CS5112: Algorithms and Data Structures for Applications

Lecture 8: Bits

Richard Bowen

Cornell University

CORNELL TECH

Administrivia

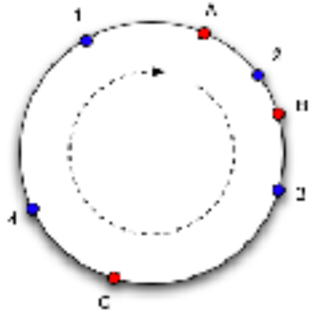
- Schedule this week:
  - Today's lecture by Richard Bowen (me!)
  - Thursday lecture by Prof. Ari Juels
  - Thursday evening clinic by Richard Bowen (also me!)

Today

- Finish up Chord Algorithm and Skip Lists
- “Bits”
  - Measuring information
  - Compression
  - Huffman Coding

Chord Algorithm

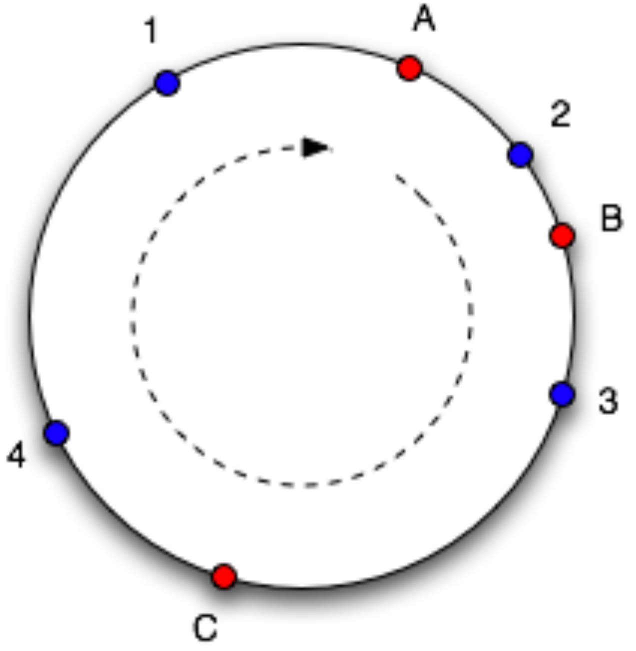
- Reminder of setting: distributed hash table, consistent hashing
- Each node in network gets an id
  - E.g., m-bit hash of its ip address, …
- We imagine the 2^m numbers from our hash function in a circle
- “successor” of any m-bit number = next node around the circle



Consistent hashing: lookup

- Some node wants to lookup a key. Needs to find successor(hash(key)).

- One option: every node stores a full list of nodes
  - Expensive updates for adding/removing node
- Another option: every node knows its own successor
  - Cheap updates, O(n) lookup.



Strategy comparison

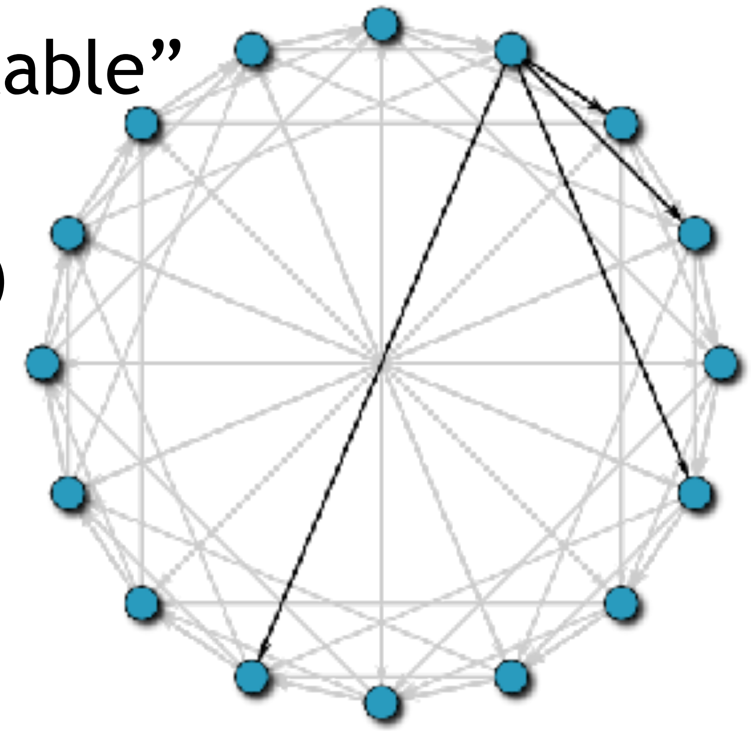
- Every node knows every other node:
  - O(n) nodes updated on every node insertion
  - O(n^2) storage
  - O(1) hops to discover key-value location
- Every node knows its own successor
  - O(1) nodes updated per node insertion
  - O(n) storage
  - O(n) hops to discover key-value location

Chord algorithm

- An intermediate tradeoff: store a “finger table”
- Each node n stores the address of

$$successor(n + 2^i \bmod 2^m)$$

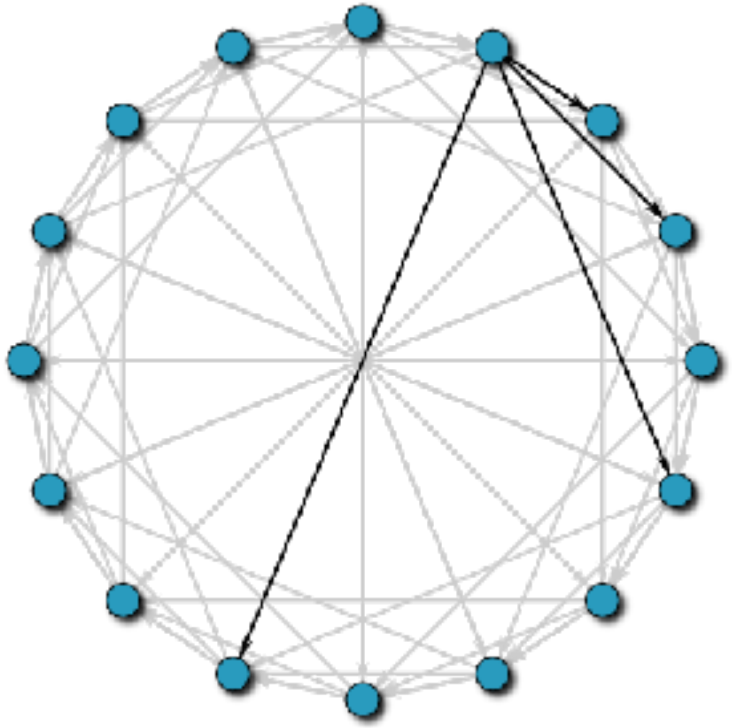
- for each 0 <= i < m.



» Figure by Seth Terashima (Tetra7 (talk)) - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10089321

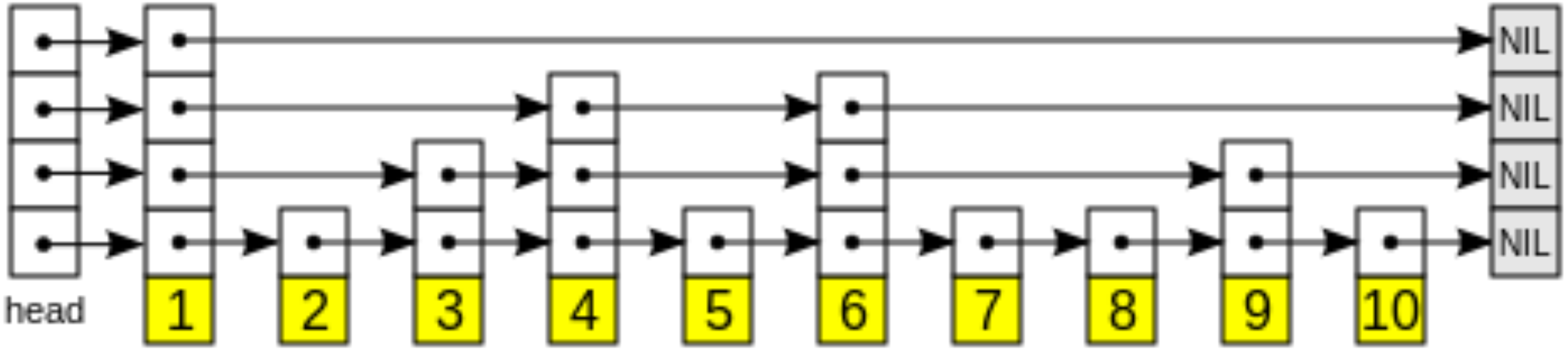
Chord: algorithm

- Storage now just O(nm).
- Lookup is logarithmic:
  - At each step you go about half the remaining distance to the correct node.
- Inserting a node touches O(log n) other nodes as well (glossing details)



Skip lists

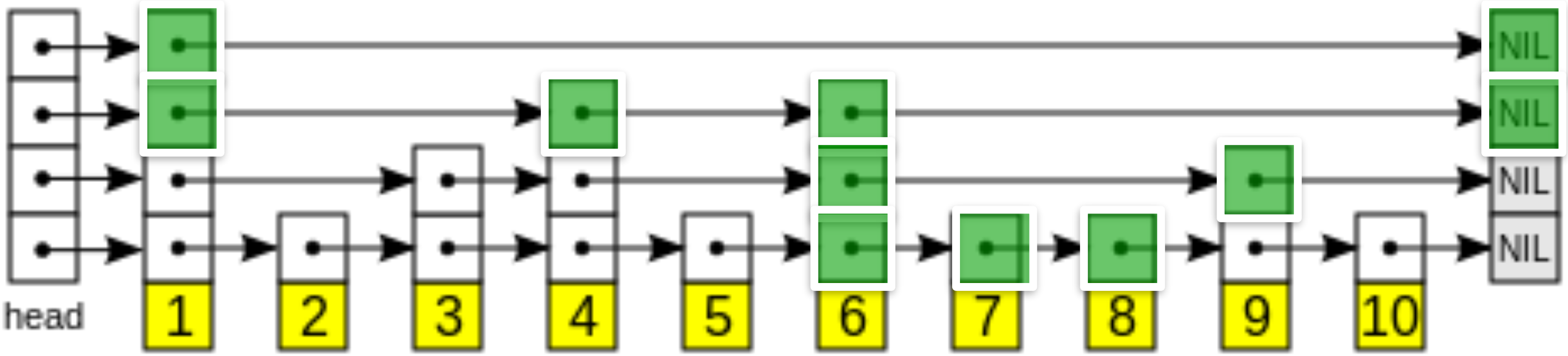
- Can we find an element in a sorted linked list, quickly?
  - Similar to Chord lookup
  - Hierarchy of ‘express lanes’, randomly generated
  - Each node has a small number of “next” pointers instead of 1



- Figure by Wojciech Muła - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=4871915

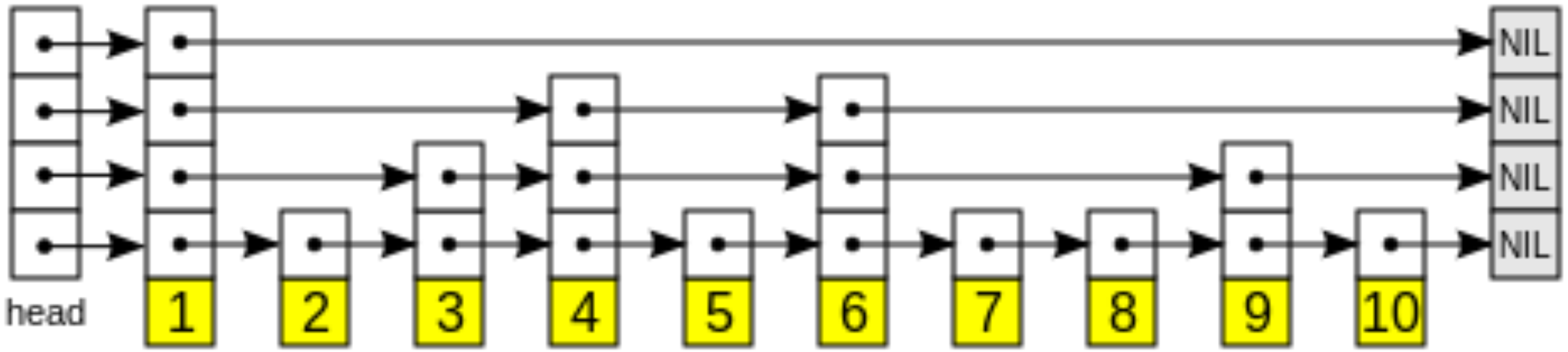
Skip Lists: lookup

- Start in "fastest" lane
- Walk along until *next step* would be too big (or off the end)
  - Then drop down to next slowest lane
- Example: looking up 8



Skip Lists: “fast lane” design

- To get log(n) time lookup, each step should take you about halfway to the goal
  - Can achieve this by randomly selecting heights of nodes



Set-of-items data structures

	Insert time	Lookup time
Linked List (unsorted)	O(1)	O(n)
Array (sorted)	O(n)	O(log n)
Skip list	O(log n)	O(log n)

Bits

- What is a "bit"?
  - Abstractly: a variable with one of two values
  - Physically: a device that can store one of two values
  - As a unit, a measure of information:
    - Length — meters
    - Time — seconds
    - Information — bits

Bits as measure of randomness

- I want to randomly choose from 16 people, with a fair, 4-sided die. What should I do?
- Just roll it twice; 16 equally-likely outcomes.

Image Credit: Clément Bucco-Lechat, https://commons.wikimedia.org/wiki/File:4-sided_die.jpg

Unfair dice

- What about an unfair 4-sided die?
- Same strategy is unfair: AA has probability 1/4, not 1/16.

Outcome	Prob
A	1/2
B	1/4
C	1/8
D	1/8

Unfair dice

- Strategy: roll several times. Write down the code shown on the right. Repeat.
- Once there are at least 4 bits written down, stop. 1st 4 bits = person to choose.
- Example: ABC -> 010110 -> person 5.

	Prob.	Code
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

Is this fair?

- Every bit is 0 with probability 1/2. Why?
- 2 cases:
  - The bit starts a new code? 0 only if the roll was A.
  - The bit is in the middle of a code? Still true (check!)

	Prob.	Code
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

How long will this take?

- (Fair die always takes 2 rolls)
- Unfair die:
  - Never takes 1 roll
  - Takes 2 rolls sometimes: BB, BC, DB,…
  - Takes >=3 rolls otherwise
- Takes longer!
- Therefore we say that this gives us fewer bits of randomness than a fair die.
- Have to keep going until you have 4 bits!

	Prob.	Code
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

CORNELL TECH

Communication

- I roll my fair 4-sided die 10k times and transmit the results to you
  - Best I can do is encode with 2 bits per roll
  - Takes 20000 bits every time

Communication

- How about the code for the unfair die I used previously?
- Length of stream is now random, but on average it is:

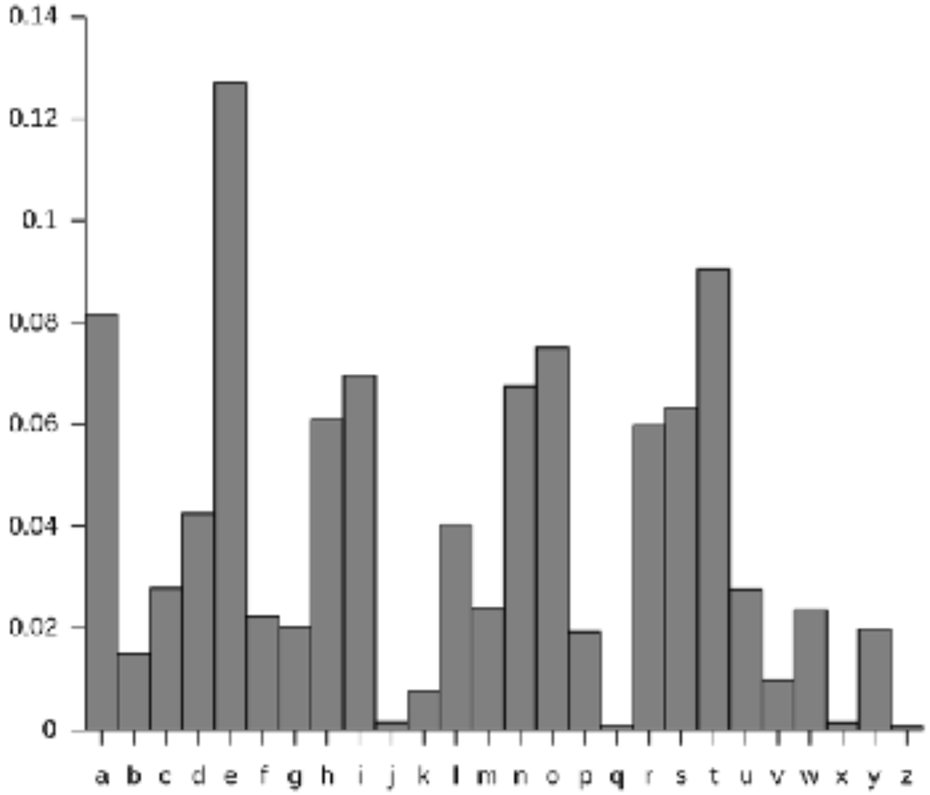
$$10000 \bullet (\tfrac{1}{2} \cdot 1 + \tfrac{1}{4} \cdot 2 + \tfrac{1}{8} \cdot 3 + \tfrac{1}{8} \cdot 3)$$

- On average, only 17500 bits to transmit this.
- We say: “1.75 bits of randomness” per roll.

	Prob.	Code
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

Concrete Example: Compressing/Encoding Text

- English text: 26 letters (+ space).
- Not very uniform! E, T much more likely than Z, X.
- Looks like an unfair die.



https://en.wikipedia.org/wiki/File:English_letter_frequency_(alphabetic).svg

Concrete Example: English Text

- Naively: could just use 5 bits per character.
- Let’s use a *variable-length* encoding instead
  - Intuition: use shorter codes for more common letters to reduce the average length

Concrete Example: English Text

- Is this a good code? (the rest of the alphabet not shown)
  - Common letters (E, A) get short codes! Uncommon get long codes…
- What is “0111”? AB or EZ?

Letter	Code
A	“01”
B	“11”
E	“0"
Z	“111”

Prefix-free codes

- Need a code f() so that no two letters X,Y have f(X) is a prefix of f(Y)
- Guarantees no ambiguity (why?)

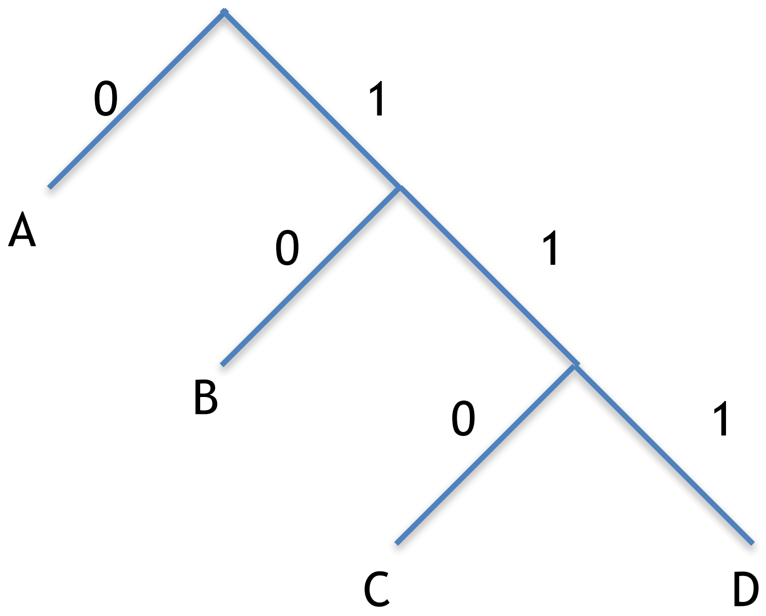
Letter	Code
A	“01”
B	“11”
E	“0"
Z	“111”

Prefix-free codes

- Back to my unfair die:
  - Is this code prefix-free?
  - Decode this bitstring:
    - 011010111
    - ACBD

	Prob.	Code
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

Prefix-free codes = binary trees

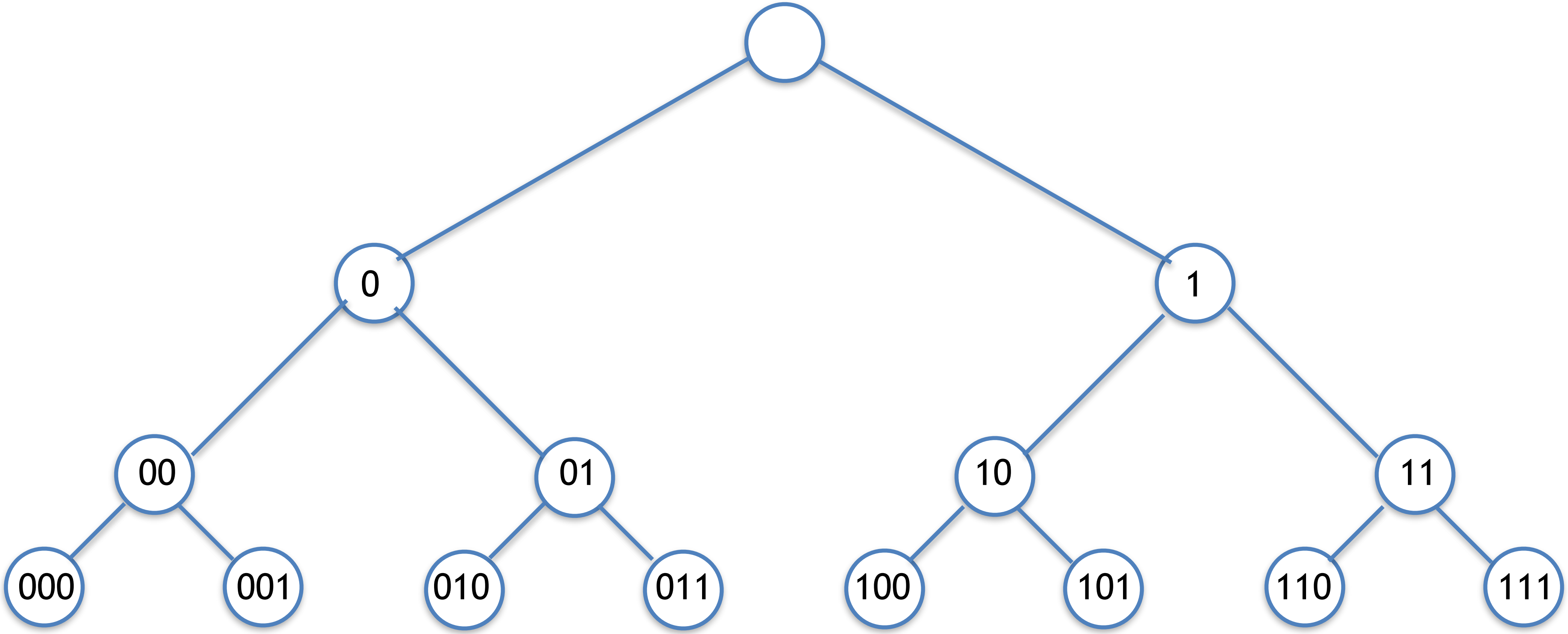


- 011010111 -> ACBD

	Prob.	Code
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

Prefix-free codes = binary trees

- A little more on why ancestors = prefixes



CORNELL TECH

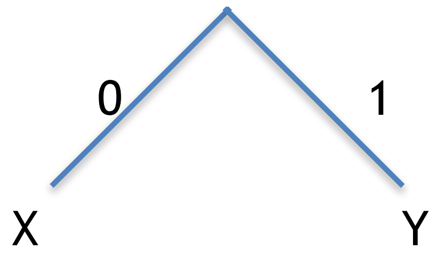
Constructing prefix-free codes

- Let's compute a prefix-free code for these letter frequencies
- We'll use a *Huffman Encoding* (1952, David Huffman)

Letter	Prob.
A	5/16
B	3/16
C	3/16
D	3/16
E	2/16

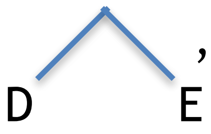
Huffman coding algorithm

1. Find the lowest-2 probability symbols, X and Y, with probs P(X) and P(Y)
2. Combine them into a tree like this:
3. Treat this tree as a single symbol with prob P(X)+P(Y)
4. Repeat

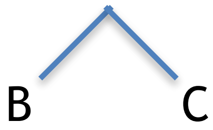


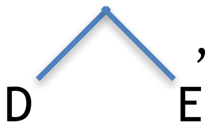
Huffman coding example

A, 5/16 B, 3/16 C, 3/16 D, 3/16 E, 2/16

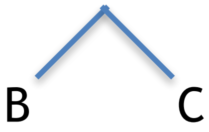
A, 5/16 B, 3/16 C, 3/16 

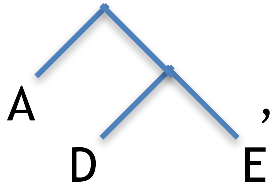
,5/16

A, 5/16 

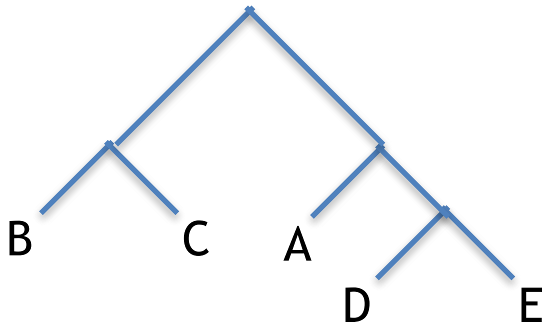
,6/16 

,5/16

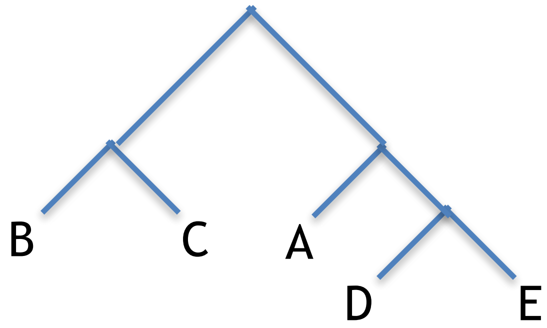


,6/16 

,10/16



Huffman coding example



Letter	Prob.	Code
A	5/16	“10”
B	3/16	“00”
C	3/16	“01”
D	3/16	“110”
E	2/16	“111"

Huffman coding algorithm

- Intuition: highly-probably things get added later -> shorter codes.
- The code is “optimal” in some sense
- On English, gets down to ~4.1 bits per letter
    - Can do much better by considering the 27*27 *pairs* of letters as symbols
    - Can do even better… stay tuned for when we talk about *autoencoders*

Bits

- Measuring information in bits in this way is essential to how we think about computer science.
  - Encryption schemes whose outputs look random ("1 bit per bit") have strong guarantees ("one-time pad" is unbreakable!)
  - Compression algorithms aim to have "1 bit per bit" as well — otherwise, you can compress further
  - Used in theory as well, for example, sorting algorithm time complexity lower bounds