CS 181AG
Lecture 8

Intro to Prefix Lookup

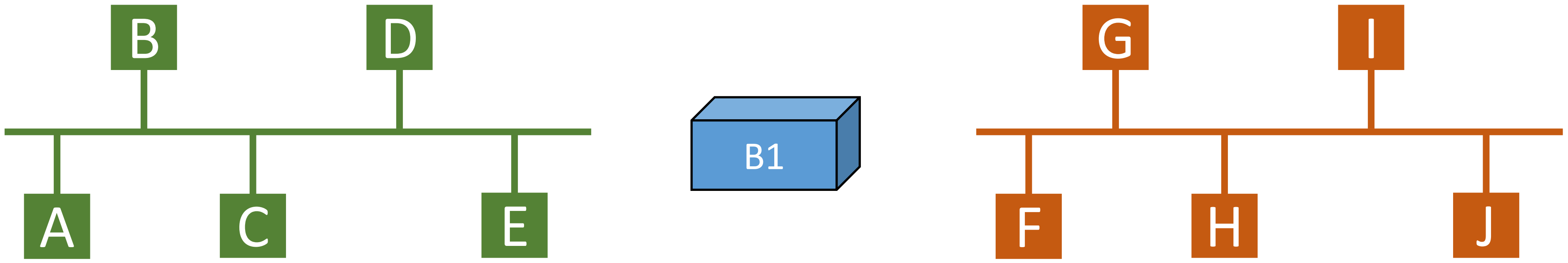
Arthi Padmanabhan

Sep 26, 2022

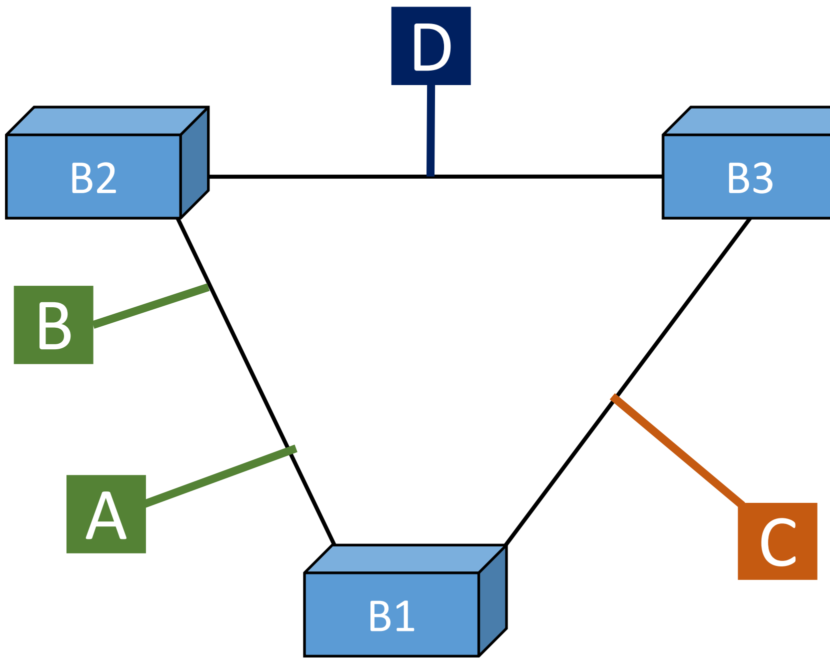
Recap

- Putting it together:
  - We first learned about local networks (expanded by bridges)

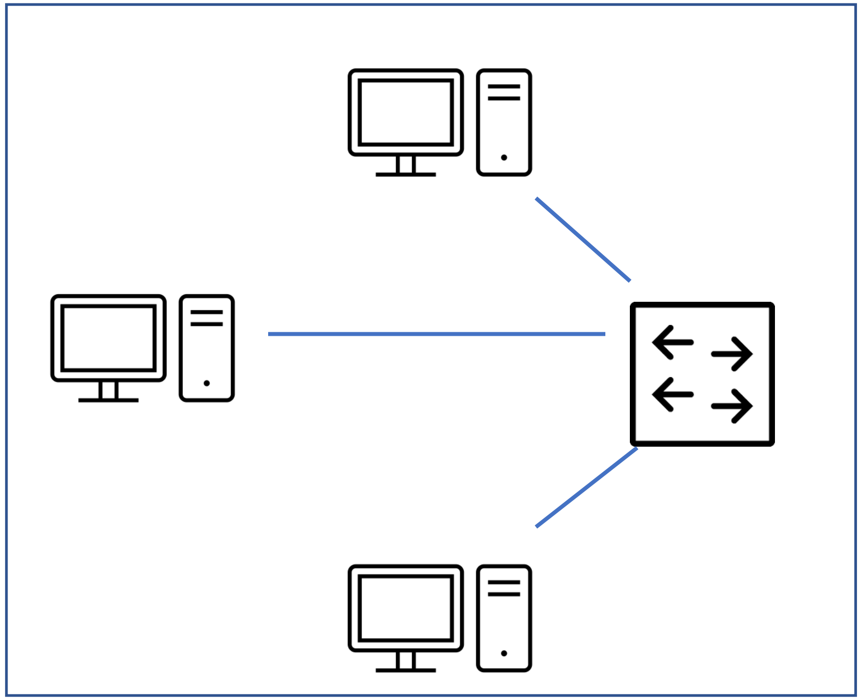
Putting It Together



Putting It Together



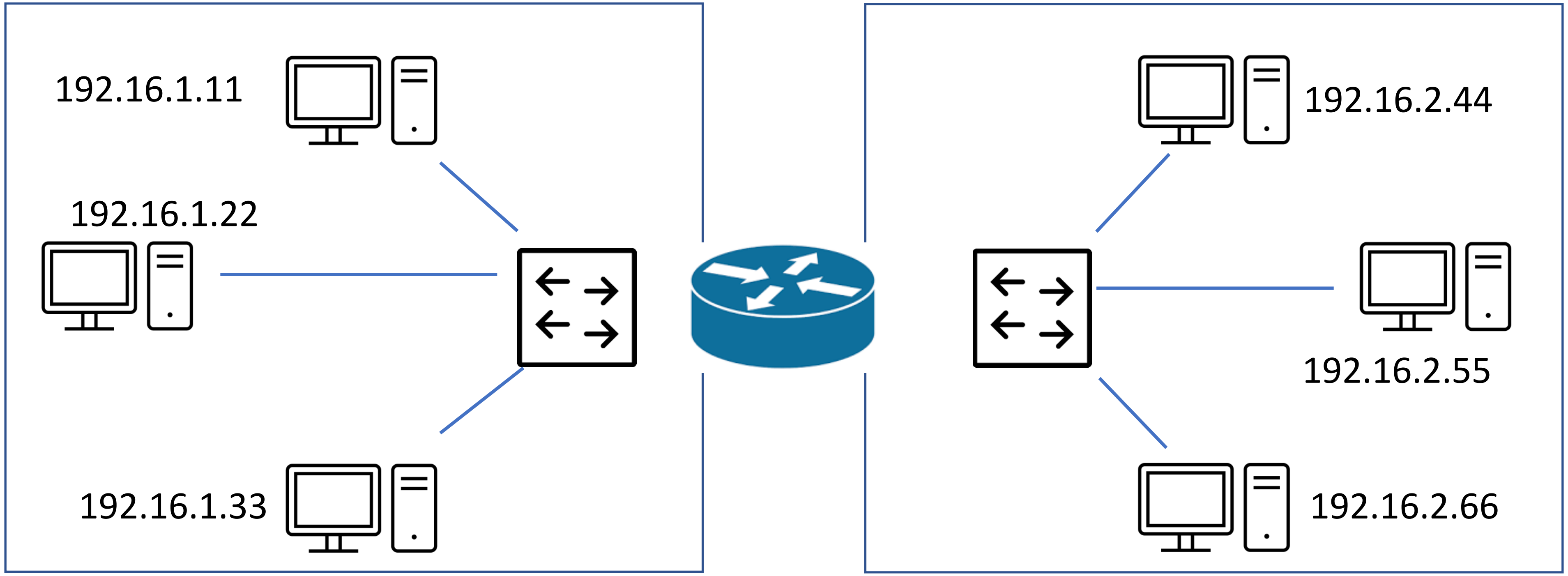
Putting It Together



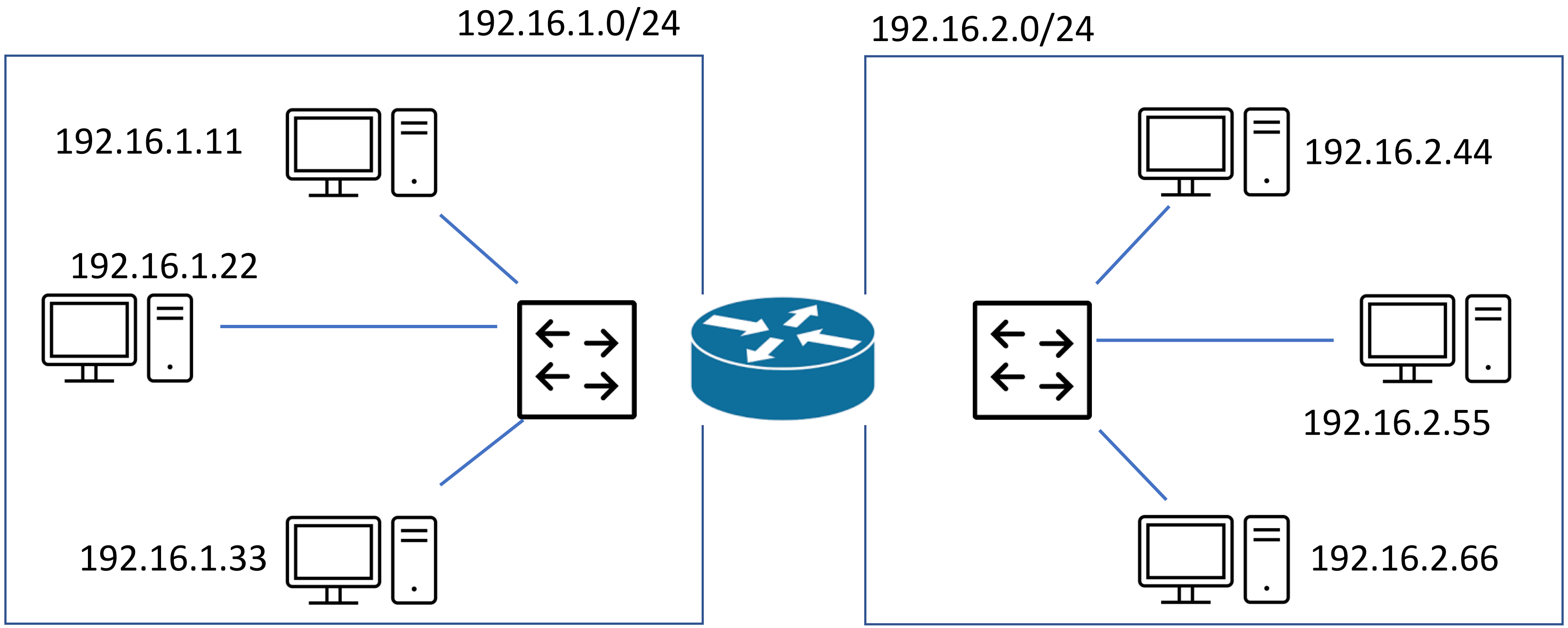
Recap

- Putting it together:
  - We first learned about local networks (expanded by bridges)
  - To connect different local networks, we need **routers**, which forward based on **IP addresses**

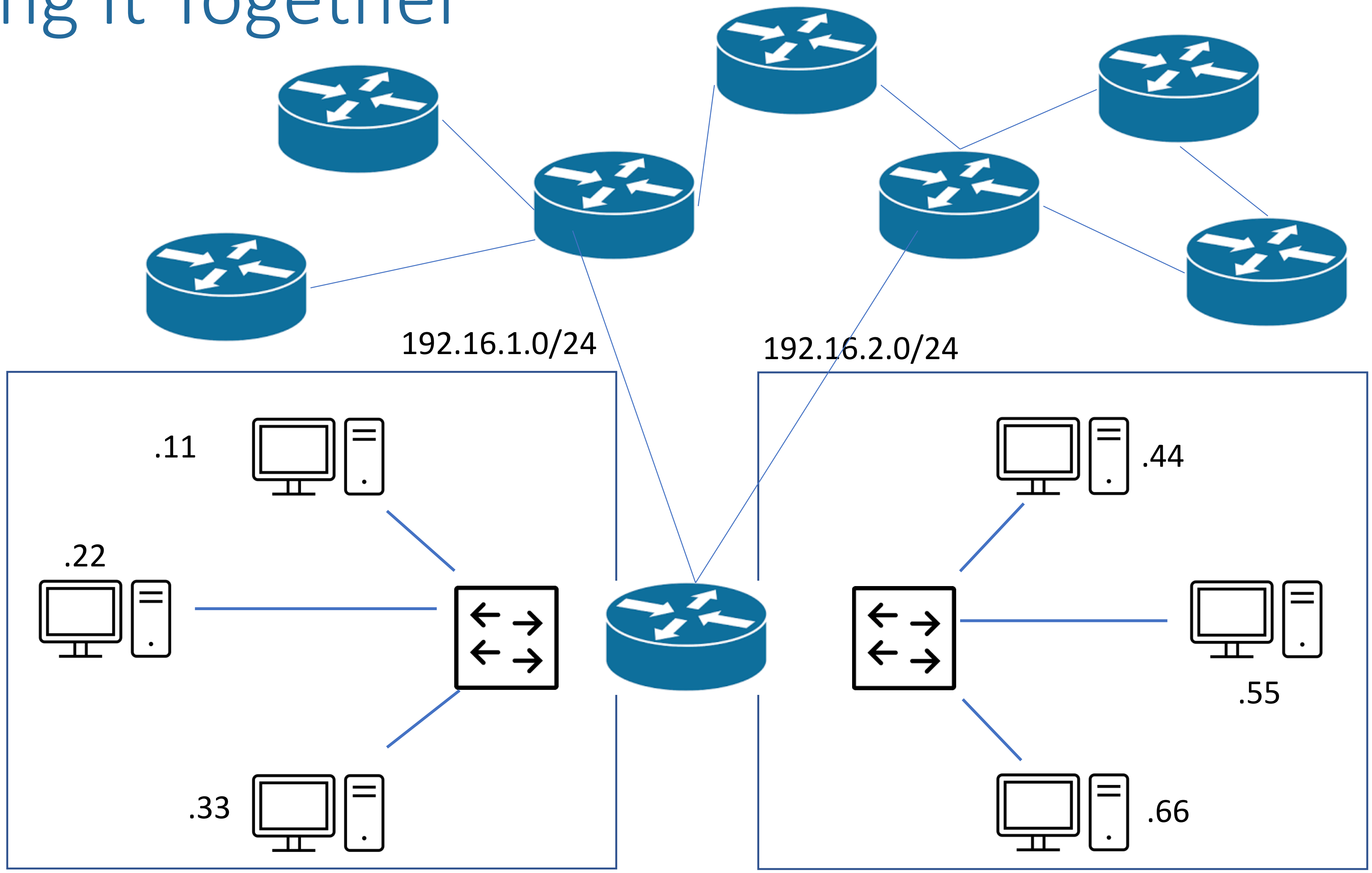
Putting It Together



Putting It Together



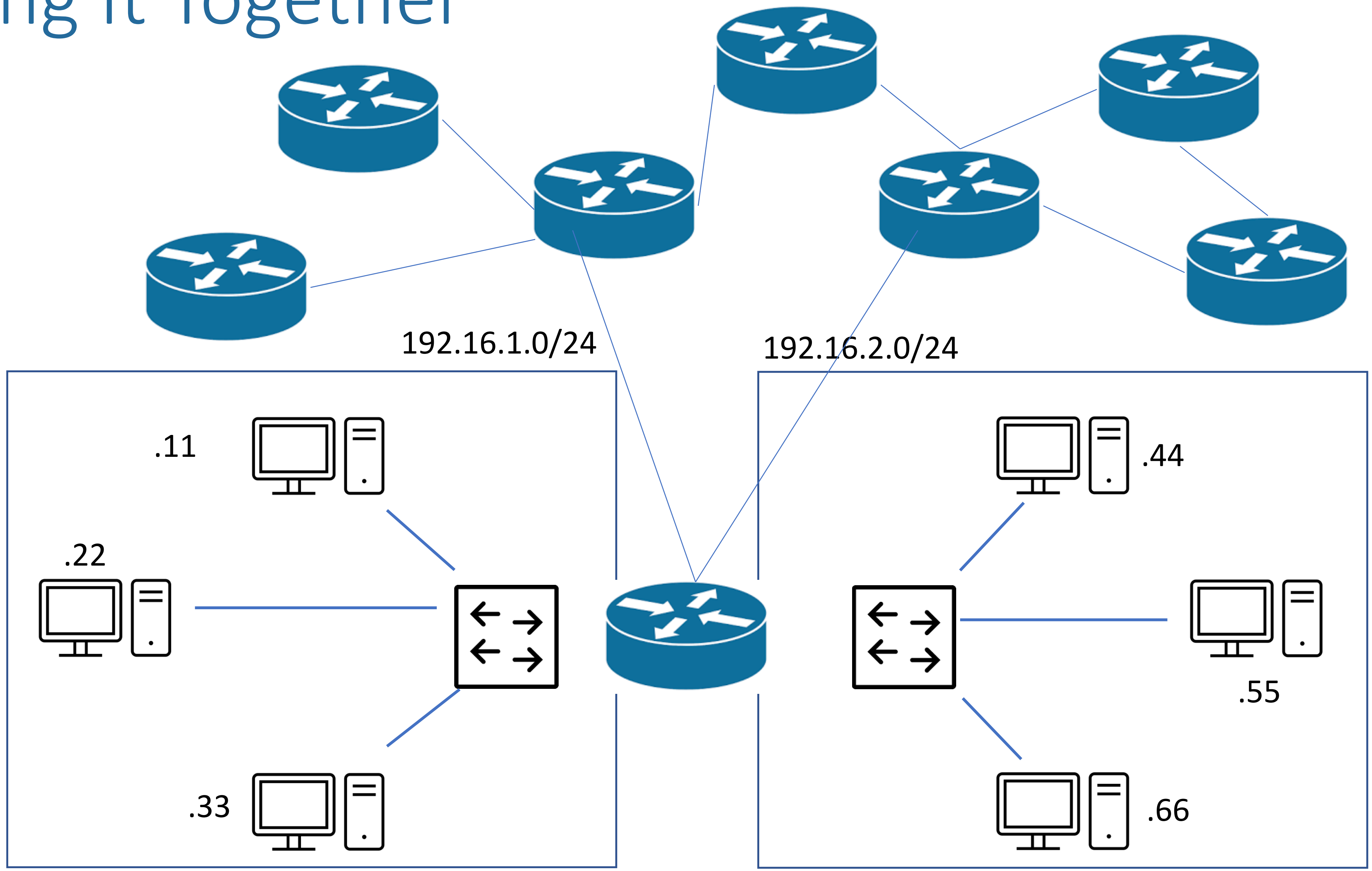
Putting It Together



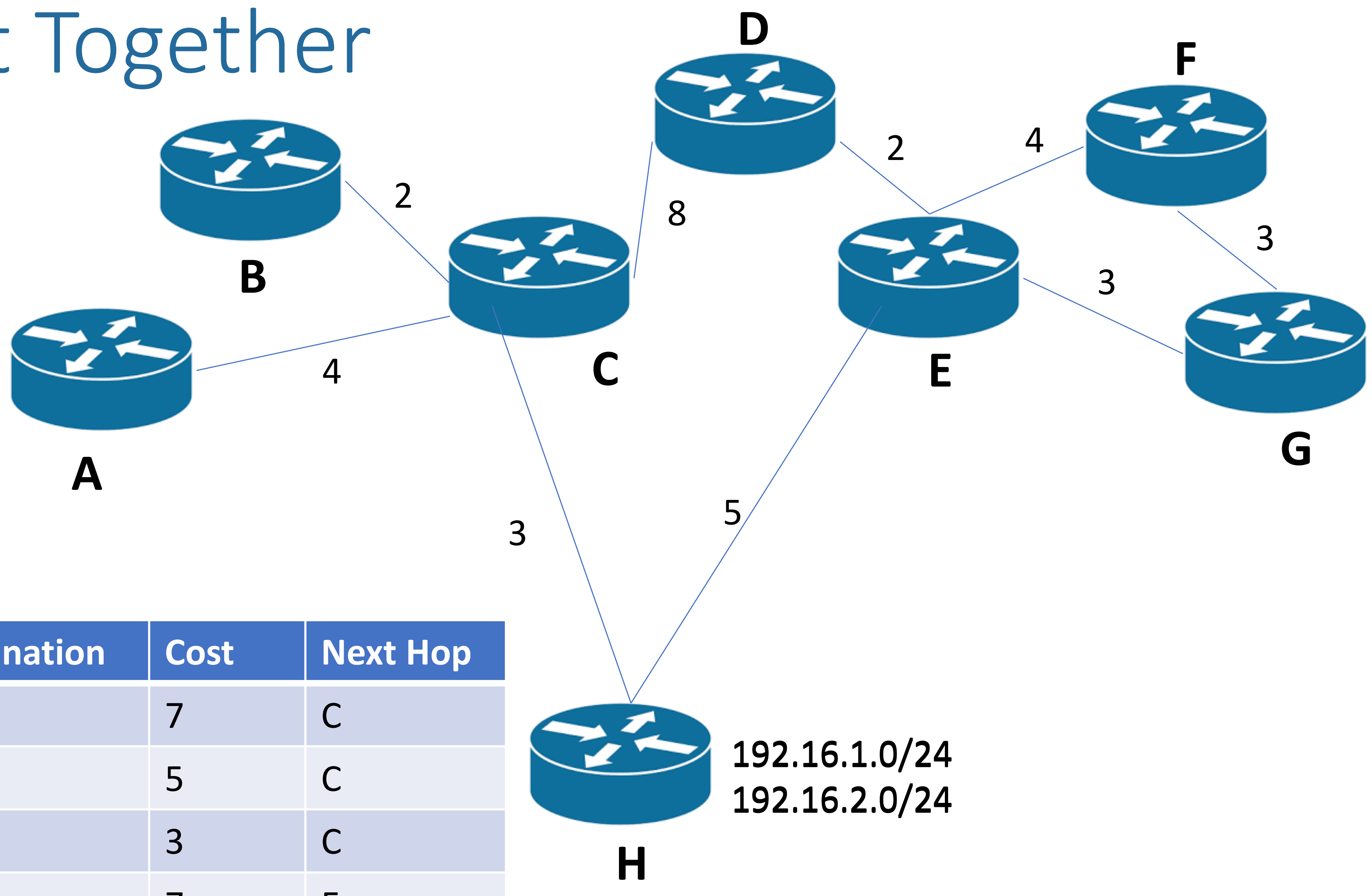
Recap

- Putting it together:
  - We first learned about local networks (expanded by bridges)
  - To connect different local networks, we need **routers**, which forward based on **IP addresses**
  - Routing Protocols determine how to populate the Forwarding Information Base (FIB)

Putting It Together



Putting It Together



Destination	Cost	Next Hop
A	7	C
B	5	C
C	3	C
D	7	E
…		

Recap

- Putting it together:
  - We first learned about local networks (expanded by bridges)
  - To connect different local networks, we need **routers**, which forward based on **IP addresses**
  - Routing Protocols determine how to populate the Forwarding Information Base (FIB)
  - We look up the FIB by finding the **longest matching prefix**

Putting It Together

Destination	Next Hop
192.168.74.0/24	Router 1
192.168.74.192/28	Router 2
192.168.74.204/30	Router 3
10.1.120.0/21	Router 4
0.0.0.0/0	Router 5

1. 192.168.74.198
2. 192.168.74.207
3. 10.1.128.12
4. 192.168.74.208
5. 10.1.125.74
6. 192.168.73.0

Recap

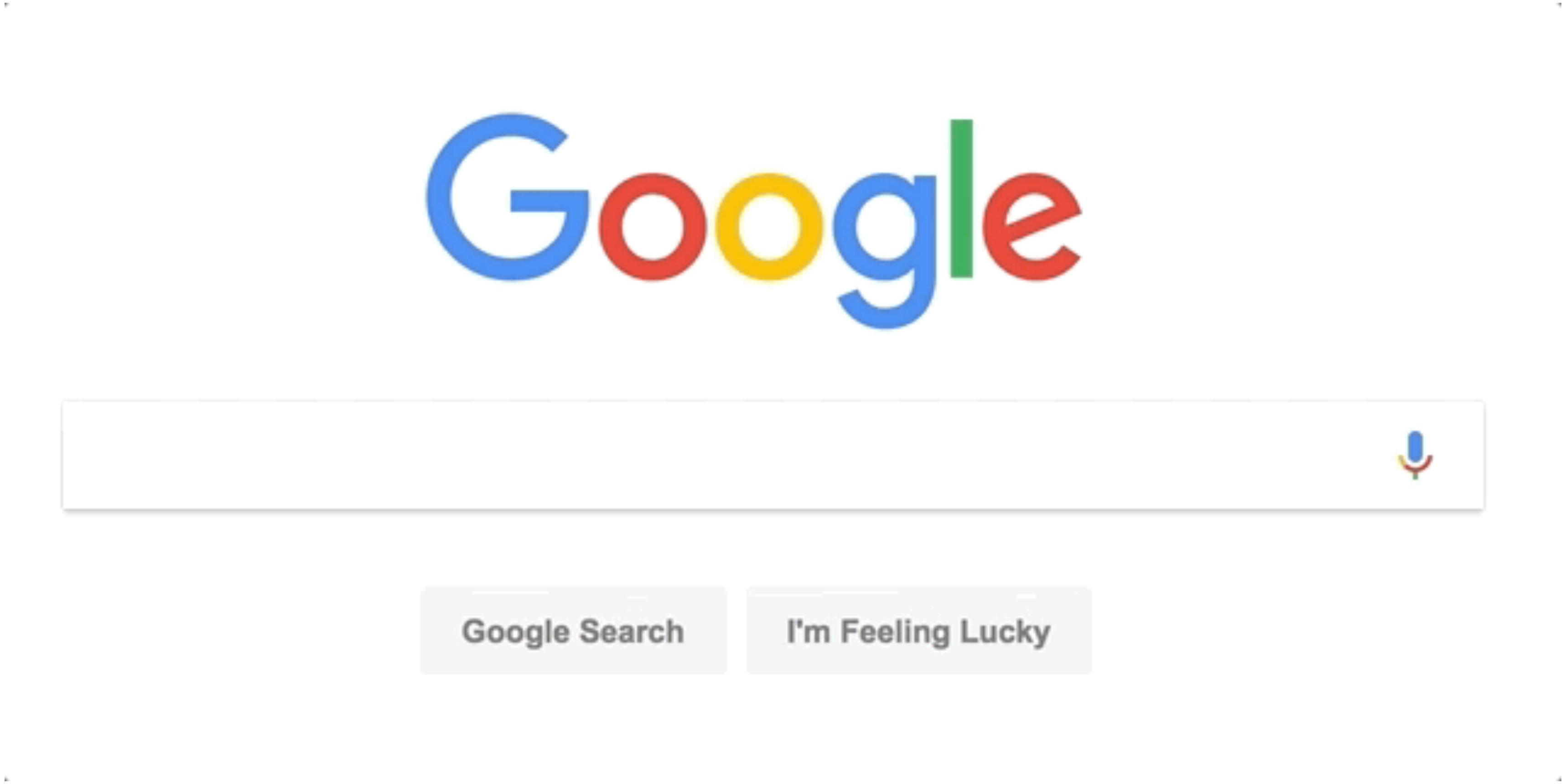
- Putting it together:
  - We first learned about local networks (expanded by bridges)
  - To connect different local networks, we need **routers**, which forward based on **IP addresses**
  - Routing Protocols determine how to populate the Forwarding Information Base (FIB)
  - We look up the FIB by finding the **longest matching prefix**
  - Prefix Lookup Problem explores how to quickly look up the FIB so we can forward packets without creating a bottleneck -> topic for this week!

Prefix Lookup Goals

1. Reduce lookup speed (number of memory accesses)
    - To run at wire speed, we need each packet to take:
        - 320 nsec at 1 Gbps
        - 32 nsec at 10 Gbps
        - 8 nsec at 40 Gbps
2. Lower storage in memory
    - Lower storage lowers each individual memory access time
3. Lower time to update prefixes

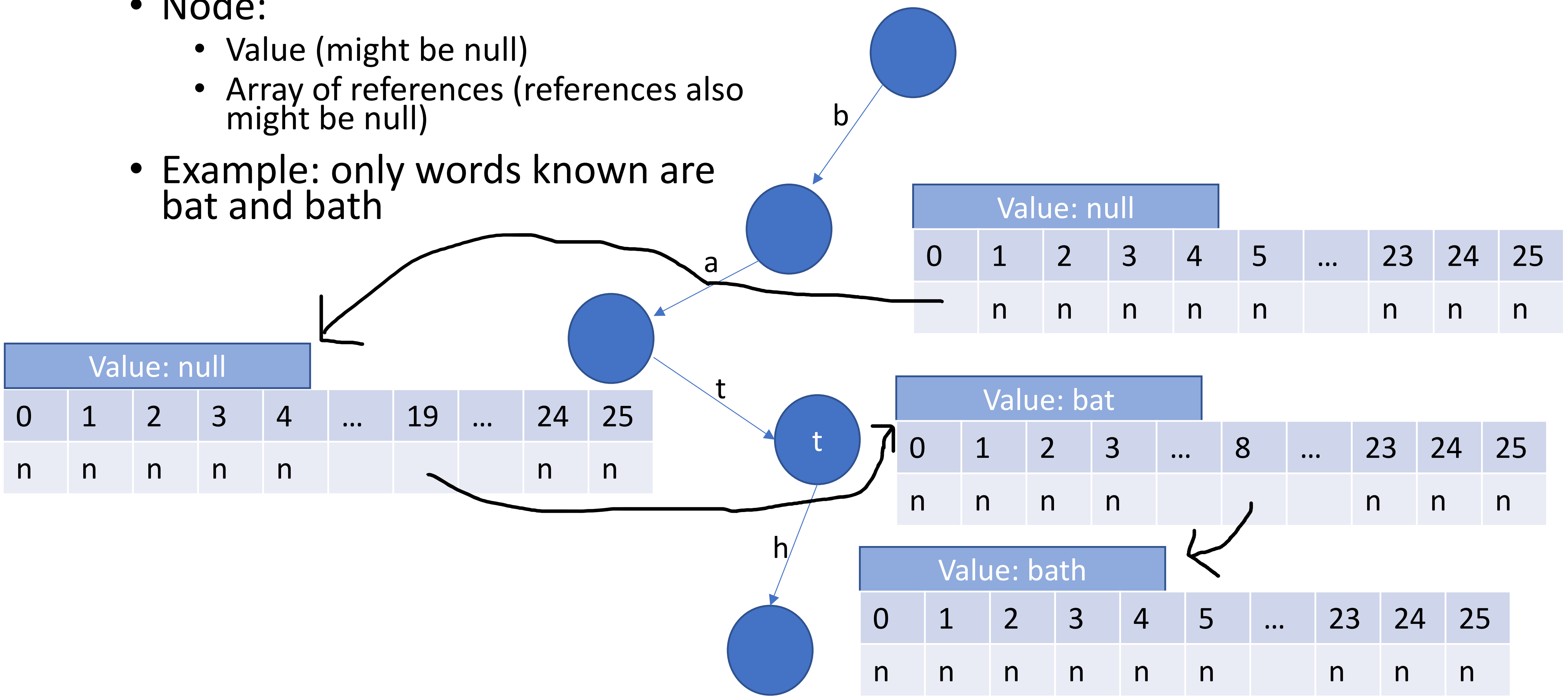
Trie

- A tree where all possible branches are predetermined (by alphabet, constant set of numbers, etc)


Google
Google Search
I'm Feeling Lucky

Trie Data Structure

- Node:
  - Value (might be null)
  - Array of references (references also might be null)
- Example: only words known are bat and bath

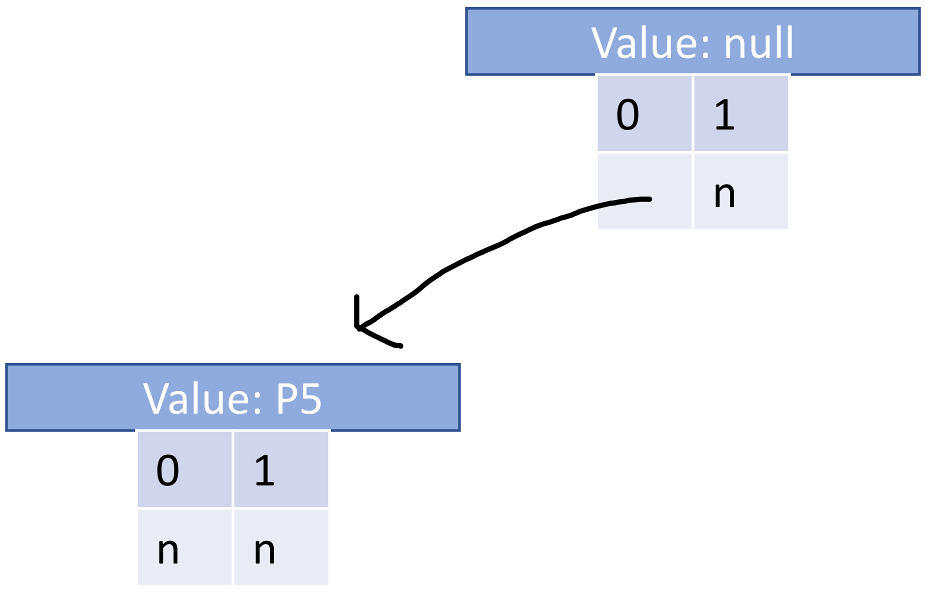


IP Address Reminders

- The following are different representations of the same network:
  - 192.168.74.0/24
  - 192.168.74.0 with subnet mask 255.255.255.0
  - 110000001010100001001010*
- Note: to study algorithms, we’ll use shorter IP prefixes but realistically they can be 0-32 bits

Unibit Trie

- Branch left for 0, right for 1
- P5: 0*



Unibit Trie

- Branch left for 0, right for 1
- Represent this database

Interface	Prefix
P1	101*
P2	111*
P3	11001*
P4	1*
P5	0*
P6	1000*

Draw Trie:

Additions

- Follow the tree using the bits in the prefix
- If node already exists, mark node with value (interface)
- Otherwise, create new node(s) for new prefix

Additions

- Follow the tree using the bits in the prefix
- If node already exists, mark node with interface
- Otherwise, create new node(s) for new prefix
- Add P7, P8, P9, P10, P11 to your existing tree

Interface	Prefix
P1	101*
P2	111*
P3	11001*
P4	1*
P5	0*
P6	1000*
P7	100000*
P8	100*
P9	110*
P10	011*
P11	11*

Lookups

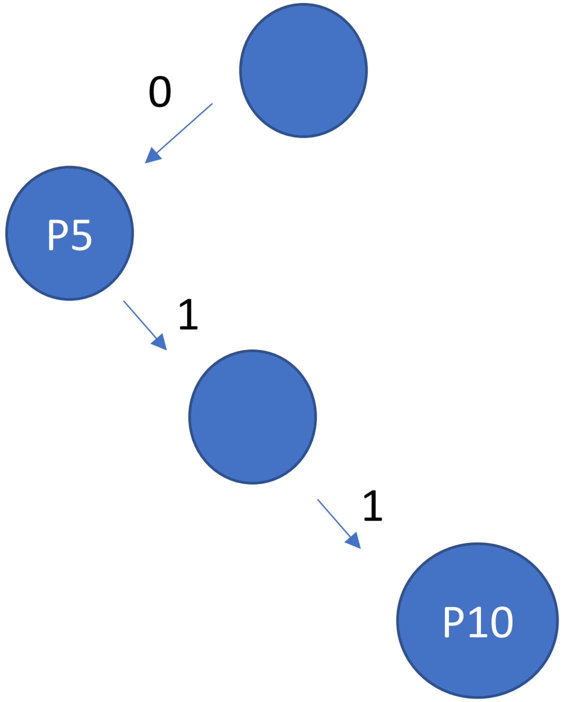
- Follow the tree using the bits in the prefix
- Keep track of best matching interface so far, and update each time we encounter a node that stores an interface

Lookups

- Follow the tree using the bits in the prefix
- Keep track of best matching interface so far, and update each time we encounter a node that stores an interface
- IP addresses are 32 bits, but given the first 8, which interface should packet be sent to?
  - 10010010
  - 11001100
  - 10000101
  - 01000000

Deletions

- Follow the tree using the bits in the prefix
- If node is internal (not leaf), remove the interface from the node
- If node is a leaf, remove the node AND any one-way branches that lead to it



Deletions

- Follow the tree using the bits in the prefix
- If node is internal (not leaf), remove the interface from the node
- If node is a leaf, remove the node AND any one-way branches that lead to it
- Delete P10
- Delete P11

Multibit Tries

- Unibit Tries: worst-case memory accesses: 32
  - If each takes 10 nsec, this takes 320 nsec -> not good enough for faster (more common) wires
- Can we look at more than one bit at a time?
  - If we could look up 4 bits at a time, we could lower this to 8 memory accesses
  - Main problem: what about prefixes like 11001*?

Prefix Expansion

- If using a **stride** of m (looking at m bits at a time), transform the existing database such that all prefix lengths are multiples of m
- If we're trying to look at 4 bits at a time, 10* -> P1 would become 4 entries:
  - 1000 -> P1
  - 1001 -> P1
  - 1010 -> P1
  - 1011 -> P1
- 11001* -> P2: 8 entries:
  - 11001000, 11001001, 11001010, 11001011, 11001100, 11001101, 11001110, 11001111 -> all to P2

Prefix Expansion

- What happens if there is a collision, i.e., we expand and the expanded prefix already exists?
  - Use the interface for the prefix that was originally longer
- 11001* - P2
- 11001111* - P15
- Expansion of 11001* to multiples of 4 includes 11001111*
- Choose P15 because its original prefix (11001111*) was longer

Transform database to multiples of 3

Interface	Prefix
P1	101*
P2	111*
P3	11001*
P4	1*
P5	0*
P6	1000*
P7	100000*
P8	100*
P9	110*

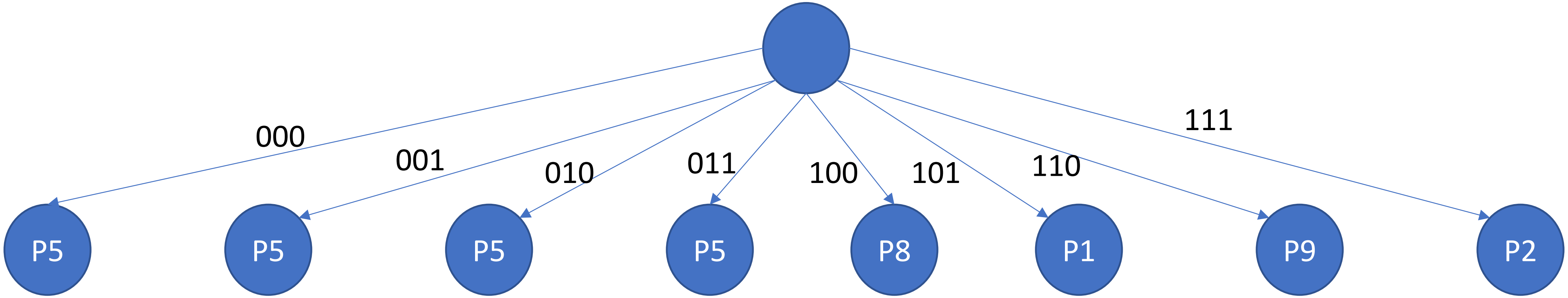
Prefix Expansion

- If using a stride of m (looking at m bits at a time), transform the existing database such that all prefix lengths are multiples of m
- Result: fewer prefix lengths, more prefixes

Form trie with expanded prefixes

Form trie with expanded prefixes

- Practice: Fill out rest of trie



Tries Stored in Memory

000	P5
001	P5
010	P5
011	P5
100	P8
101	P1
110	P9
111	P2

000	
001	
010	P3
011	P3
100	
101	
110	
111	

000	P7
001	P6
010	P6
011	P6
100	
101	
110	
111	

Fixed Stride Length

- In example so far, within each level in the trie, we look at the same number of bits

Variable Stride Length

000	P5
001	P5
010	P5
011	P5
100	P8
101	P1
110	P9
111	P2

2 bits

00	
01	P3
10	
11	

3 bits

000	P7
001	P6
010	P6
011	P6
100	
101	
110	
111	

Optimal Stride Length

- For each table, what is the minimum stride length we could use while keeping all information? Assume for now we can only use one table

000	P6
001	P6
010	
011	
100	
101	
110	P3
111	P3

000	P6
001	P6
010	
011	
100	
101	
110	P3
111	

00	P4
01	P4
10	
11	

0000	P6
0001	P6
0010	P6
0011	P6
0100	P3
0101	P3
0110	P2
0111	P2
1000	
1001	
1010	
1011	
1100	P7
1101	P7
1110	P7
1111	P7

Choosing Optimal Stride Lengths Across Trie

- Increasing stride lowers number of lookups needed, at the cost of higher memory usage
- Using variable stride lengths can help compress tries to lower their memory usage
- Tries can be further compressed – we'll look at this next time
- There are some non-trie options for IP lookup – also for next time