PREFIX TREES)

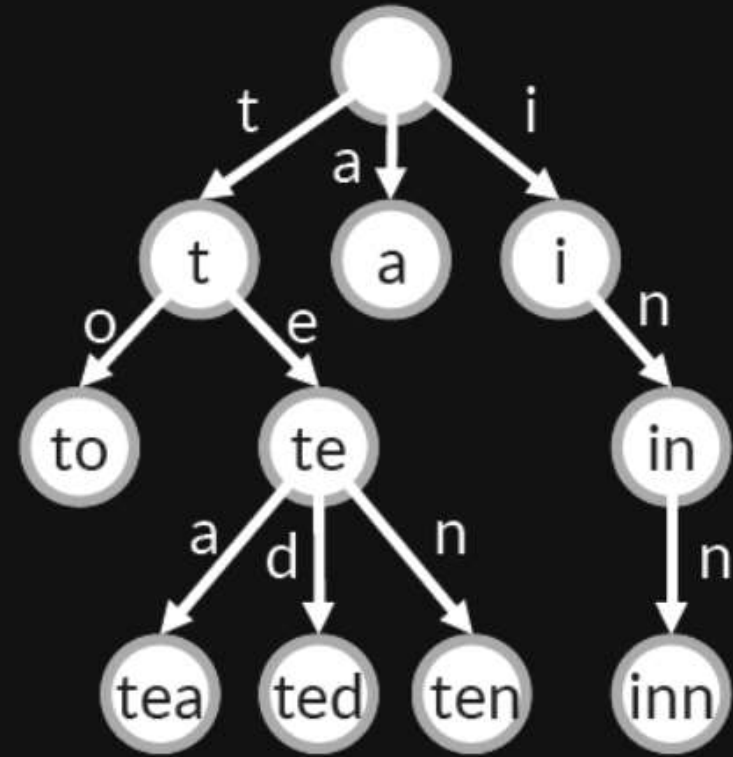
- We have focused on binary trees, where each node has 0, 1, or 2 children.
- In principle, you can operate with trees of various branching factors.
- As an example, we consider *tries*, also called *prefix trees*, a data structure often used to maintain dictionary entries of fixed length.

THE COST OF TRIE OPERATIONS

- Roughly, the time taken to search a trie (answer a ismember? query) or insert grows linearly with the length of the key.
- This can be much worse than a binary search tree!
- However, for large sets of data of roughly the same length (e.g., an English dictionary) this can be a good solution.
- Note that no “balancing” is required (cf. our discussion of search trees).

TRIES: EXAMPLES, AN INFORMAL DEFINITION

- Idea: Suppose set items are *words* over an alphabet.
 - English words are over the alphabet $\{a, b, c, \dots, z, A, B, \dots, Z\}$
 - Positive integers are words over the alphabet $\{0, 1, 2, \dots, 9\}$
- In a trie over the alphabet A , each node may have up to $|A|$ children.
- Words are “stored” at the end of the path they index.
- Thus, each leaf corresponds to a stored element.



A trie over the English alphabet containing A, to, tea, ted, ten, and inn.

IMPLEMENTING THE TRIE

- Imagine a trie for storing large numbers.
- One natural way to implement a node:

```
(trie0 trie1 ... trie9)
```

- Each node must store a (sub)trie for each numeral {0, 1, ..., 9}.

Any issues?

- Note that tries are often very *sparse*: a given node may often have only a few nonempty subtries.
- For this reason, we may choose the following implementation:

```
((0 . trie0) (1 . trie1) ... (9 . trie9))
```

where the subtries are left out if they are empty.

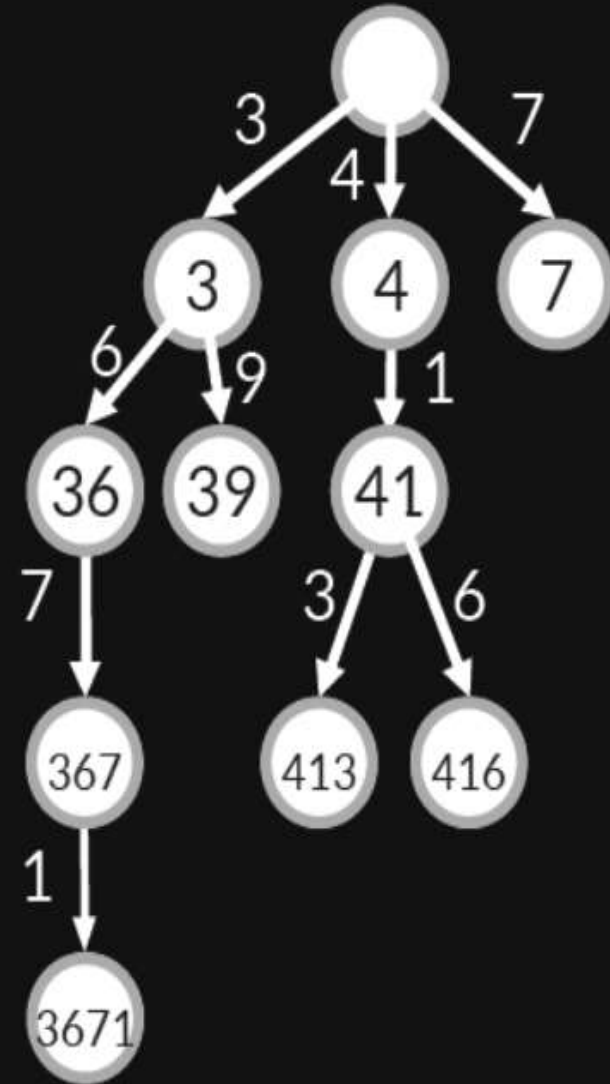
AN EXAMPLE WITH THIS IMPLEMENTATION

- Then, the top node:

$((3 \cdot t3) (4 \cdot t4) (7 \cdot t7))$

- Its leftmost child:

$((6 \cdot t6) (9 \cdot t9))$



Abstract Data Types: Buying a Dog



- Let's say you want to buy a dog. Almost all potential dog buyers assume the following:
 1. The dog can bark.
 2. The dog can walk.
 3. The dog can eat.
 4. The dog can be petted (very important!)

No dog buyer ever asks:
What should the functionality of my dog be?



Why don't they ask this important question?

No dog buyer ever asks: What should the functionality of my dog be?

- That is because the functionality of the dog is assumed!
- We assume that the dog can eat, walk and be petted because those are the built in dog functions we expect.
- In the field of computer science we have similar requirements for certain data structures.
- Without knowing the back-end or gritty details, we want certain data structures to have built in functionality.

ABSTRACT DATA TYPES

- Computer Scientists frequently organize their thinking about data types with the idea of an **Abstract Data Type**.
- An **abstract data type** is a high-level description of the functionality provided by a data type, without any reference to exactly how it is implemented.
- This “**abstraction layer**” hides implementation details: ***any implementation of an ADT can be “plugged in” to a computing infrastructure that requires it.***

What abstract data type have we seen already?

Lists

Binary
Trees

Tries

How are these data types related?

THE SET ADT

- Recall our discussion of the Set ADT.
- A Set datatype must offer two functions: $\text{insert}(x, S)$ and $\text{ismember}(x, S)$.
 - (It also provides a distinguished object called `emptyset`).
 - To fully describe the ADT, we describe the functions it provides and, more importantly, the *semantics* that these functions offer.
 - For Set, this is easy: $\text{ismember}(x, S)$ returns `True` if $\text{insert}(x, S)$ has ever been called, and `False` otherwise.
- Note that we have defined the behavior of these operations without specifying the implementation.

A RICHER SET ADT

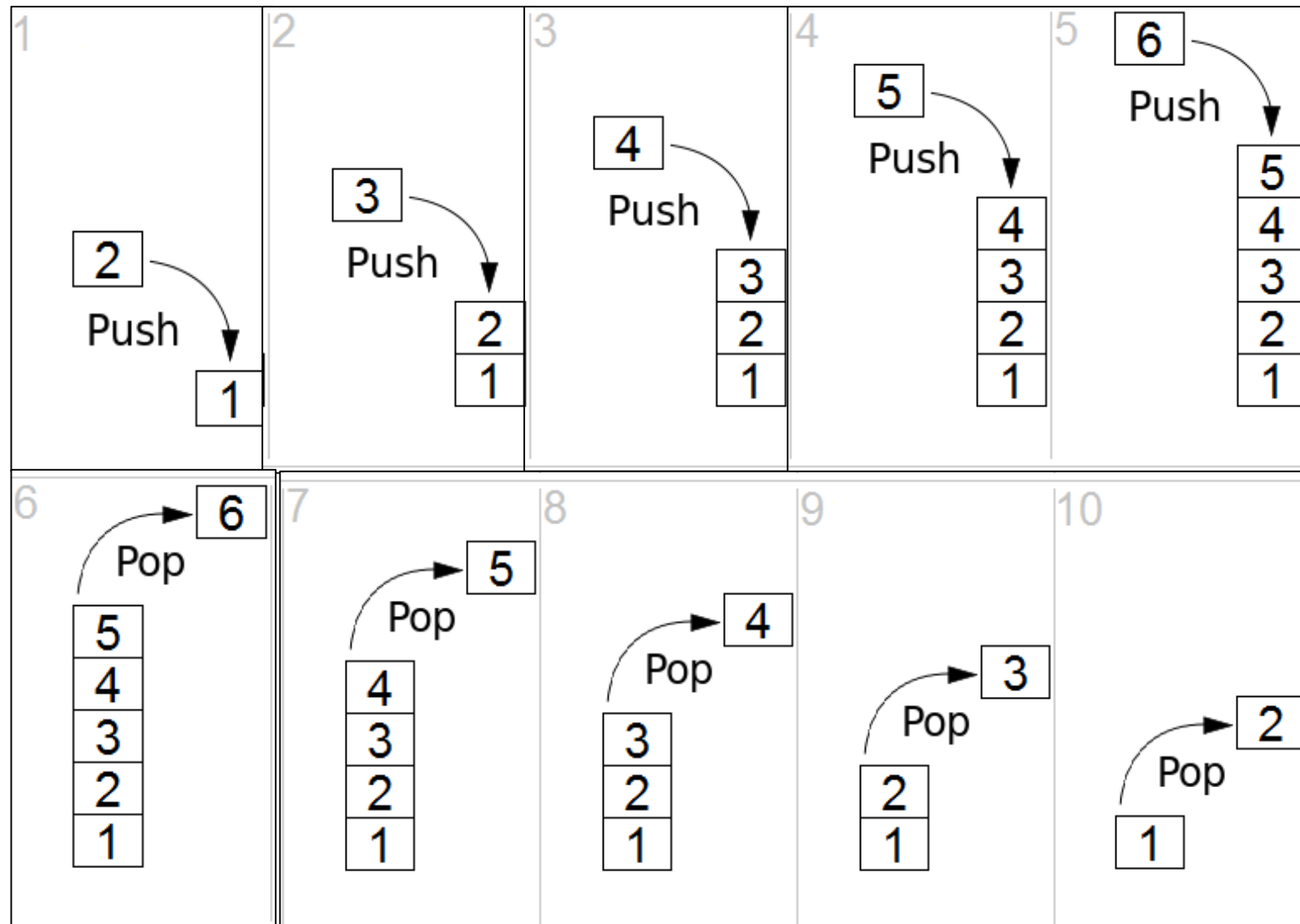
- We've been discussing a very basic notion of the Set ADT.
- You could ask for much more:
 - $\text{Union}(S_1, S_2)$
 - $\text{Intersection}(S_1, S_2)$
 - $\text{Delete}(x, S_1)$
- Producing an efficient implementation of such a rich set ADT is a deep mathematical issue: it was a longstanding research problem with fairly sweeping successes in the late 1990s.

Stacks

- A container of objects, similar to a stack of coins or Pez dispenser.
- Objects can be inserted at any time.
- Only the top (last placed object) can be removed. Last-in-first out (LIFO).
- Pushing – An object is added to the top of the stack.
- Popping – Removing the top object (the last one added) from the stack.



Simple Pushing and Popping Example



Stack Code in Python

```
1 class Stack:
2     def __init__(self):
3         #Keep track of the top of the stack
4         self.topIndex = -1
5         #Python list to hold the data
6         self.data = []
7
8     #Method to add an object to the stack
9     def push(self, object):
10        self.topIndex = self.topIndex + 1
11        self.data.append(object)
12
13    #Method to remove an object from the stack
14    def pop(self):
15        #Make sure that the stack is not empty
16        if self.isEmpty() == True:
17            return None
18        #Get the top object on the stack
19        topObject = self.data[self.topIndex]
20        self.data.pop(self.topIndex)
21        #Reduce the index of the stack by 1
22        self.topIndex = self.topIndex - 1
23        #Return the top object
24        return topObject
25
26    #Check if the stack is empty
27    def isEmpty(self):
28        if self.topIndex < 0:
29            return True
30        else:
```

```
32    #Make the stack
33    s1 = Stack()
34    #Add three elements to the stack
35    s1.push('a')
36    s1.push('b')
37    s1.push('c')
38
39    #Remove three elements from the stack
40    print(s1.pop())
41    print(s1.pop())
42    print(s1.pop())
43
44    #Now check if the stack is empty
45    if s1.isEmpty() == True:
46        print("The stack is now empty.")
```

Code Output:

C:\WINDOWS\system32\cmd.exe

```
c
b
a
The stack is now empty.
Press any key to continue . . .
```


IMPLEMENTING A STACK IN SCHEME

Stacks can (of course) be naturally implemented in terms of lists:

```
(define (push x S) (cons x S))
```

```
(define (top S) (car S))
```

```
(define (pop S) (cdr S))
```

```
(define (empty? S) (if (null? S) #t #f))
```

- This implementation seems hard to beat:
 - any of these operations requires a *fixed number* of procedure calls.

FRONT

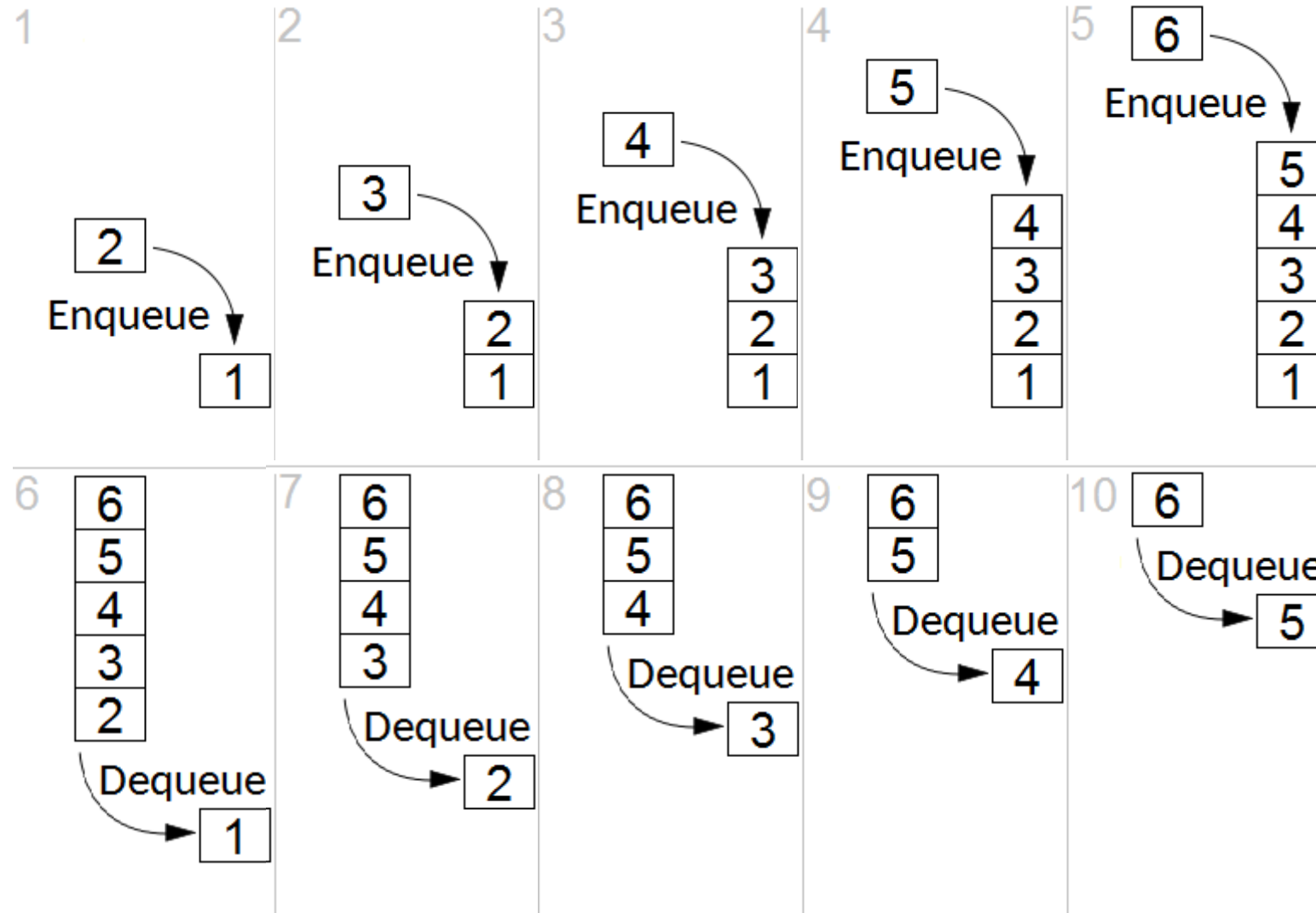
Queues

BACK



- A container of objects, similar to a line in a grocery store.
- Elements are inserted at the back and removed at the front. E.g. If you are in line first, you get served first.
 - FIFO- First in, first out.
 - Enqueue - Insert an element at the back.
 - Dequeue - Remove the element at the front.

Simple Enqueue and Dequeue Example



IMPLEMENTING THE QUEUE ADT IN SCHEME

- We can naturally implement the queue ADT as a list in Scheme.

```
(define (enqueue x Q)
  (if (null? Q)
      (list x)
      (cons (car Q)
            (enqueue x (cdr Q)))))

(define (front Q) (car Q))

(define (empty? Q) (null? Q))

(define (dequeue Q) (cdr Q))
```

Figure Sources

- <https://pbs.twimg.com/media/DoZ5YBVXsAEVAwg.jpg>
- <https://hips.hearstapps.com/hmg-prod.s3.amazonaws.com/images/funny-dog-captions-1563456605.jpg>
- <https://dogecoin.com/assets/img/doge.png>