University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 7: next Friday @ 11:59 pm
  - Lab 6: Monday 10/31 @ 11:59 pm
  - Nothing due this week
- Midterm Exam
  - Wednesday 10/19 (MW Section) and Thursday 10/20 (TuTh Section)
    - in-person, closed-book
- Weekly Live QA Session on Piazza
  - Friday 4:30-5:30 pm

Previous lecture

- ADT Priority Queue (PQ)
  - Heap implementation
- Heap Sort
- Indexable PQ

This Lecture

- Muddiest Points
- Introduction to the ADT Graph

Muddiest Points

- **Q: what is going to be on the midterm?**
- Up to and including material covered on Monday 10/10 and Tuesday 10/11
- Please check the study guide on Canvas
  - practice test on GradeScope
  - old exam
    - will try to post the answer as soon as possible

Muddiest Points

- **Q: I'm confused about entropy and the equations for that. How is it useful?**
- It helps us determine the “information content” in a file.
- In lossless compression, a file cannot be compressed to less than its entropy

Muddiest Points

- **Q: How do we determine the entropy of a given file we're trying to compress?**
- Given that
  - the file has *n* total characters and *K* unique characters,
  - *f(c)* is the frequency of character *c* in the file
- Shannon's entropy: H(file) = $\sum_{c=0}^{K} \frac{f(c)}{n} log_2(\frac{f(c)}{n})$ bits/character
- Underlying assumption:
  - the file has been generated by a source that produces **independent characters**
  - **Huffman Compression is optimal under that assumption**
  - This assumption may be wrong though!
    - repeated long strings → use LZW
    - long sequences of identical characters → use RLE (Run Length Encoding)
  - We may need to try different compression algorithms on the file

Muddiest Points

- **Q: What might be some examples in where we might pick one compression type over the other?**
- Depends on the structure of the input file
- long strings of identical values
  - → RLE
- repeated long strings
  - → LZW
- frequently occurring values
  - → Huffman
- few different characters
  - → fixed-length codewords with <8 bits

Muddiest Points

- **Q: Does entropy play a role in the implementation of the compression algorithm itself? Or is it only used for selecting the best algorithm?**
- Some compression algorithms attempt to reach Shannon's Entropy lower bound on the file size
  - e.g., Huffman Encoding and Arithmetic Encoding

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e a e d e e

0	a
1	b
2	c
3	d
4	e

Output:

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e a e d e e

0	a
1	b
2	c
3	d
4	e

Output:

4

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e a e d e e

0	a	e
1	b	a
2	c	b
3	d	c
4	e	d

Output:

4

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e a e d e e

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

4 1

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e a e d e e

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:

4 1 1

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e	a	e	d	e	e

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

4	1	1	4

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e	a	e	d	e	e

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

4	1	1	4	1

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
-

Input:

e	a	e	d	e	e

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

4	1	1	4	1	0

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4	1	1	4	1	0

0	a
1	b
2	c
3	d
4	e

Output:

e

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4	1	1	4	1	0

0	a	e
1	b	a
2	c	b
3	d	c
4	e	d

Output:

e

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4	1	1	4	1	0

0	a	e	a
1	b	a	e
2	c	b	b
3	d	c	c
4	e	d	d

Output:

e	a

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:						
	4	1	1	4	1	0

0	a	e	a	e
1	b	a	e	a
2	c	b	b	b
3	d	c	c	c
4	e	d	d	d

Output:						
	e	a	e	4	1	0

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4	1	1	4	1	0

0	a	e	a	e	d
1	b	a	e	a	e
2	c	b	b	b	a
3	d	c	c	c	b
4	e	d	d	d	c

Output:

e	a	e	d

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4	1	1	4	1	0

0	a	e	a	e	d	e
1	b	a	e	a	e	d
2	c	b	b	b	a	a
3	d	c	c	c	b	b
4	e	d	d	d	c	c

Output:

e	a	e	d	e

Muddiest Points

- **Q: Not exactly clear on move to front encoding**
- **Decoding**

Input:

4	1	1	4	1	0

0	a	e	a	e	d	e	e
1	b	a	e	a	e	d	d
2	c	b	b	b	a	a	a
3	d	c	c	c	b	b	b
4	e	d	d	d	c	c	c

Output:

e	a	e	d	e	e

Muddiest Points

- **Q: LZW doesn't seem like prefix-free compression. Do we use delimiters between the codes or is there some way to tell the numbers apart. Even just 255 contains multiple interpretations of ASCII codes (2, 55 or 25, 5)**
- Great Question!
- Each integer is the same number of bits (e.g., 12 bits)
- So, the expansion program reads 12 bits at a time
- No need for delimiters nor prefix-free encoding of the integers

Muddiest Points

- **Q: calculating the bits for the lzw compression (last question on the tophat), we didn't go over this during the example in class**
- The second column represents the output of LZW compression, i.e., the compressed file
- Each integer is 12-bit codeword (per the question)
- Total compressed file size = # codewords * 12

Muddiest Points

- **Q: Corner case of lzw expansion**
- The tricky (corner) case happens when the longest match in compressions happens to be the string that was just added in the previous step
- Expansion sees that as a codeword that is not (yet) in its codebook
- Remember that expansion builds the same codebook as compression but is one step behind
- Handling the tricky case:
  - output: previous output + first character of previous output
  - add the same string to the codebook

LZW corner case example

- Compress, using 12 bit codewords: AAAAAA

Cur	Output	Add
A	65	AA:256
AA	256	AAA:257
AAA	257	--

- Expansion:

Cur	Output	Add
65	A	--
256	AA	256:AA
257	AAA	257:AAA

Muddiest Points

- **Q: How can the uncompressed file have more entropy than compressed if the entropy is the average number of bits to represent a word?**
- In lossless compression,

  entropy of compressed file >= entropy of uncompressed file
- Since compressed file has fewer characters than uncompressed
  - entropy/char of compressed file is > entropy/char of uncompressed file

Muddiest Points

- **Q: I was wondering why LZW choose 12bits instead of any other number**
- 12 bits was used just as an example
- Actual implementation use an adaptive codeword size
  - start with 9 bits
  - when codebook full, change to 10 bits
  - when codebook full, change to 11 bits
  - …
  - when codebook full and codeword is 16 bits
    - either stop adding to codebook or reset it
    - depends on the compression ratio

Muddiest Points

- **Q: please make assignments easier**
- Yep!

Muddiest Points

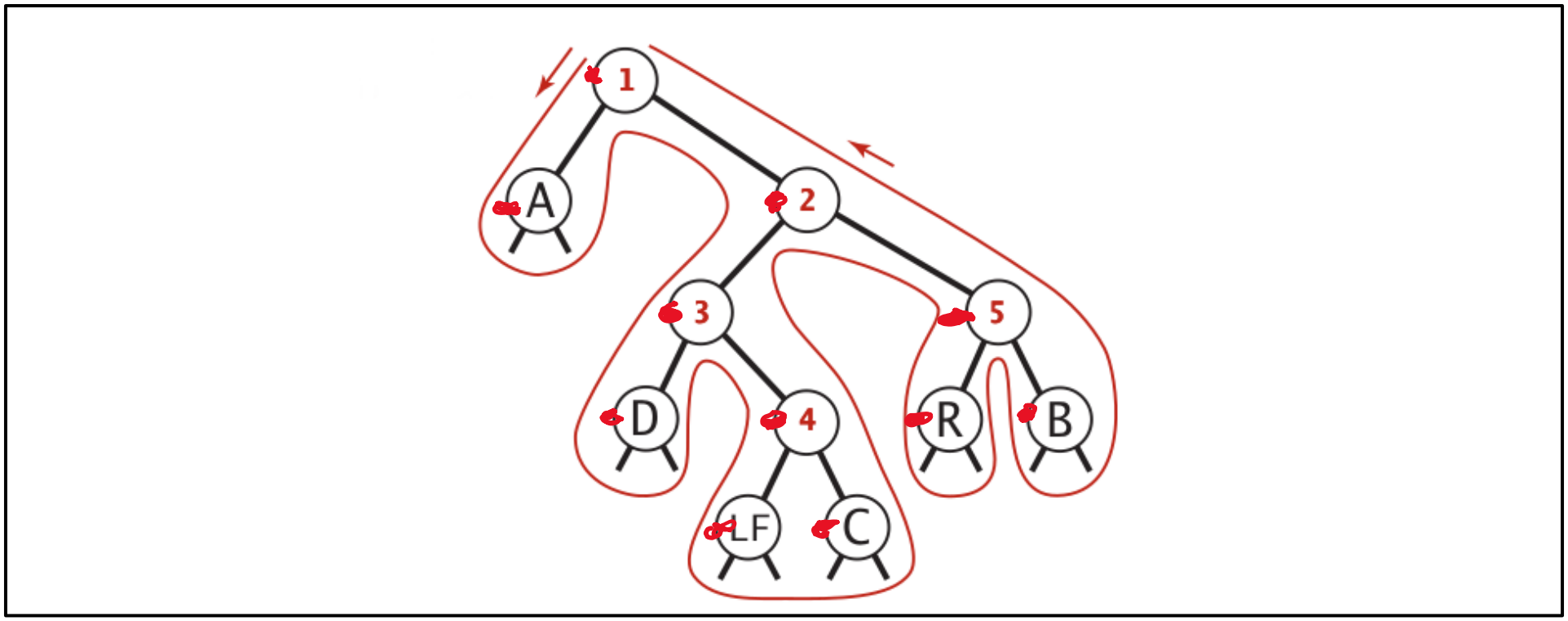
- **Q: Is there a best way to handle ties when doing Huffman compression or is it a purely arbitrary choice?**
- **Q: what are the rules for drawing out the tree in the huffman approach?**
- Ties are arbitrarily handled
- The compressed file size is the same no matter how ties are handled

Muddiest Points

- **Q: the bits needed for the compression post huffman encoding**
- Huffman compression may store the trie in the compressed file
- The trie is encoded using preorder traversal of the nodes
  - encoding internal nodes with 0
  - leaf nodes with 1 followed by the ASCII code of the character inside the leaf

Representing tries as bitstrings

Preorder traversal

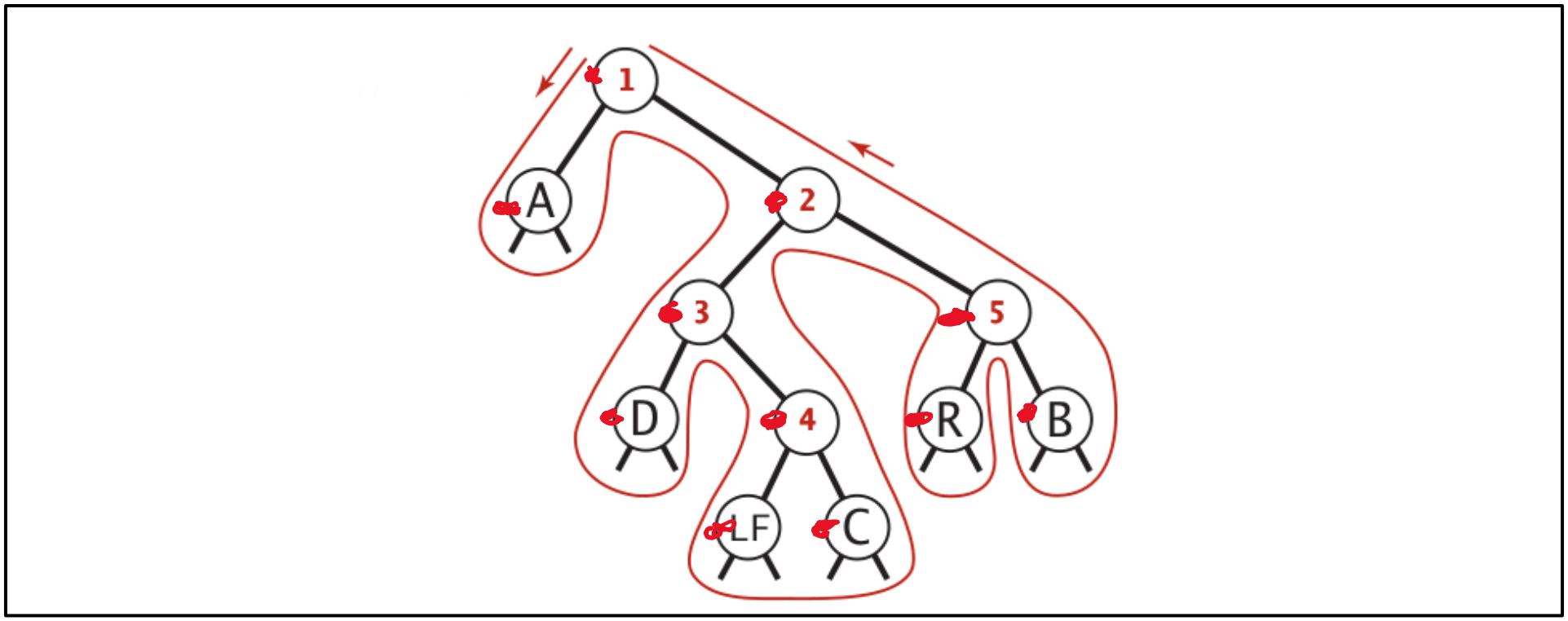


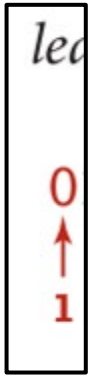
internal node → 0

leaf node → 1 followed by ASCII code of char inside

Representing tries as bitstrings

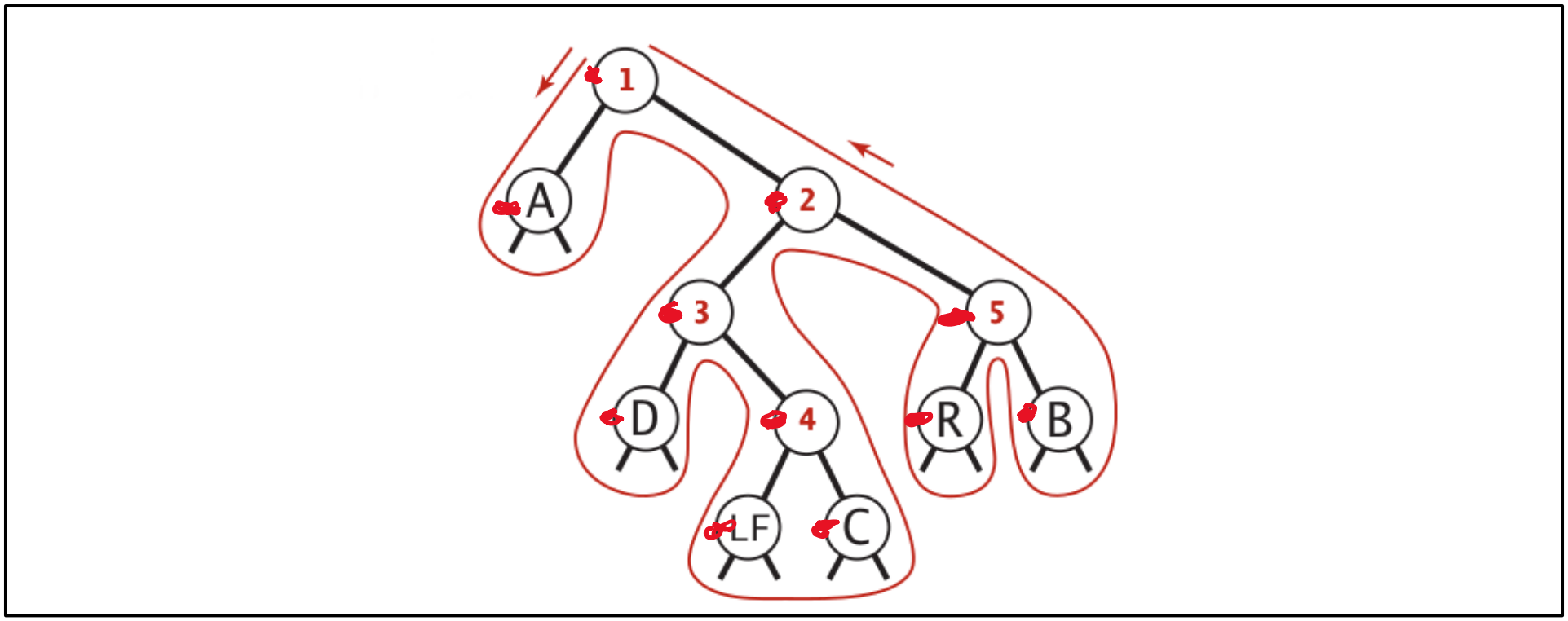
Preorder traversal

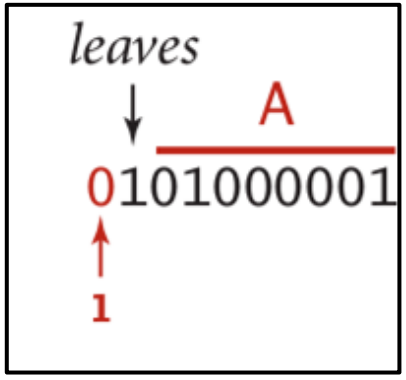




Representing tries as bitstrings

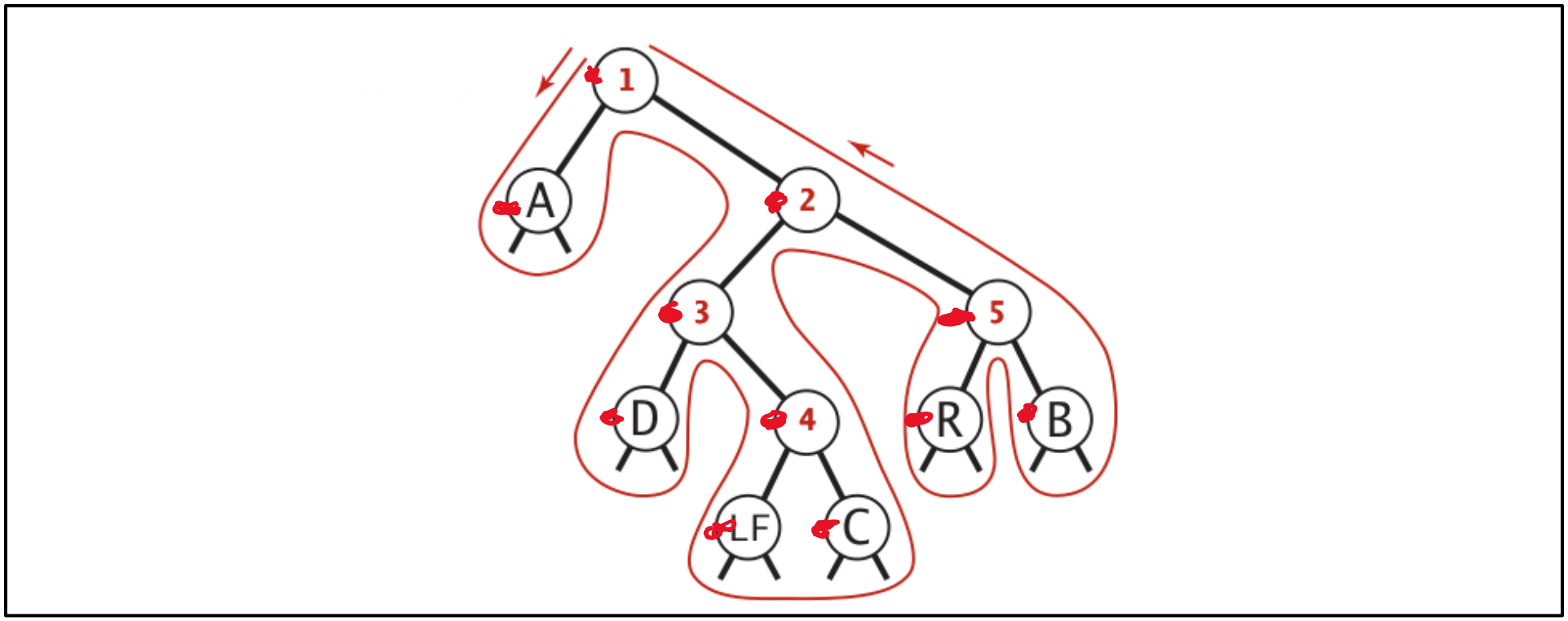
Preorder traversal

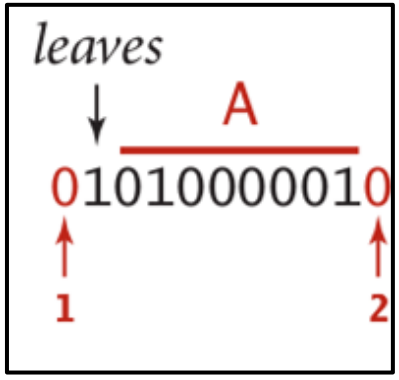




Representing tries as bitstrings

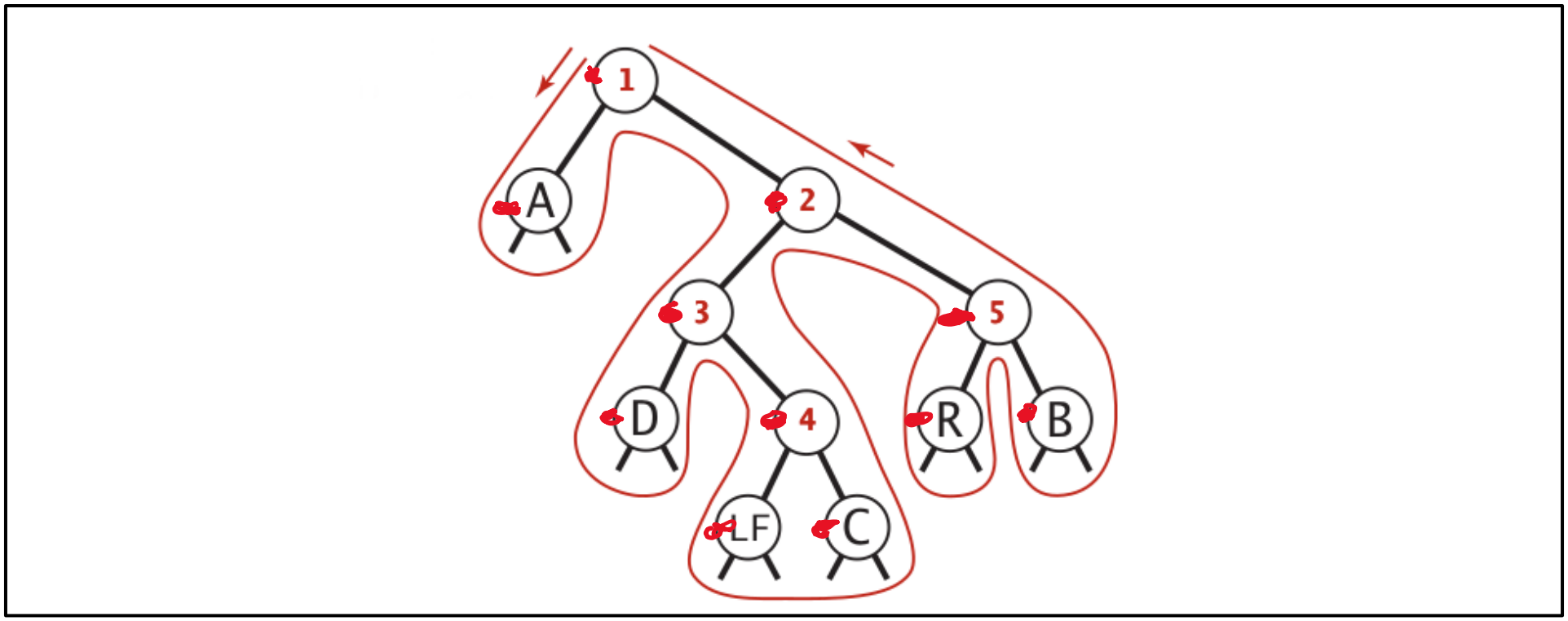
Preorder traversal

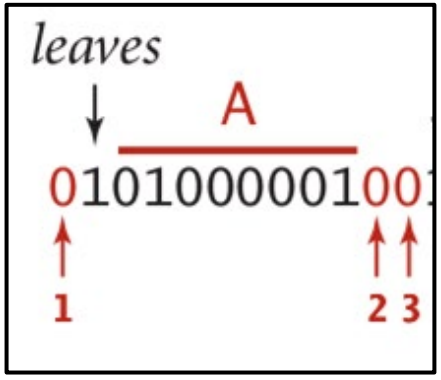




Representing tries as bitstrings

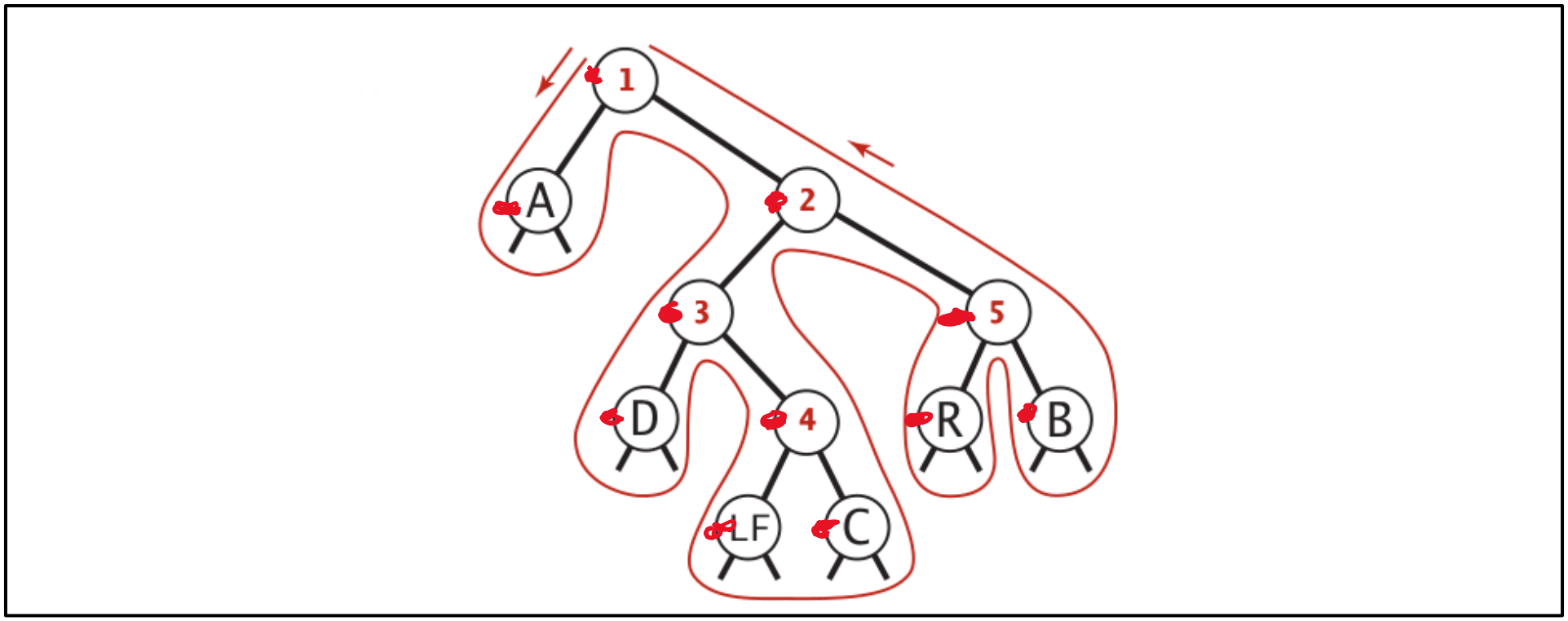
Preorder traversal

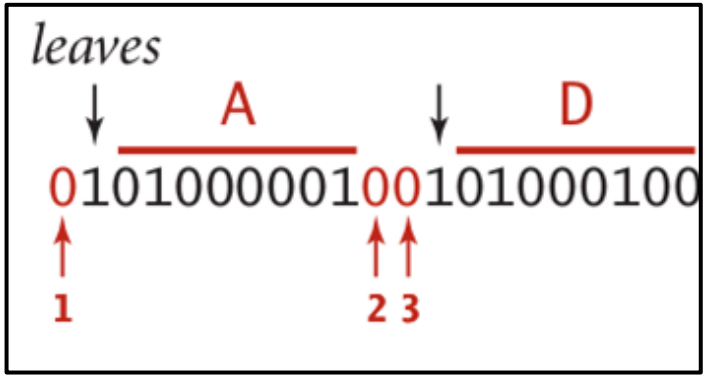




Representing tries as bitstrings

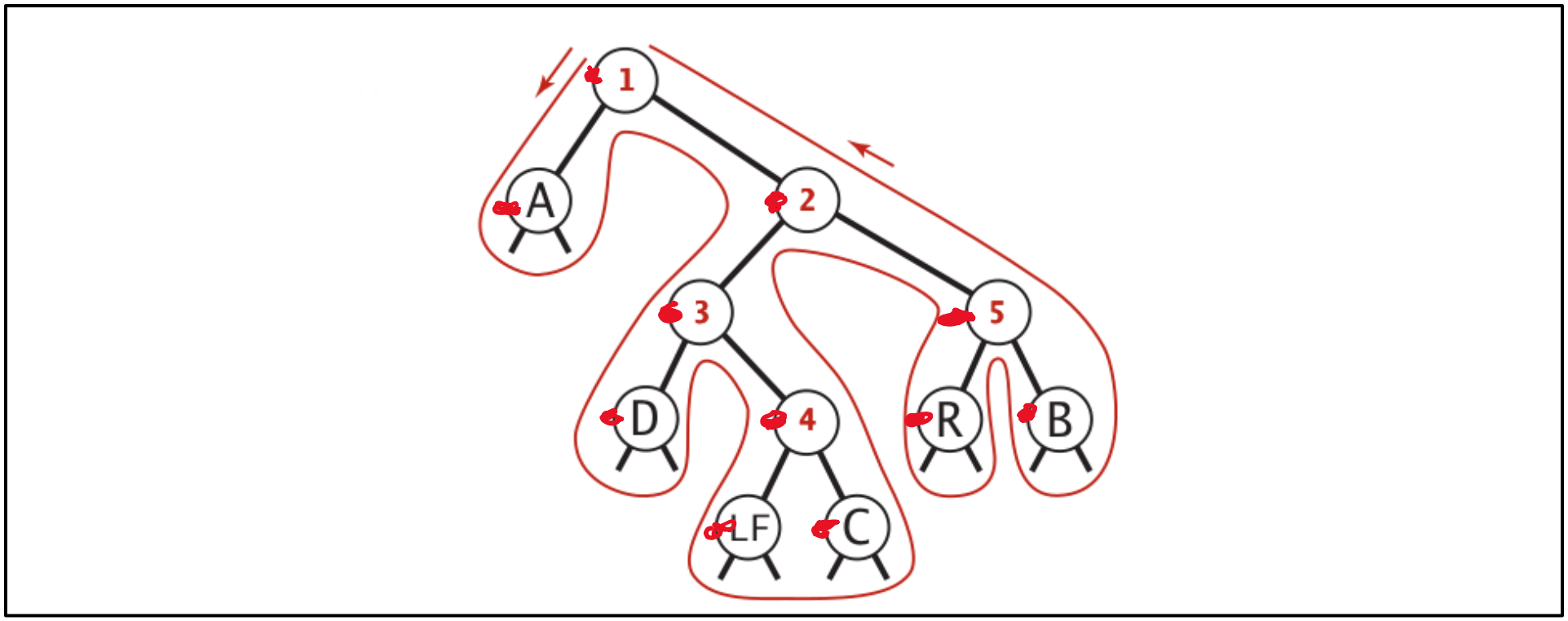
Preorder traversal

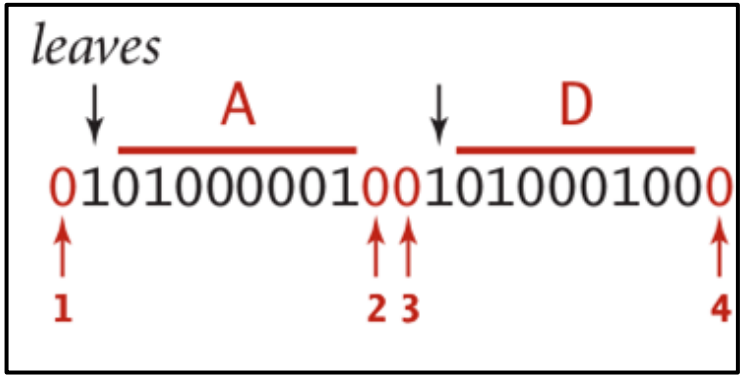




Representing tries as bitstrings

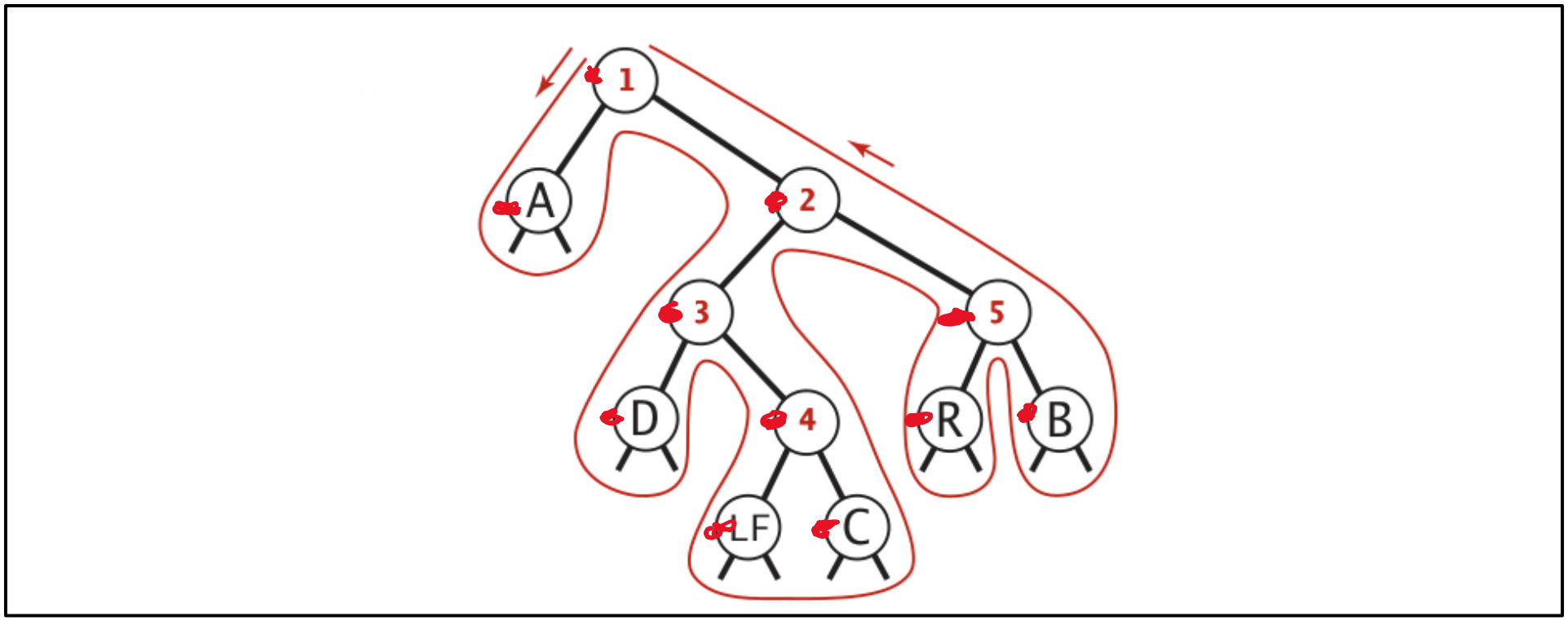
Preorder traversal

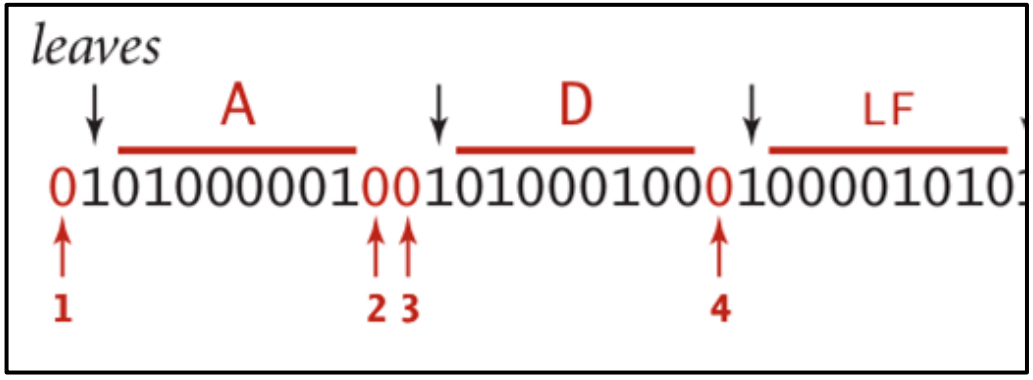




Representing tries as bitstrings

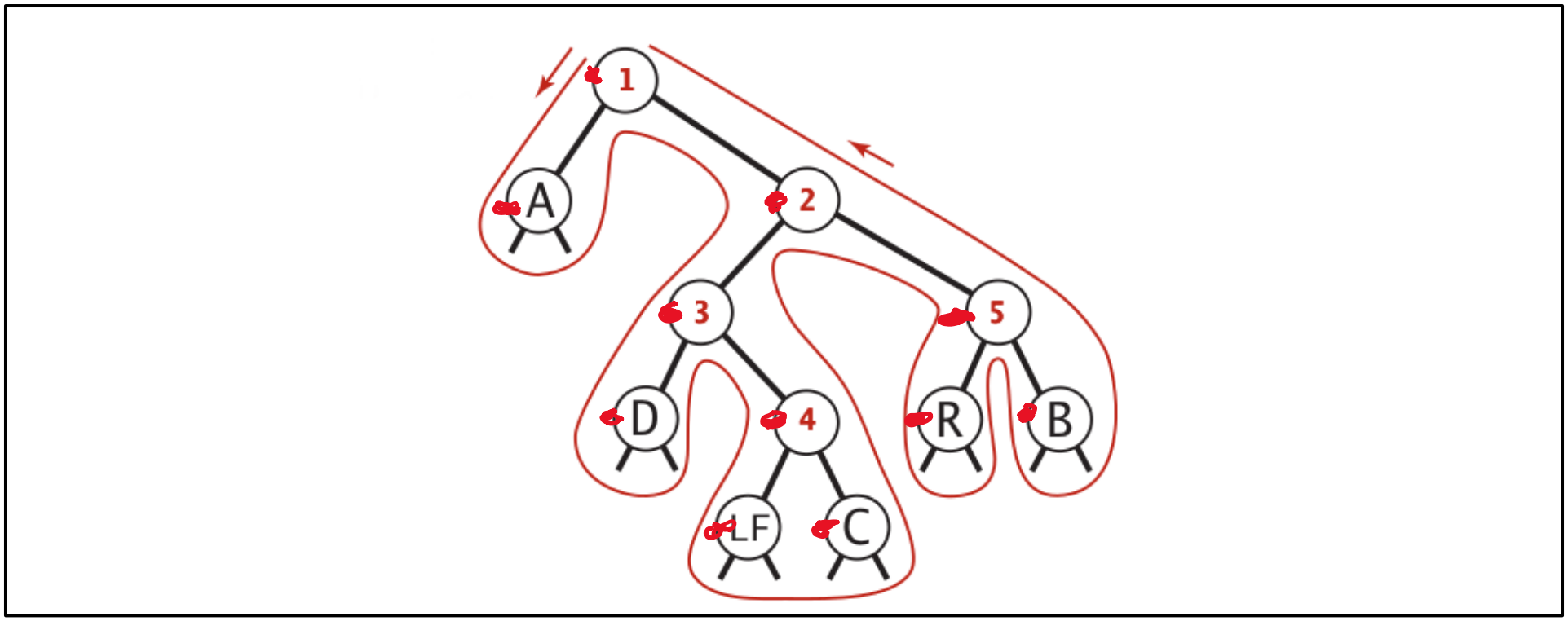
Preorder traversal

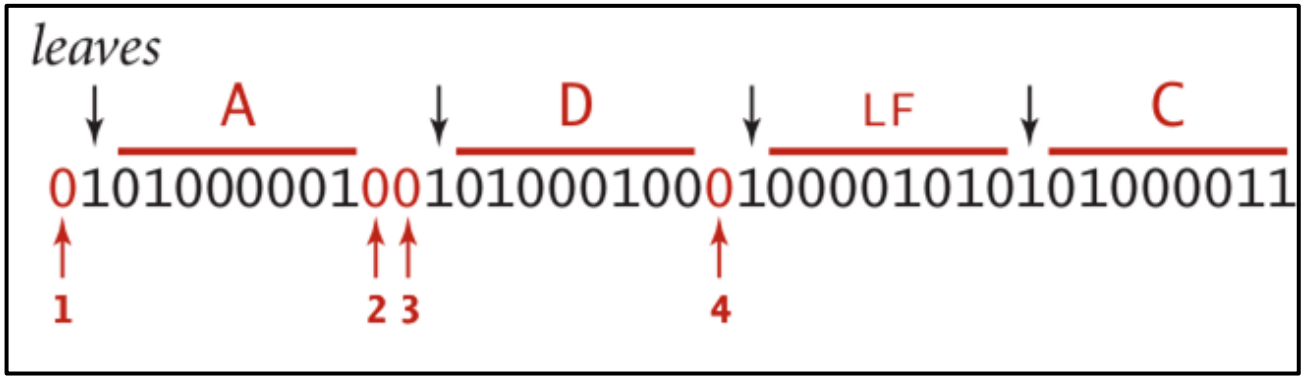




Representing tries as bitstrings

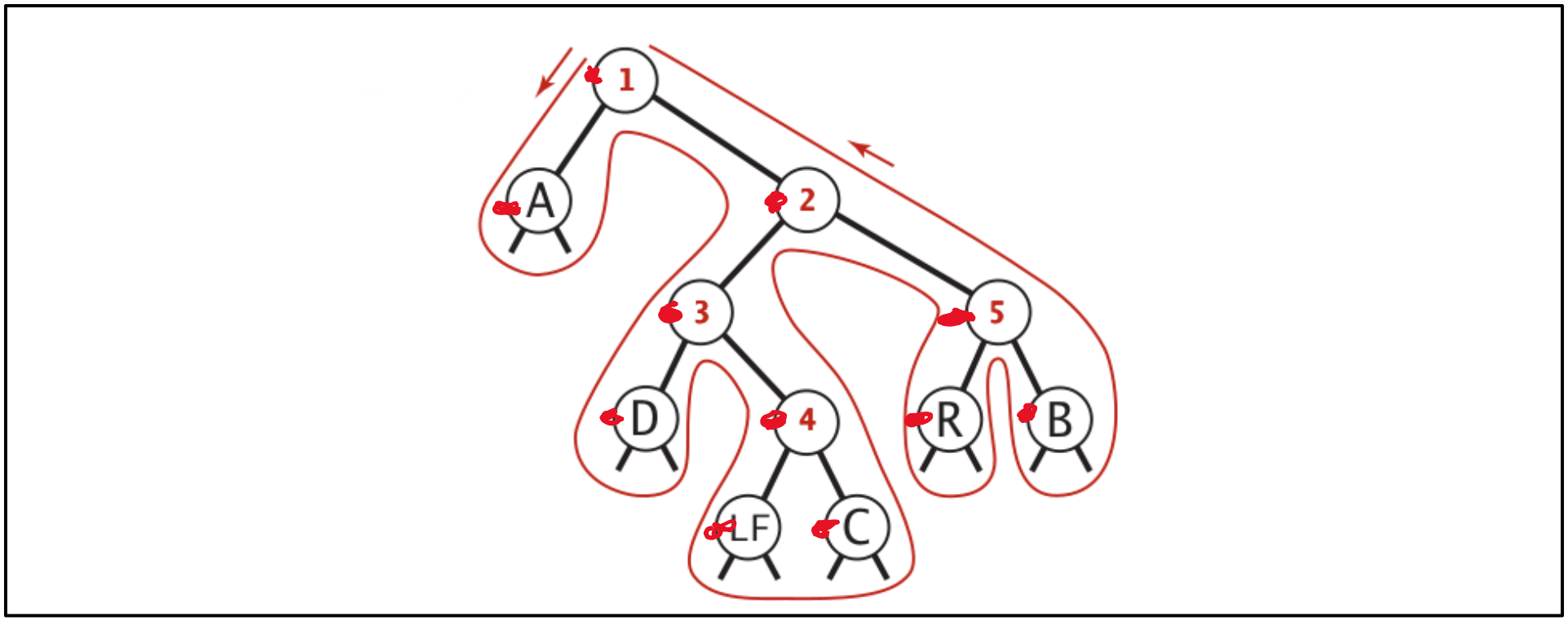
Preorder traversal

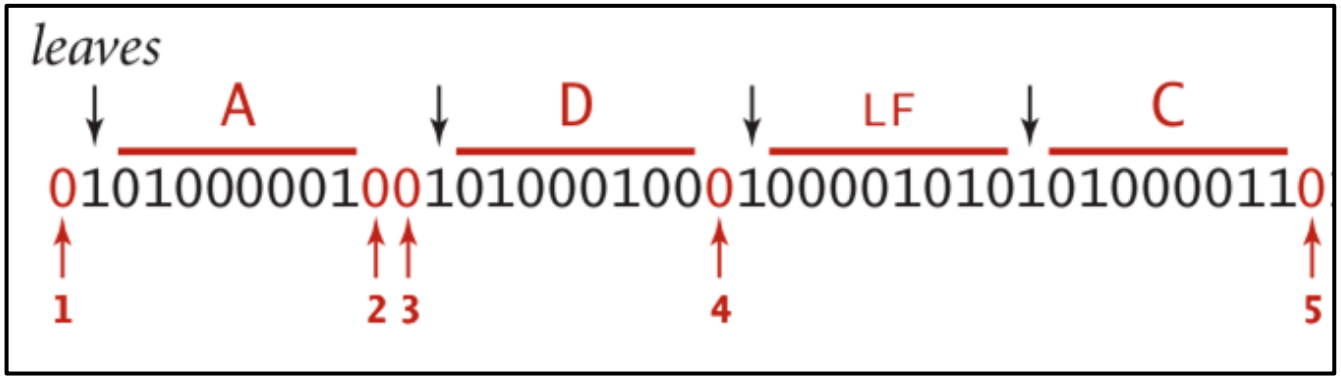




Representing tries as bitstrings

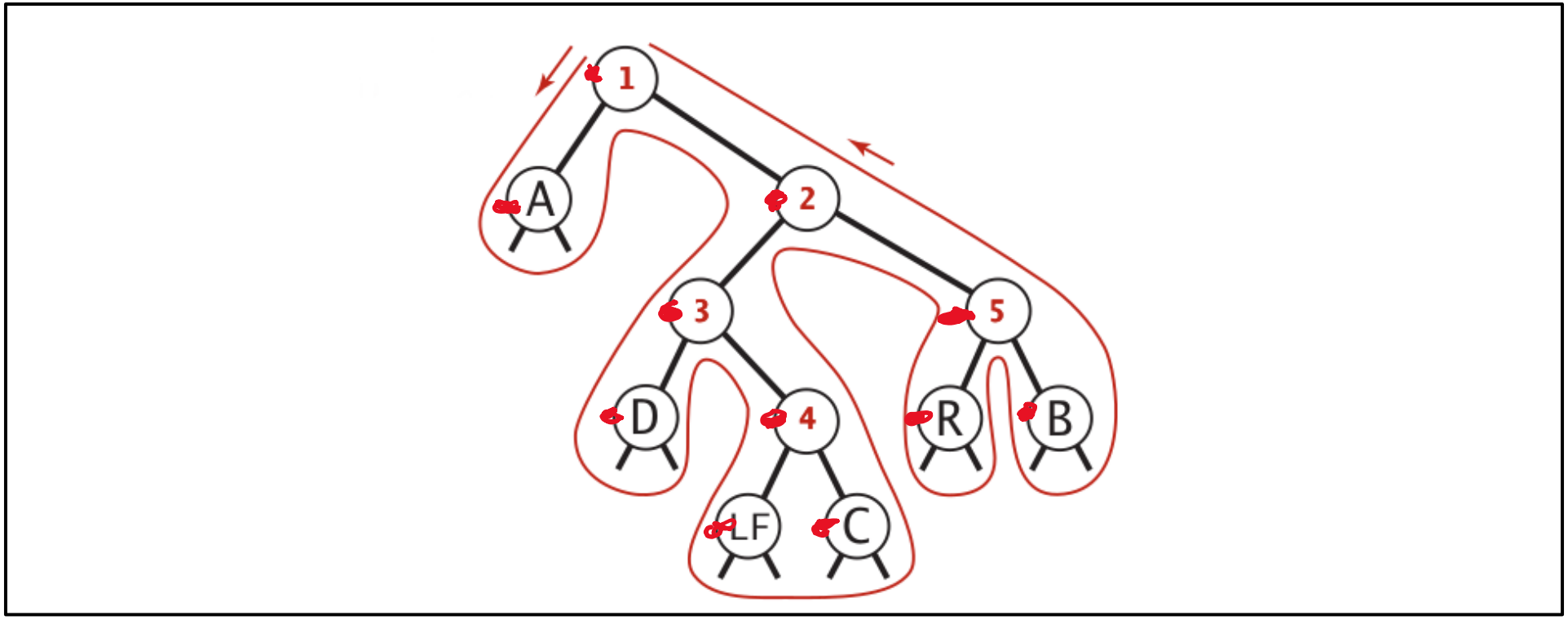
Preorder traversal

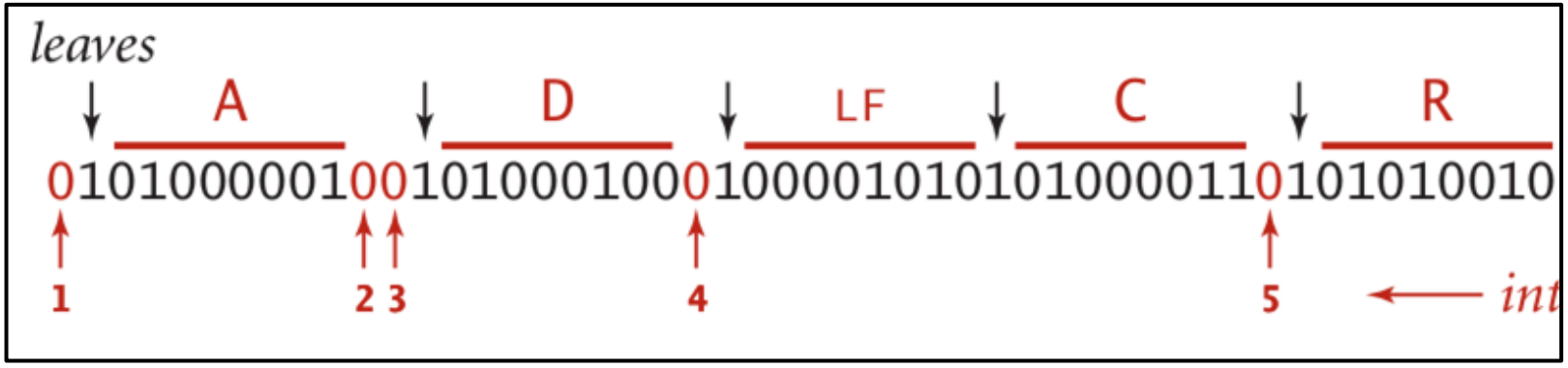




Representing tries as bitstrings

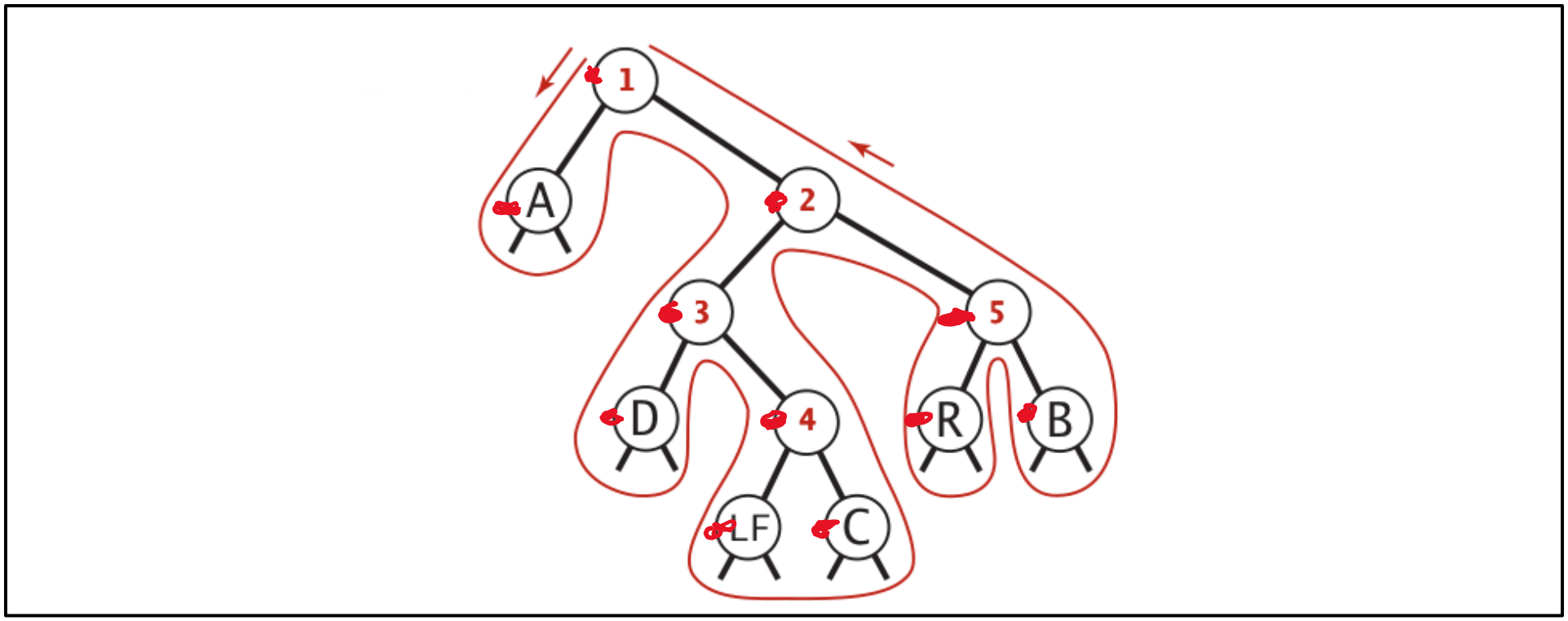
Preorder traversal

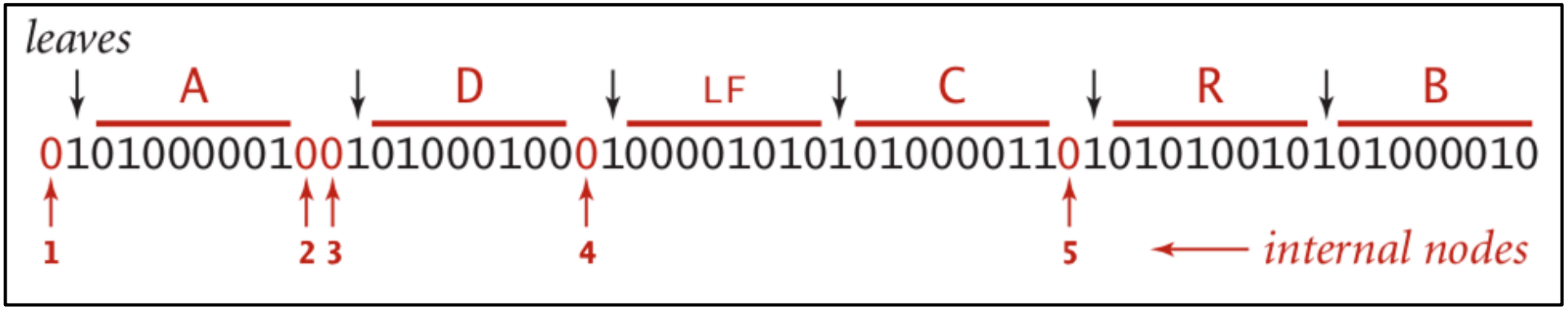




Representing tries as bitstrings

Preorder traversal





Muddiest Points

- **Q: After writing out the Trie as a bitstring, how can you tell where each character starts?**
- First, we are writing bits not characters!
- Each bit represents a node
  - except for leaves, their 1 bit is followed by 8 bits (ASCII code of char inside)
    - but that's also fixed length

Muddiest Points

- **Q: how compression and tries combine. Are you supposed to recreate the original trie while decompressing**
- The compressed file contains:
  - the trie representation in bits
  - the number of characters in the original file
  - the Huffman encoding of the file characters
- The original trie is reconstructed from the trie representation in the compressed file

Muddiest Points

- **Q: Questions are not getting responded to on Piazza in a timely manner. Students had asked several questions pertaining to items on Homework 4 last week but the questions were not addressed until AFTER Homework 4 was already graded. The questions I got marked wrong this week were the exact questions I had asked for clarification on but didn't get a response to (until after the homework was graded and returned...by that point it was too late).**
- I will hold a Live QA Session on Piazza every Friday 4:30-5:30 pm

Muddiest Points

- **Q: the overall concepts of code blocks/code words, like how do they fit into everything?**
- The input file is divided into code blocks
- Each code block is replaced by a codeword
- For Huffman:
  - code blocks are single characters
  - codewords are variable-length bit strings
- For LZW:
  - code blocks are the longest-match strings (variable length)
  - codewords are fixed-length integers (e.g., 12 bits)
- For RLE:
  - code block are long strings with identical characters (variable length)
  - codewords are fixed-length integer followed by fixed-length ASCII of the character

Muddiest Points

- **Q: Just making sure I got the order of the compression framework: We go from file to code block to code word via compression then back to code block via expansion**
- Yep!

Muddiest Points

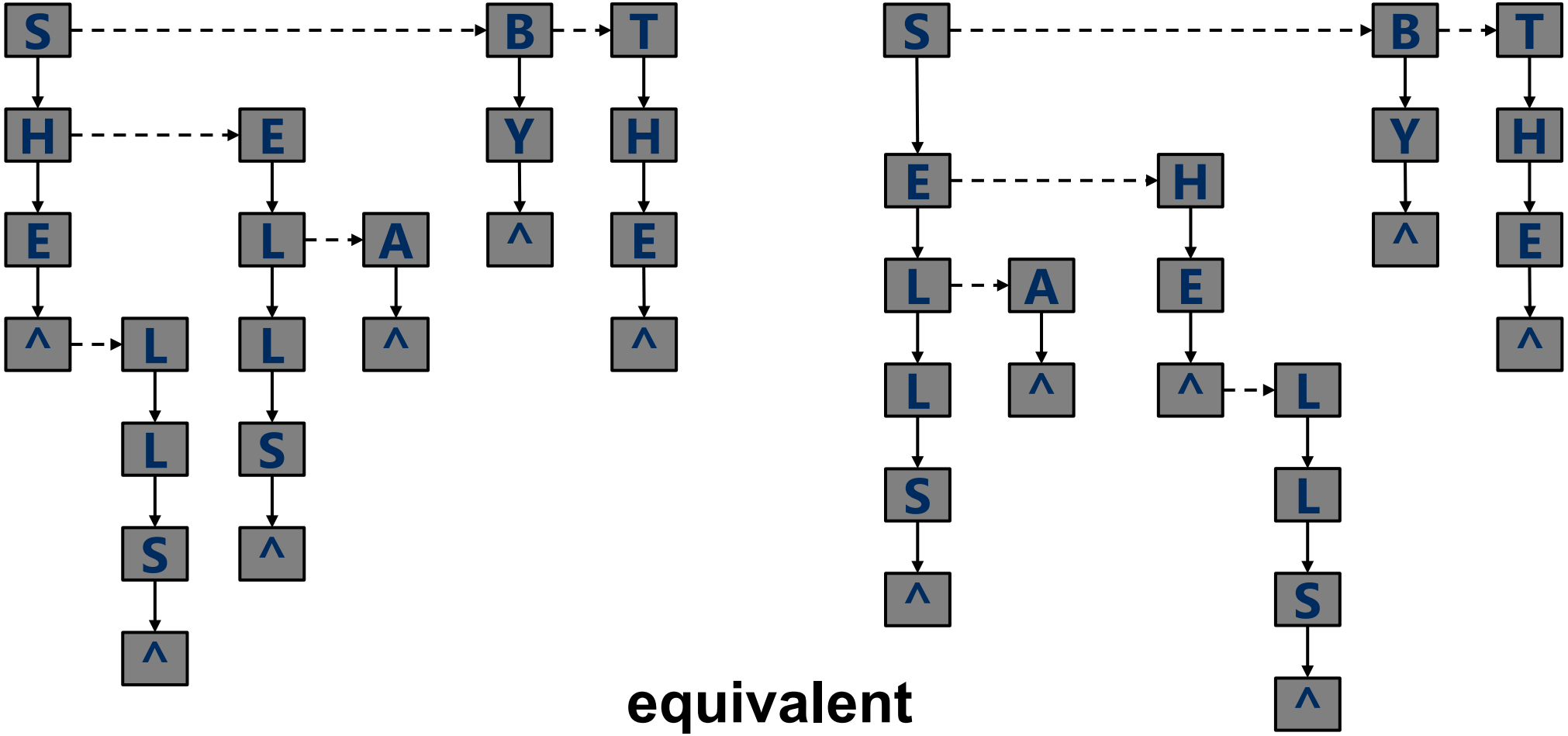
- **Q: Can we get partial credit on labs and projects?**
- We assign partial credit on projects only and only when autograder score <= 40%
  - partial credit has a ceiling of 60% of the autograder score
- Unless you think you lost points in the autograder because of an autograder error

Muddiest Points

- **Q: How are you able to create unique codes without leading to another prefix**
- Characters are leaves in the Huffman tree
- No two characters share the same path from root
- Codewords encode the root-to-leaf path
- No two codewords share a prefix

Muddiest Points

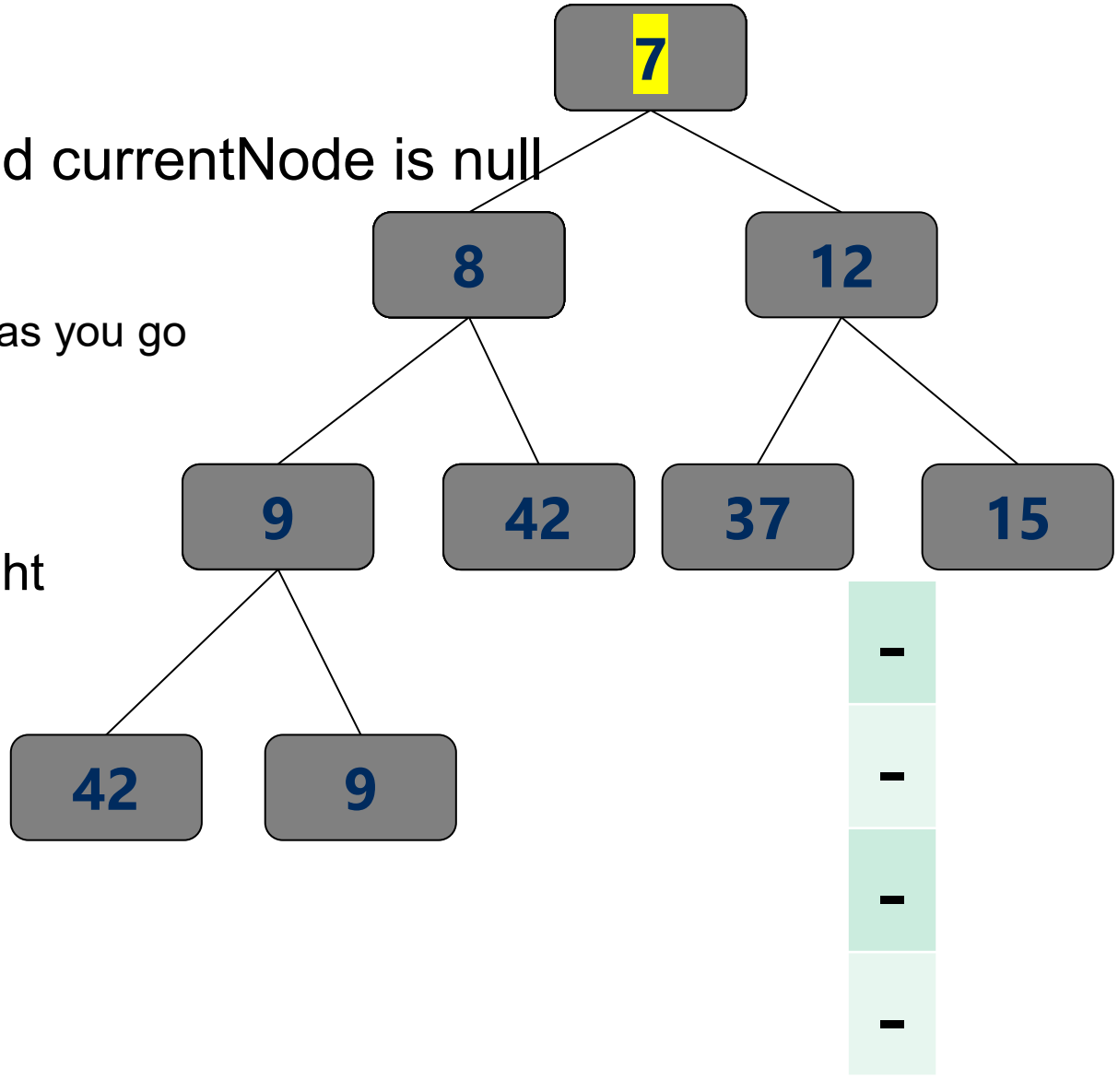
- **Q: In DLB tries, can you interchange a parent node's child with any one of the child's siblings? I think you can but you would have to change the sibling links, is that correct?**
- Correct!



Muddiest Points

- **Q: Review iterative inorder traversal**

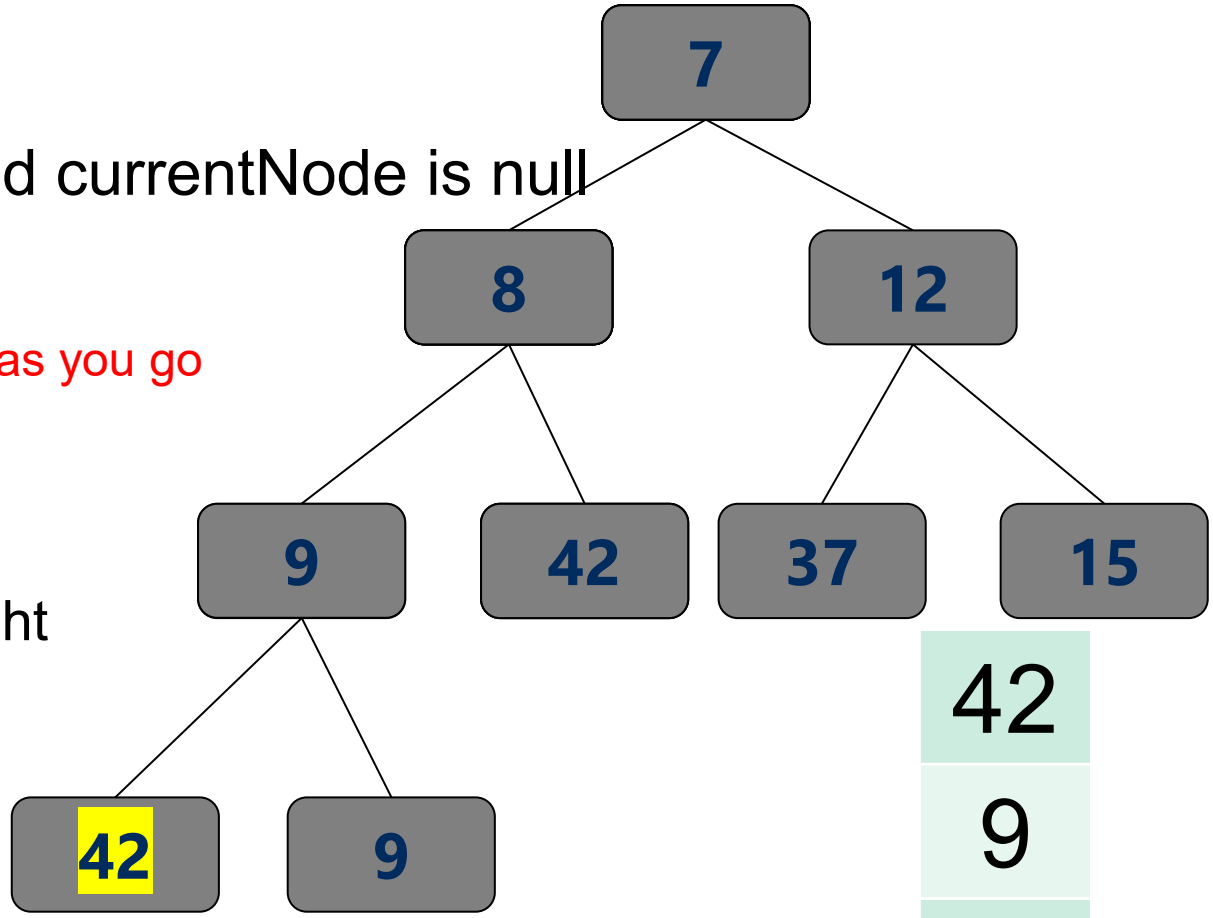
- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

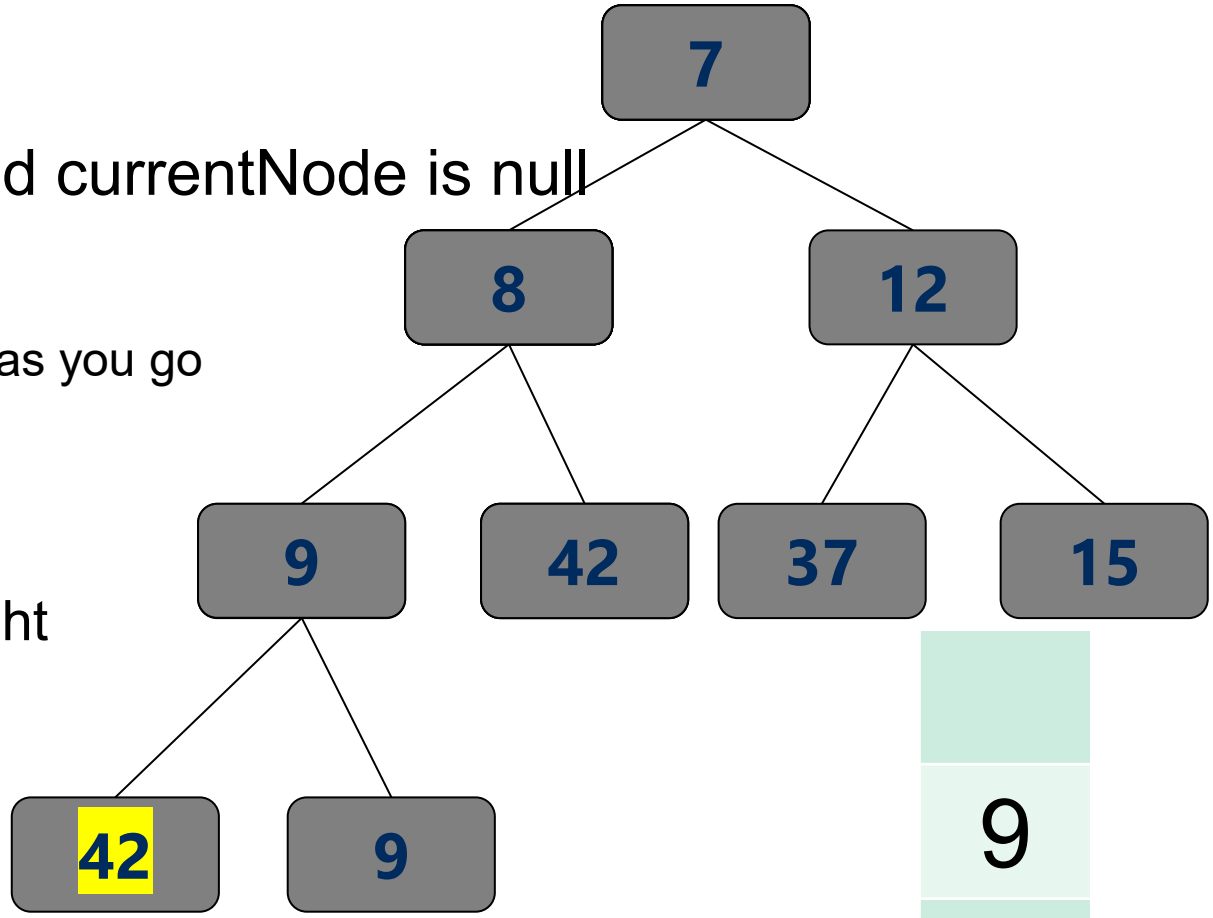


Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

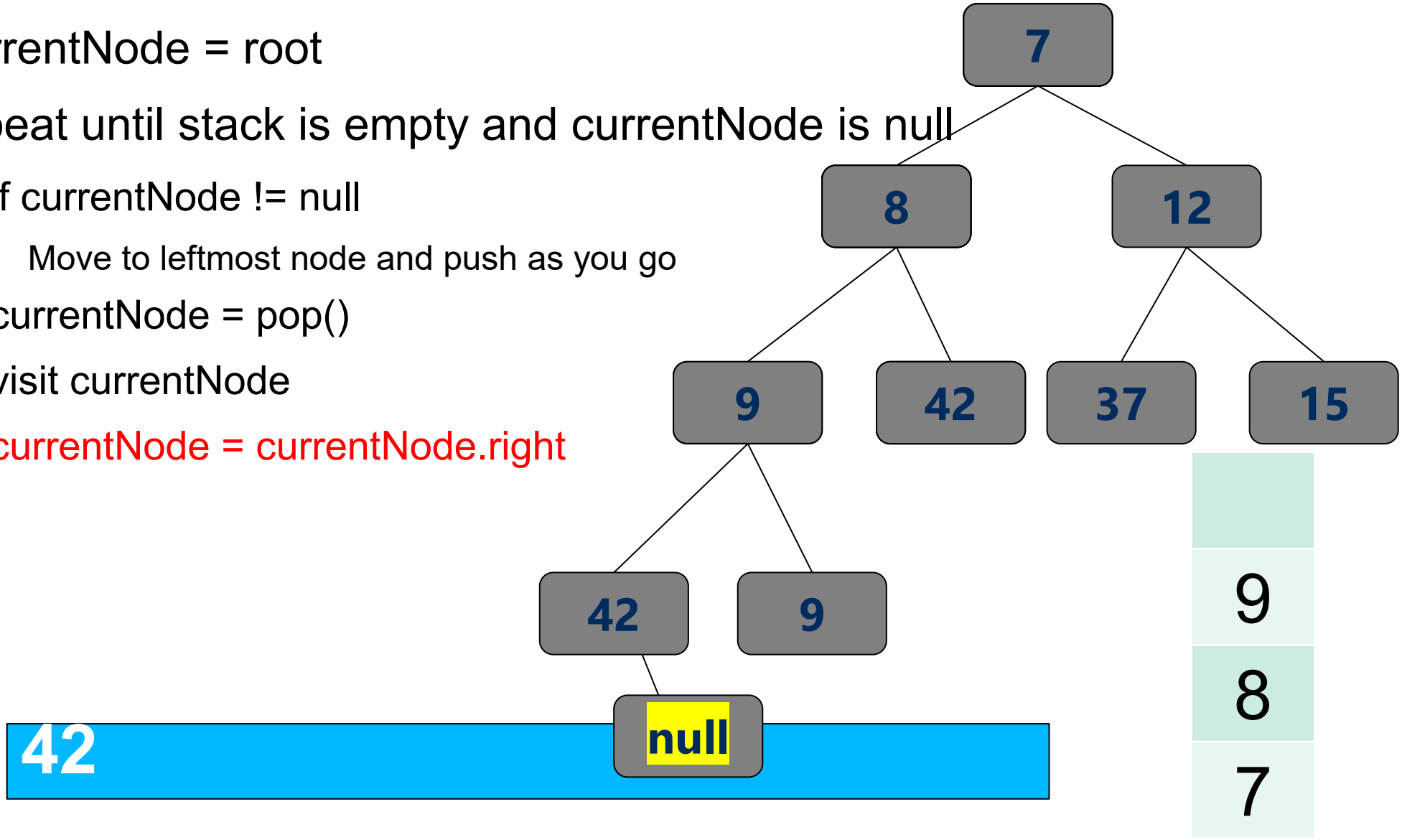
- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



Muddiest Points

- **Q: Review iterative inorder traversal**

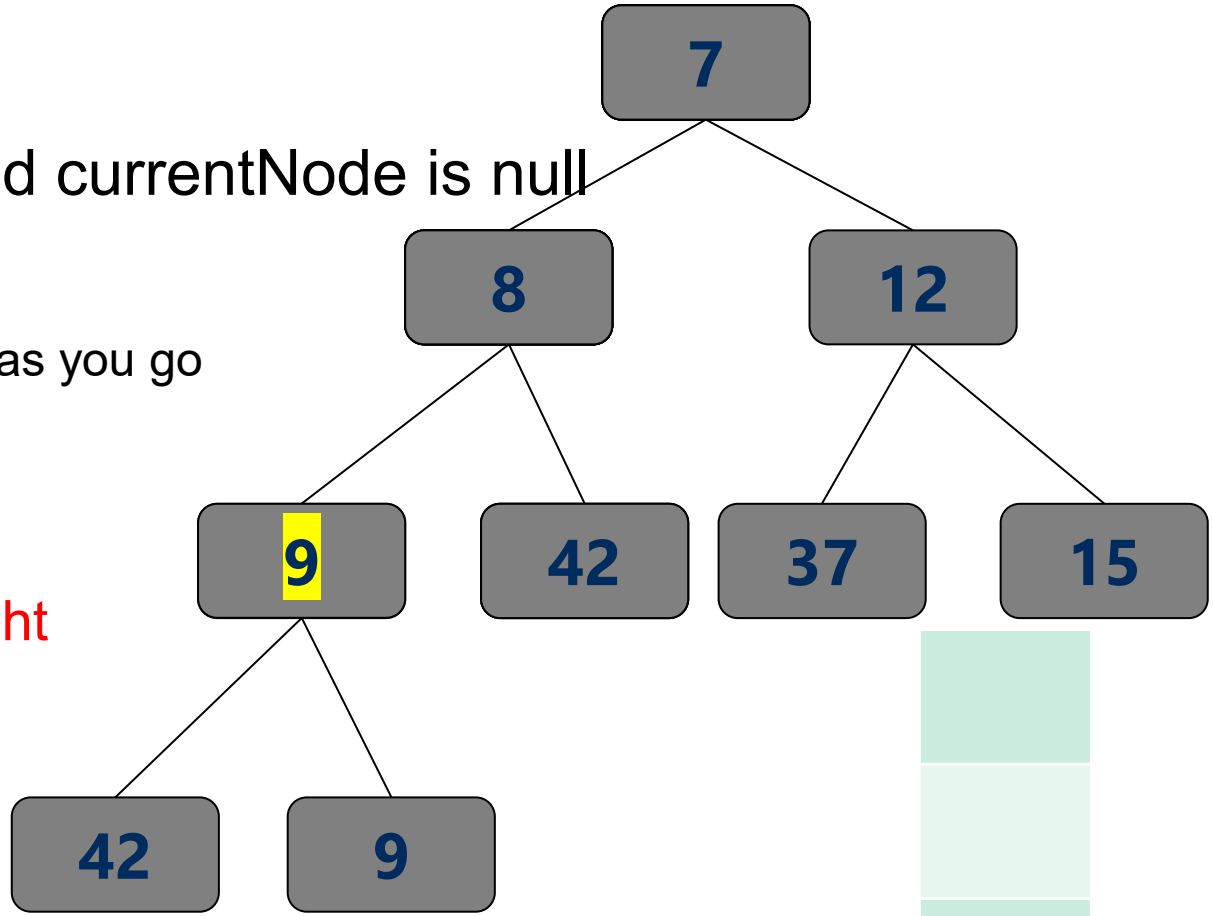
- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

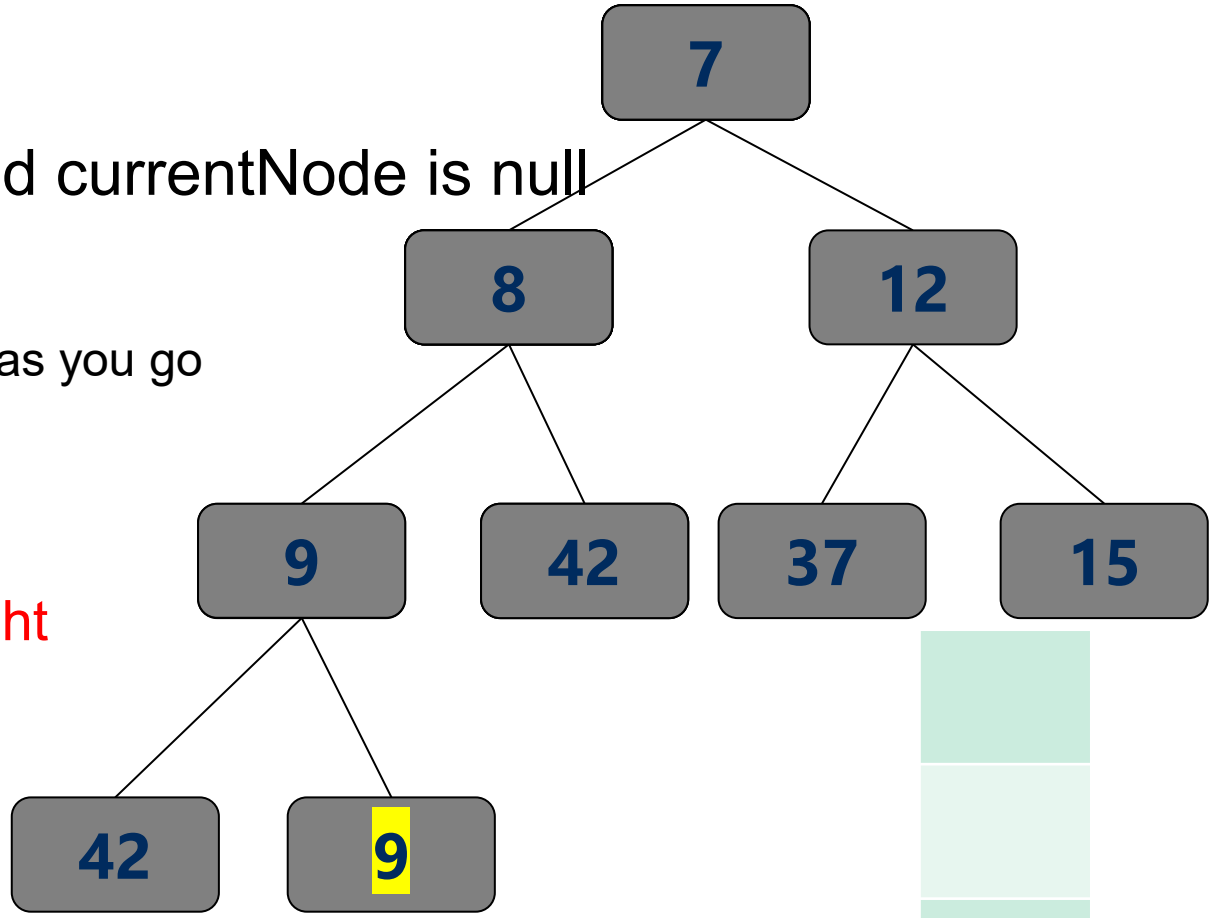


42, 9

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

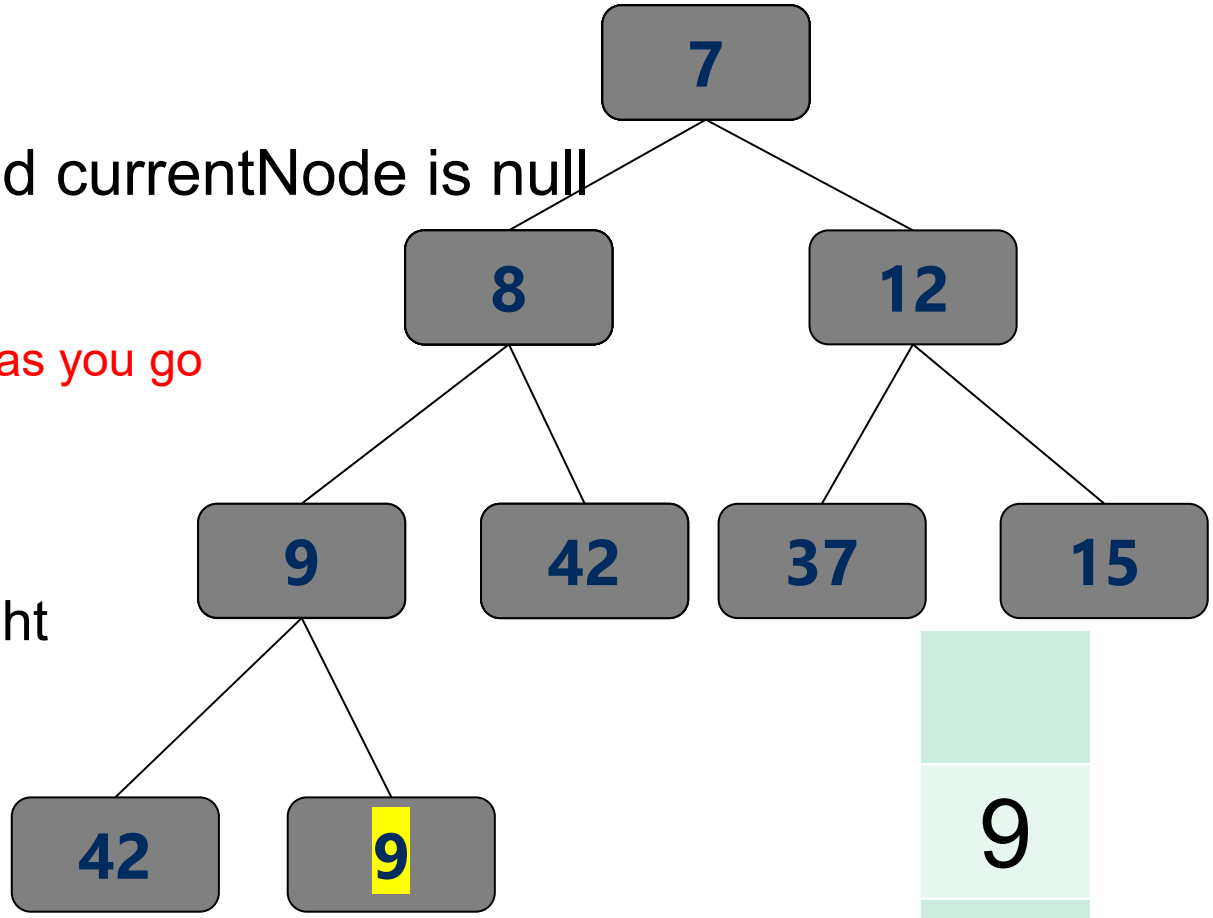


42, 9

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

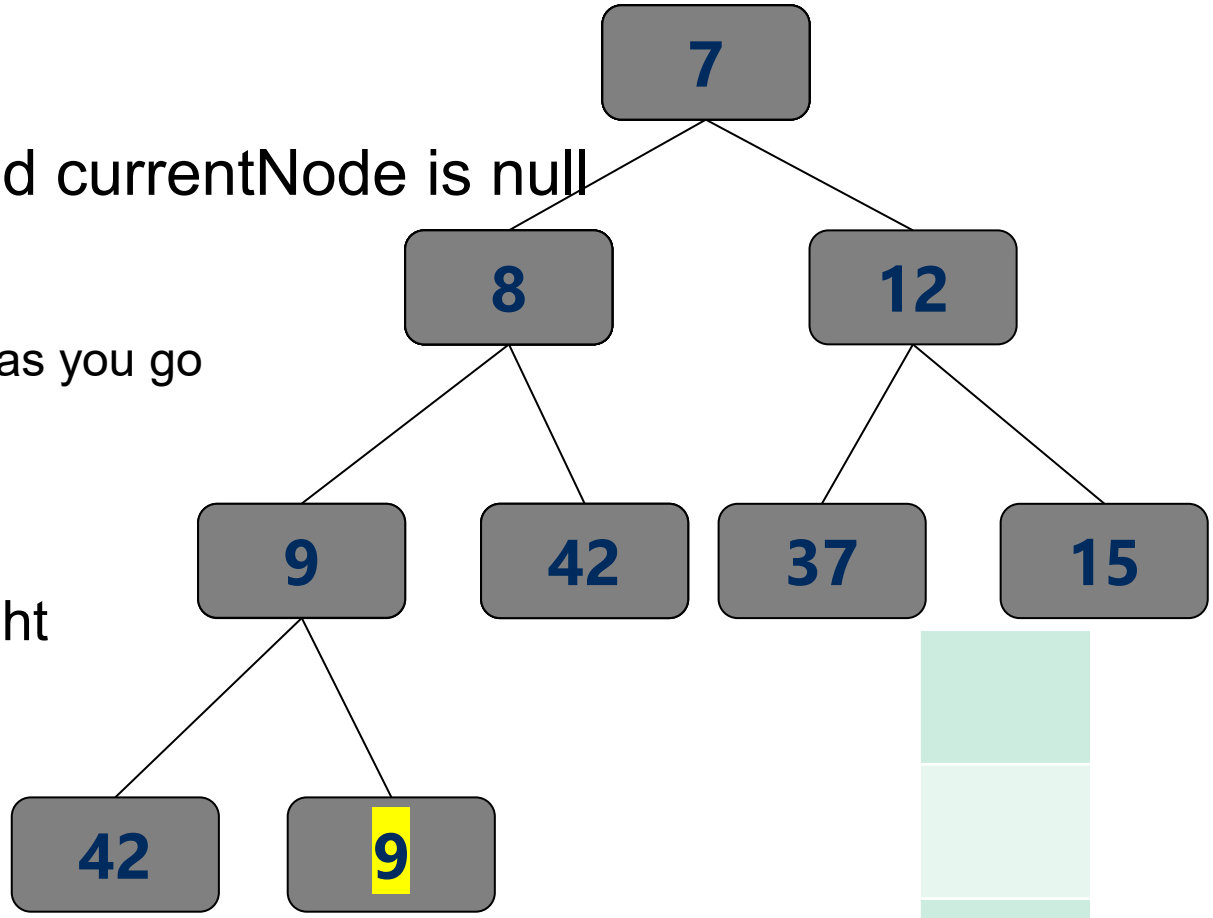


42, 9

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

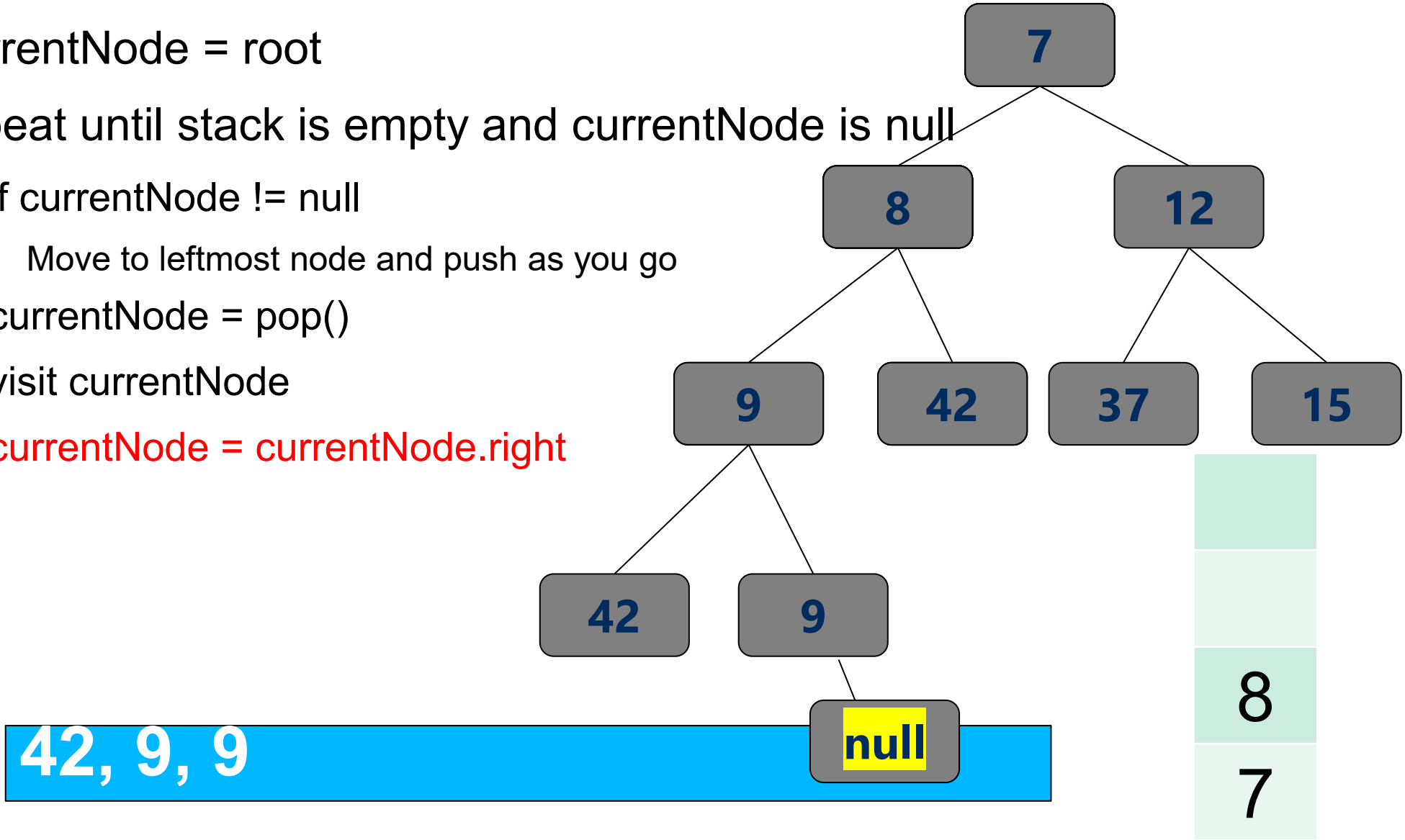


42, 9, 9

Muddiest Points

- **Q: Review iterative inorder traversal**

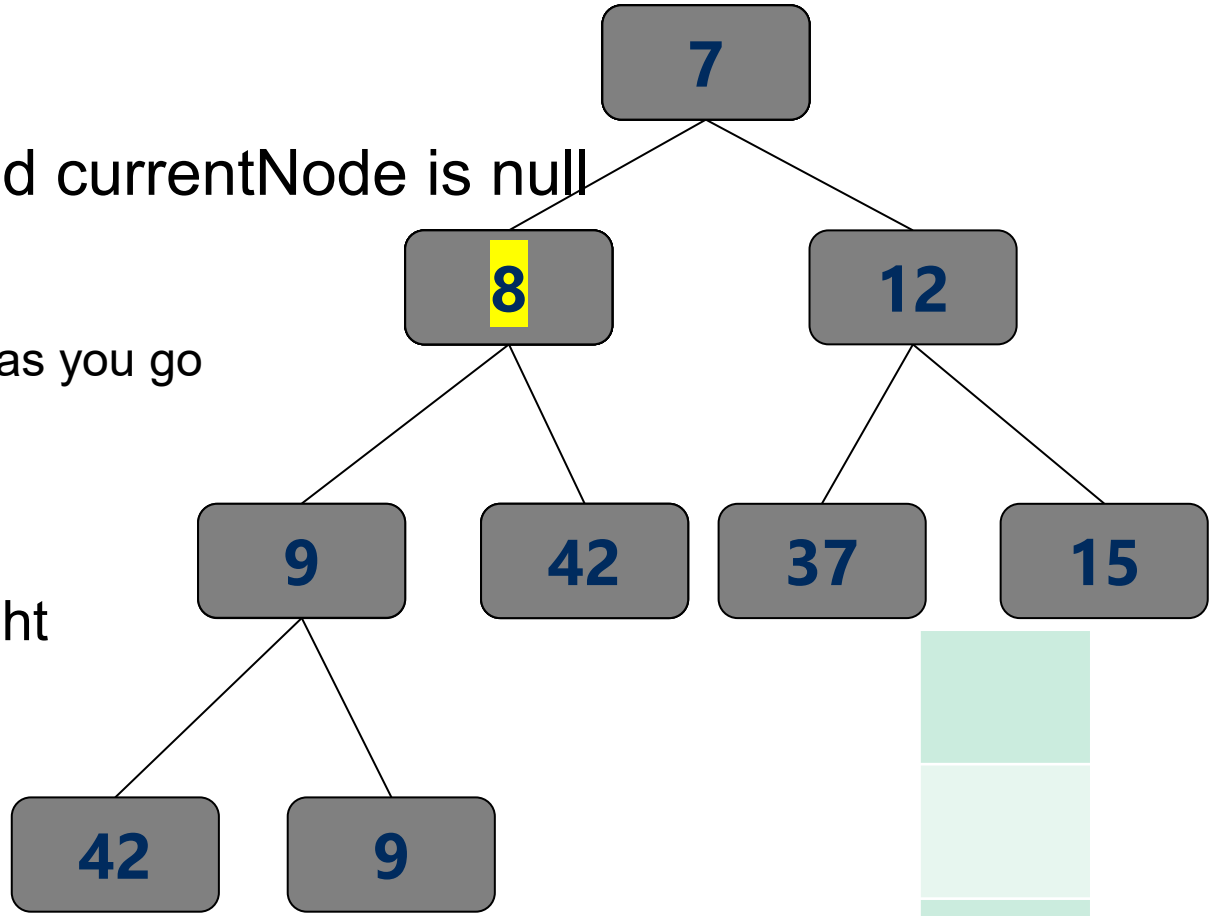
- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

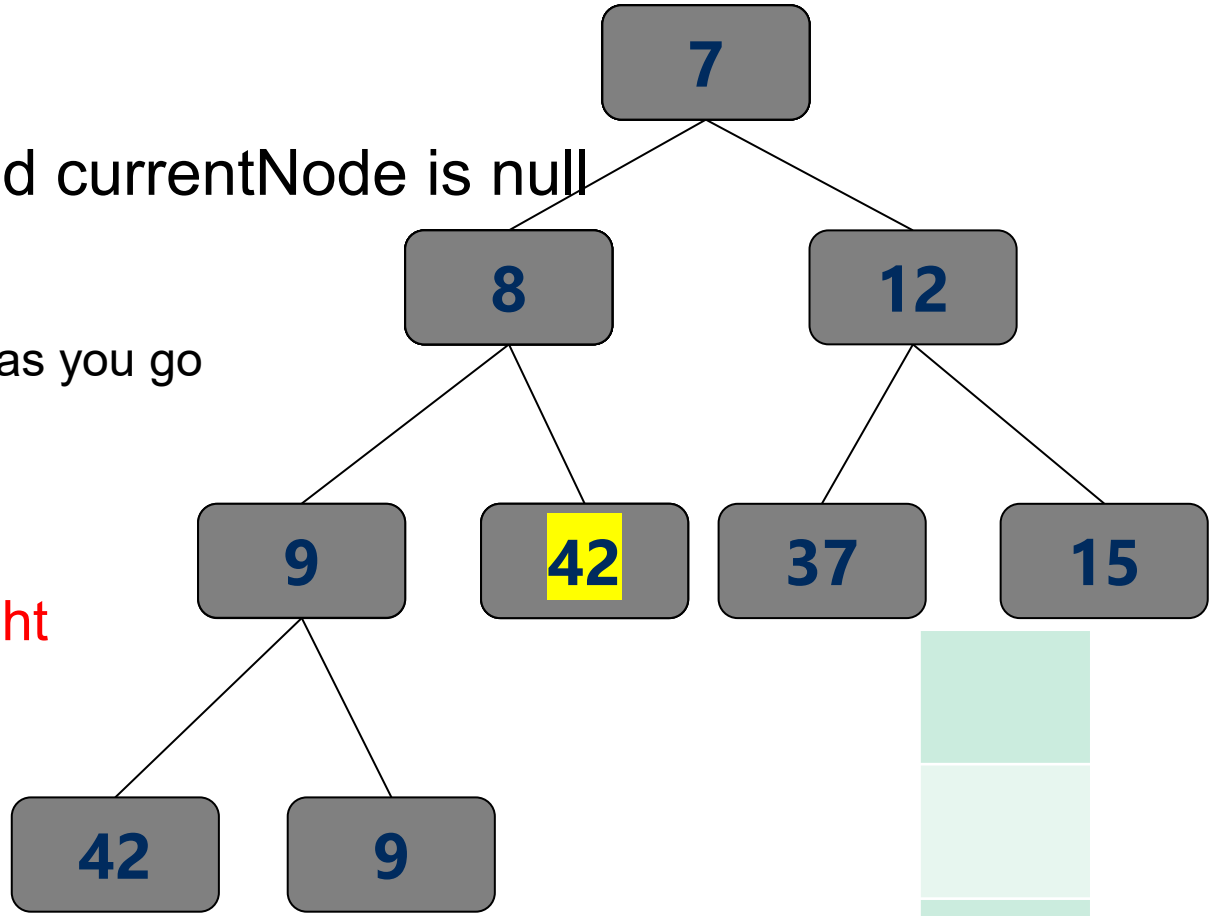


42, 9, 9, 8

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



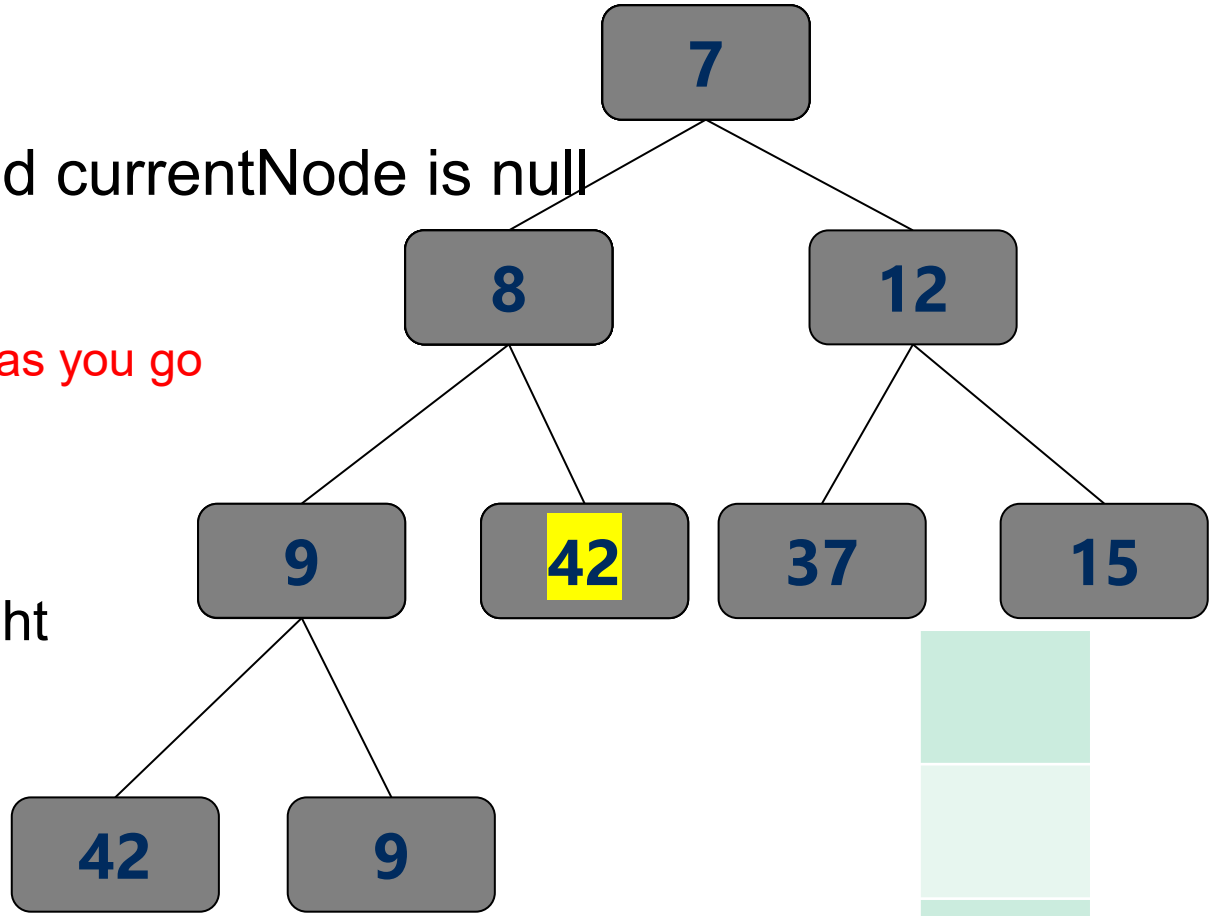
42, 9, 9, 8

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

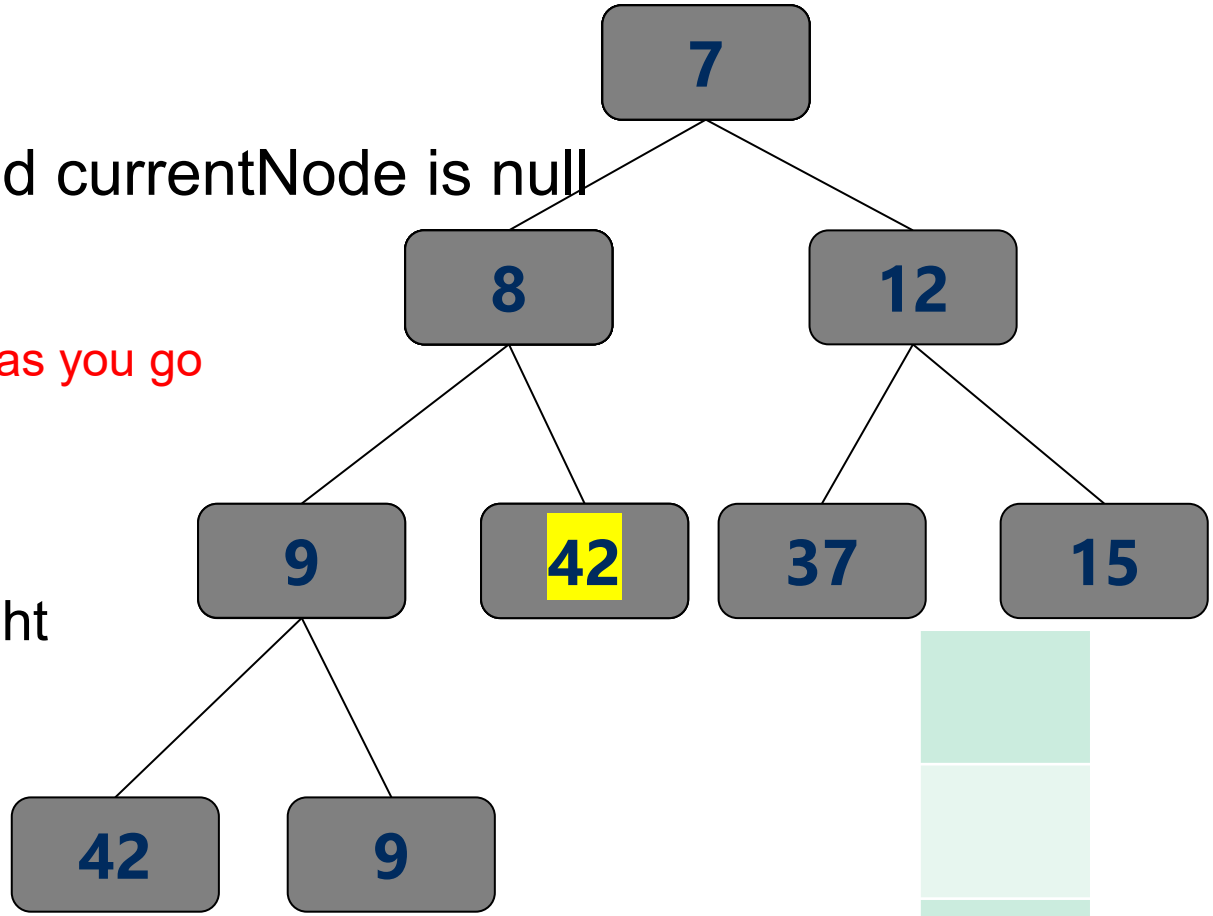


42, 9, 9, 8

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

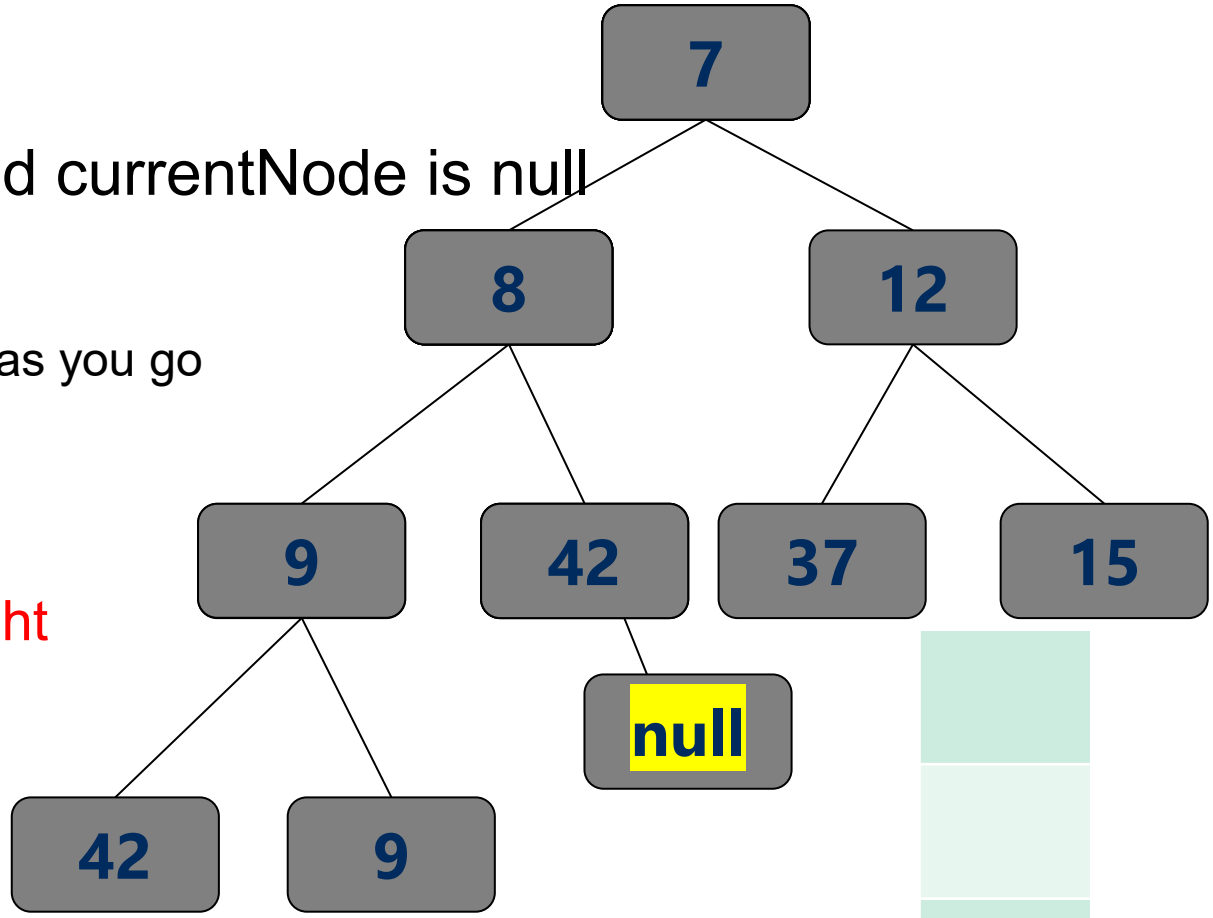


42, 9, 9, 8, 42

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

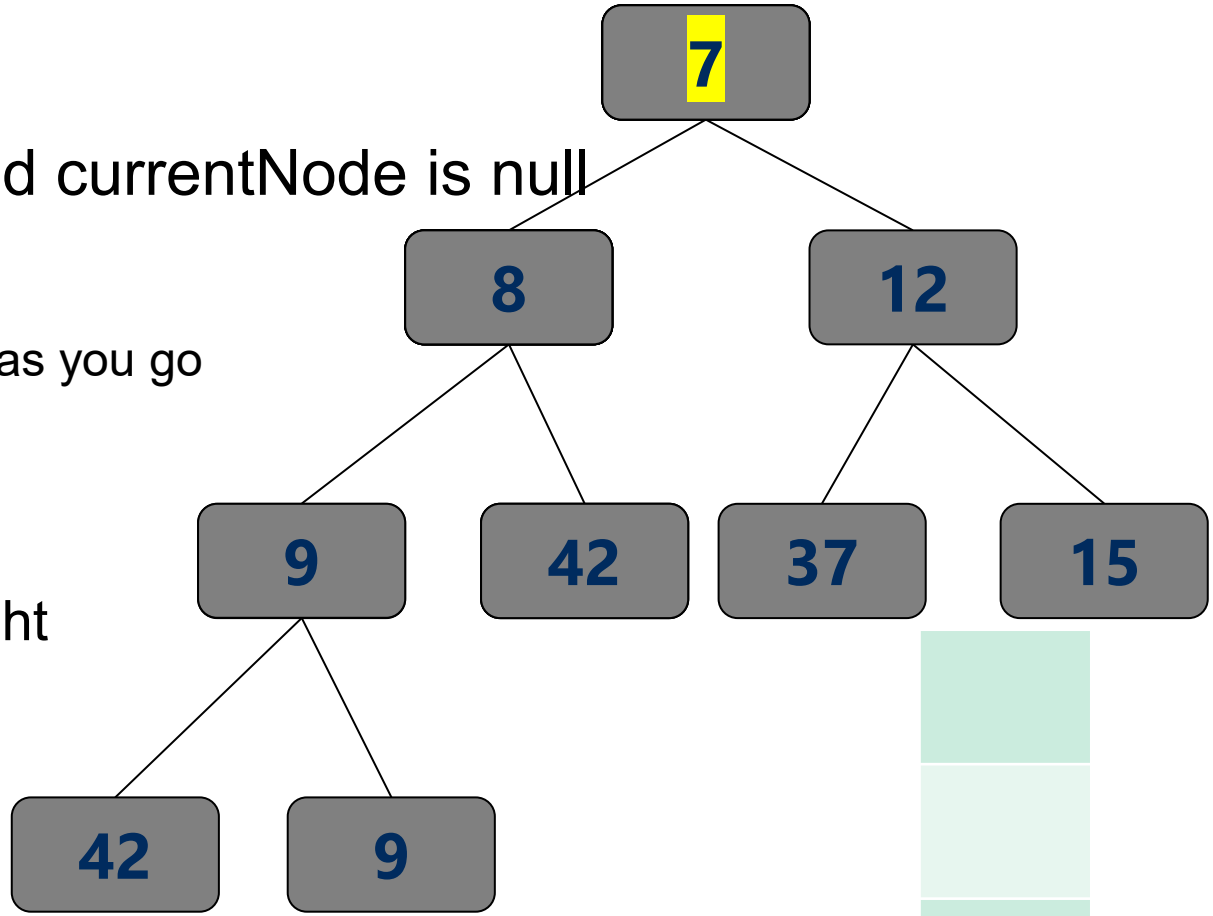


42, 9, 9, 8, 42

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



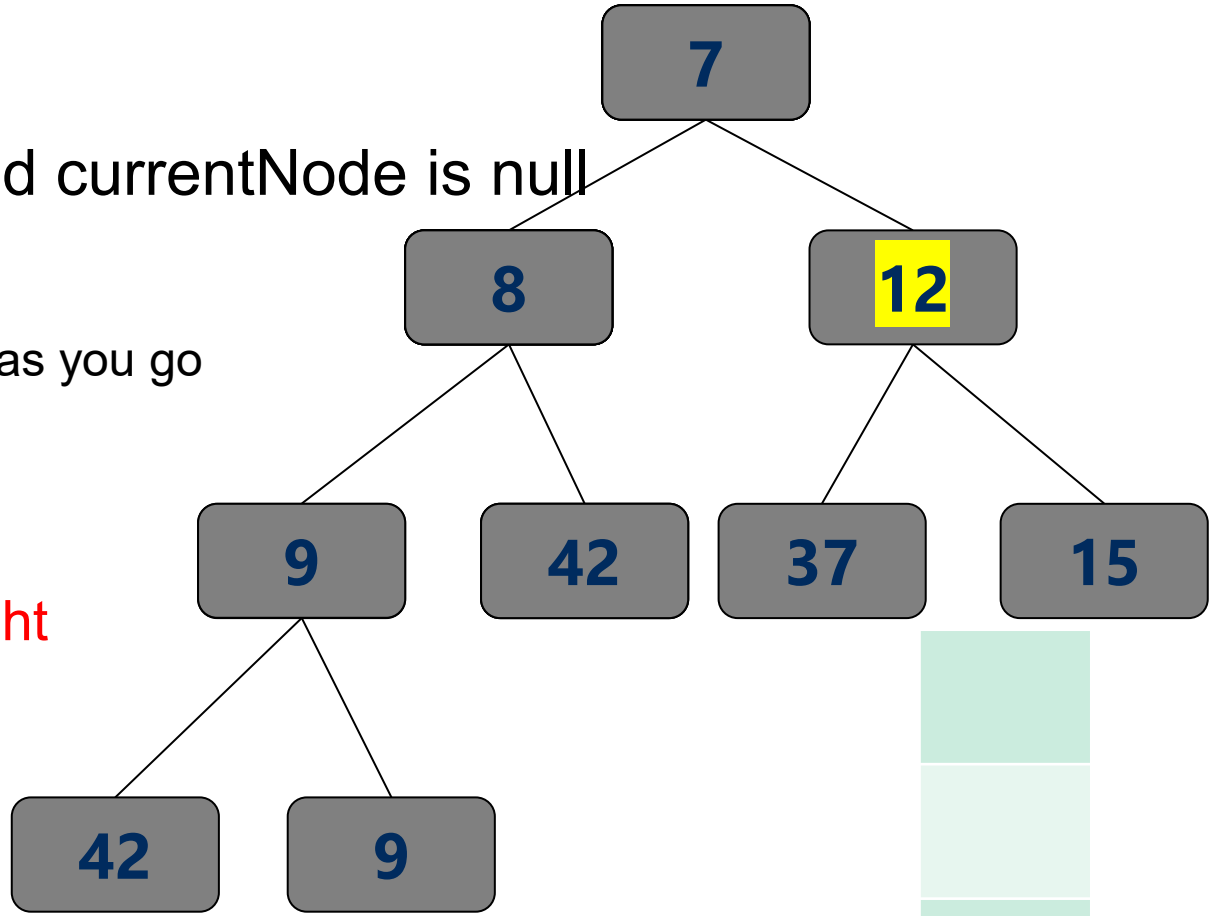
42, 9, 9, 8, 42, 7

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

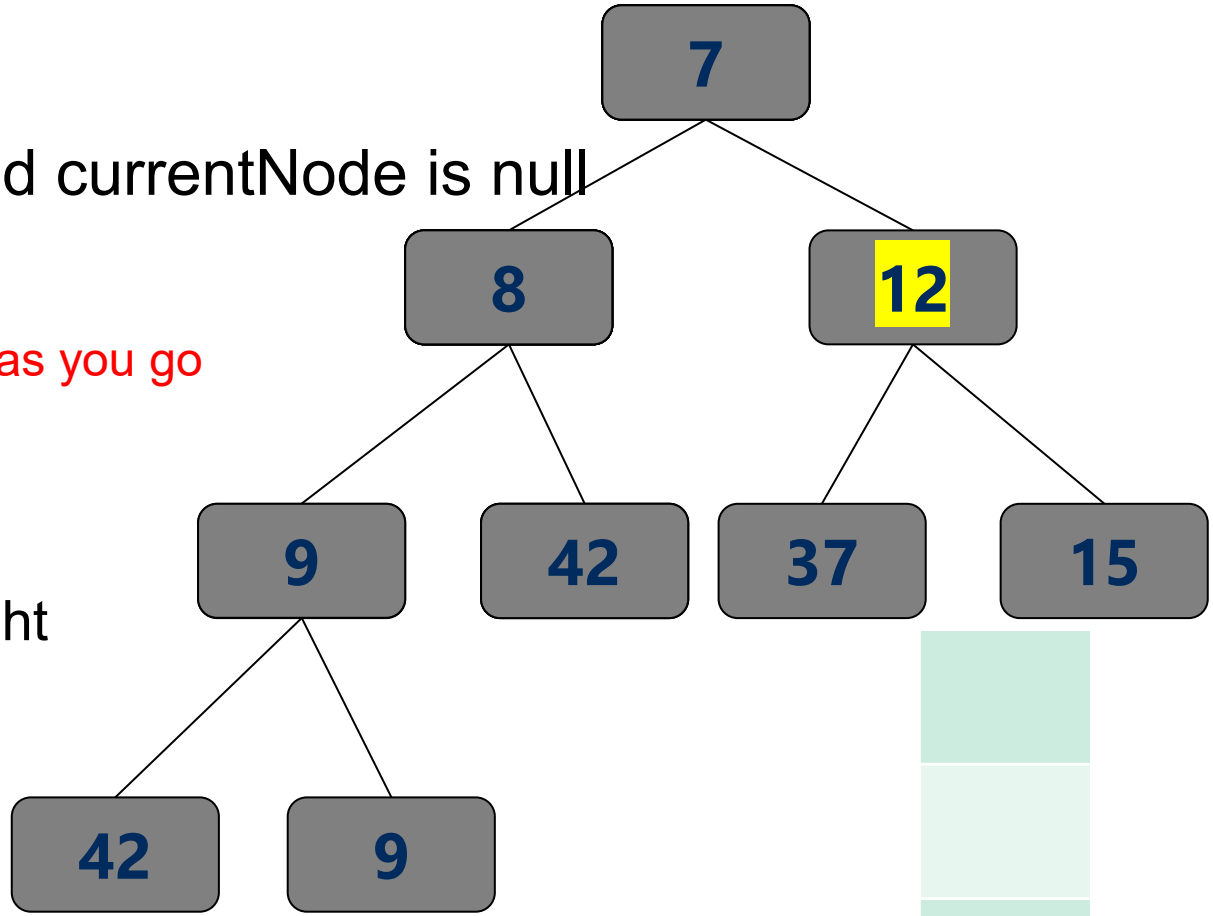


42, 9, 9, 8, 42, 7

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

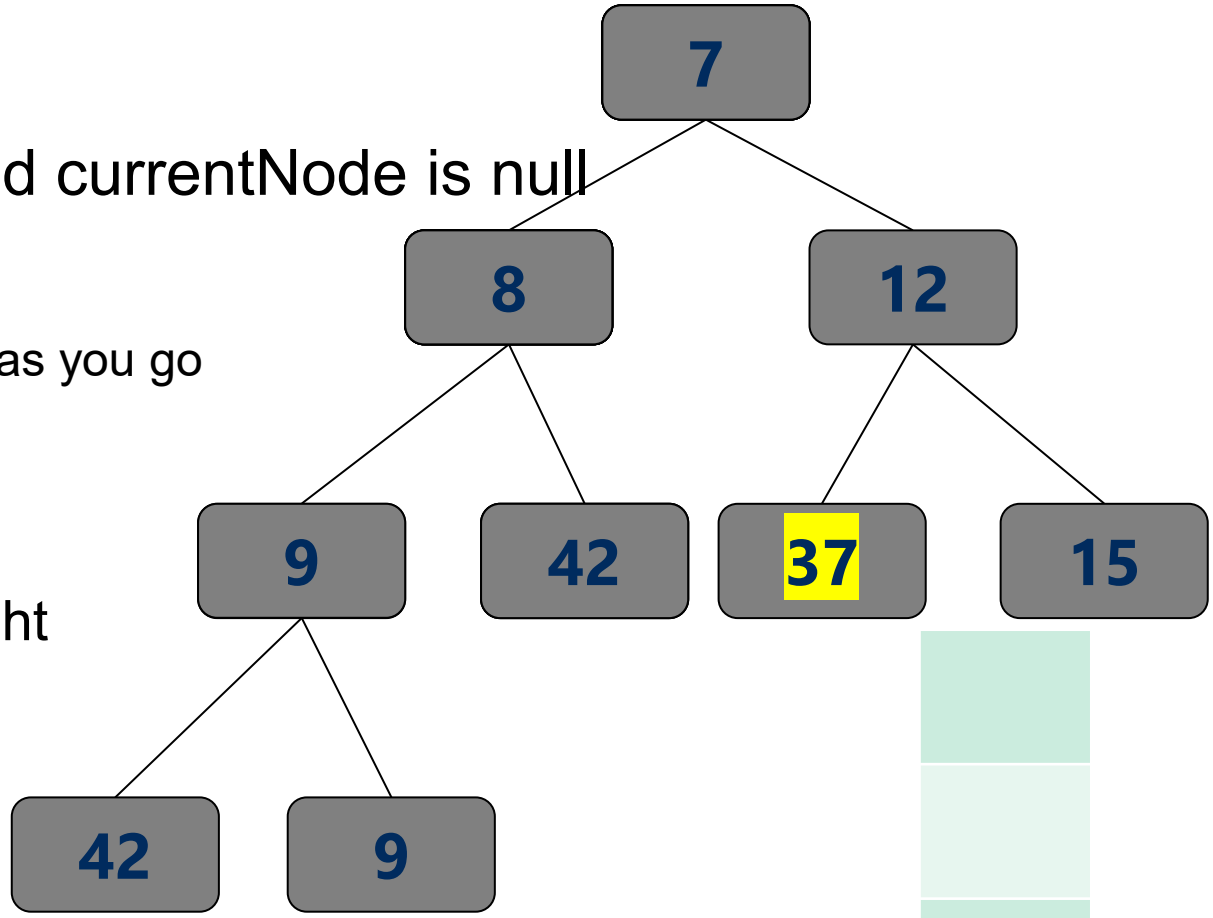


42, 9, 9, 8, 42, 7

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

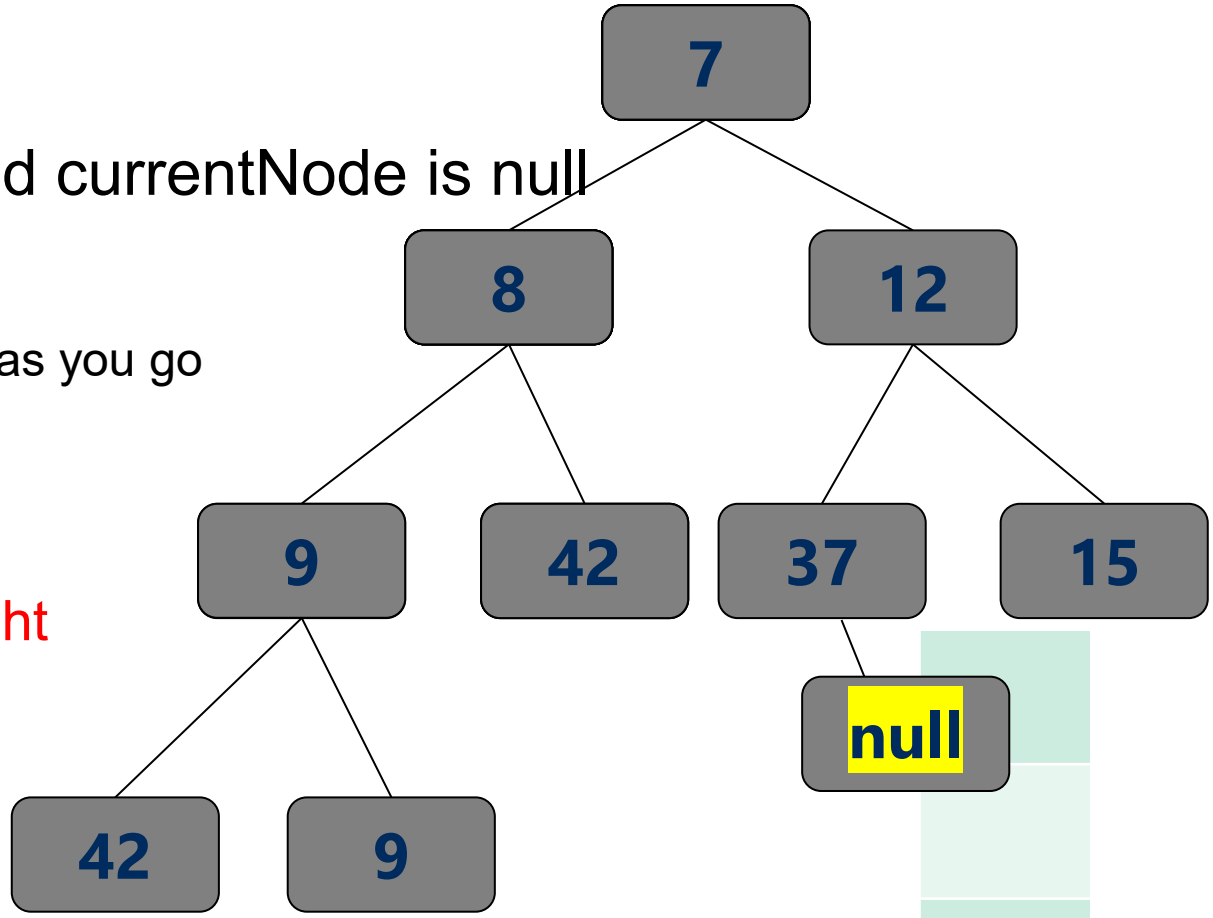


42, 9, 9, 8, 42, 7, 37

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

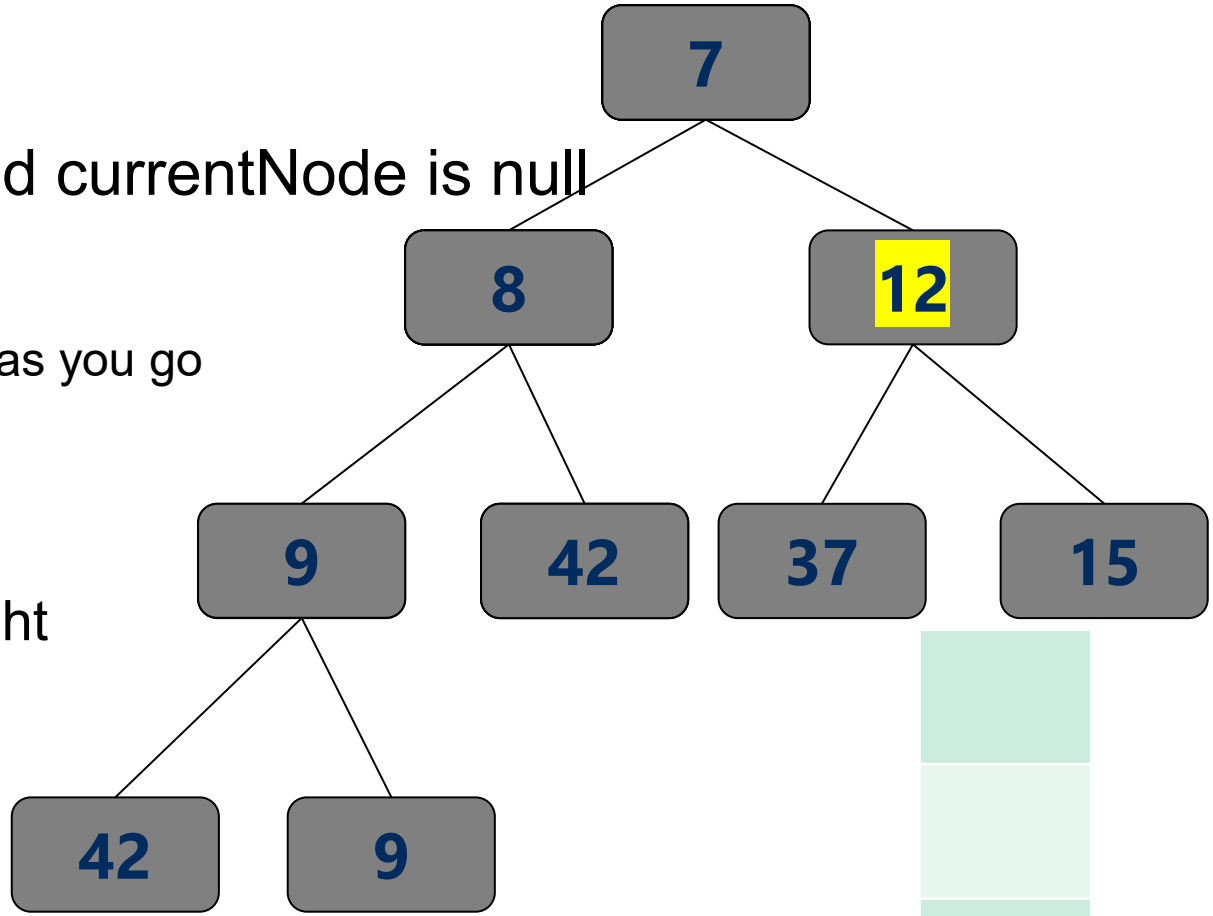


42, 9, 9, 8, 42, 7, 37

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



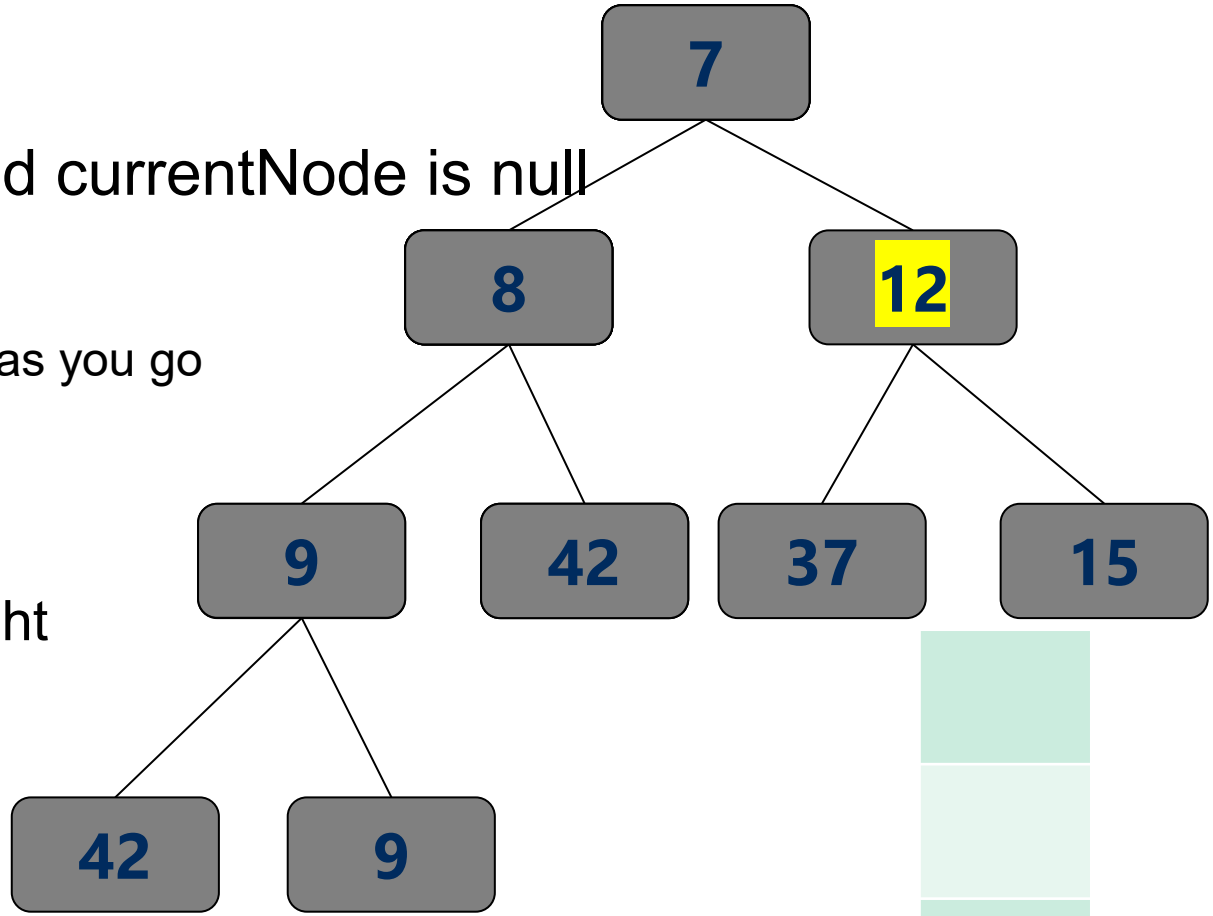
42, 9, 9, 8, 42, 7, 37

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

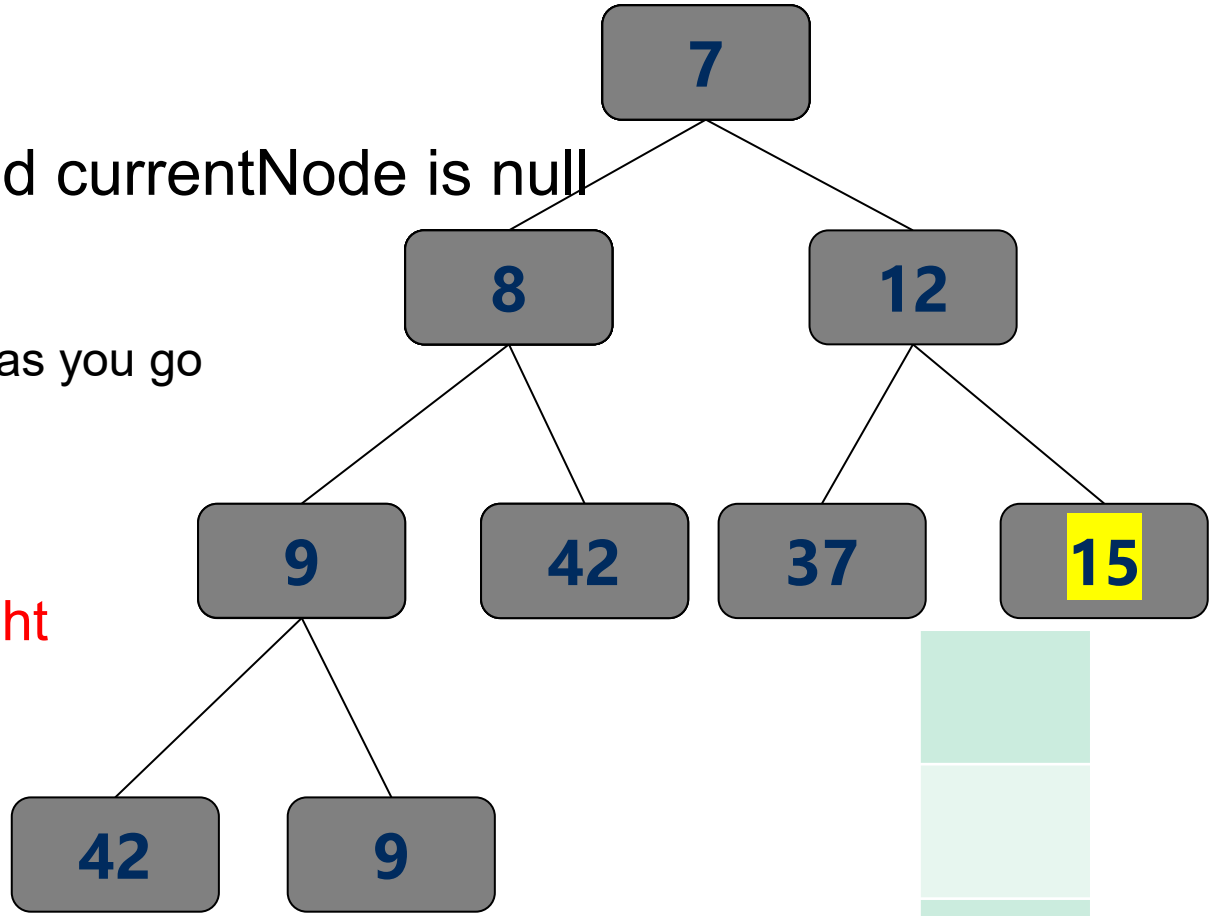


42, 9, 9, 8, 42, 7, 37, 12

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



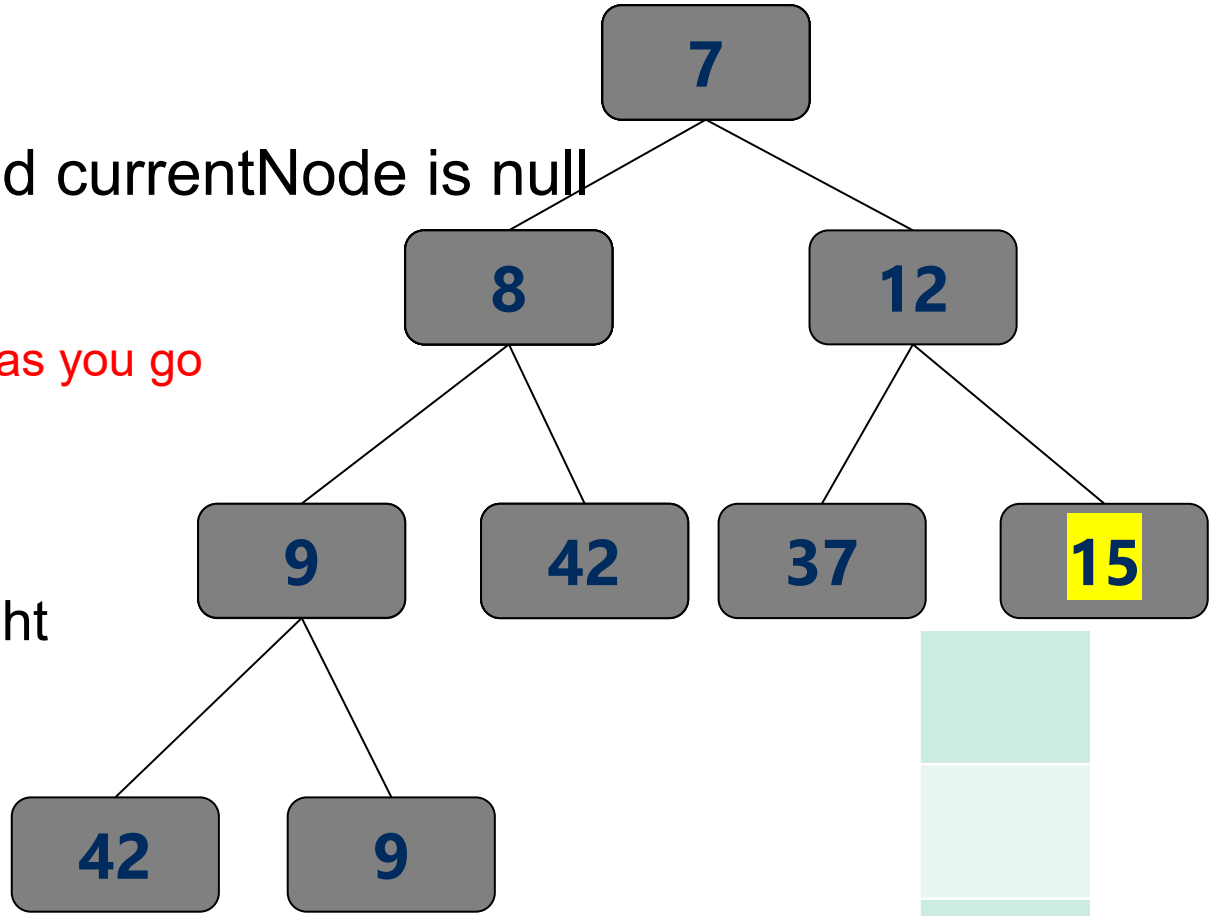
42, 9, 9, 8, 42, 7, 37, 12

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

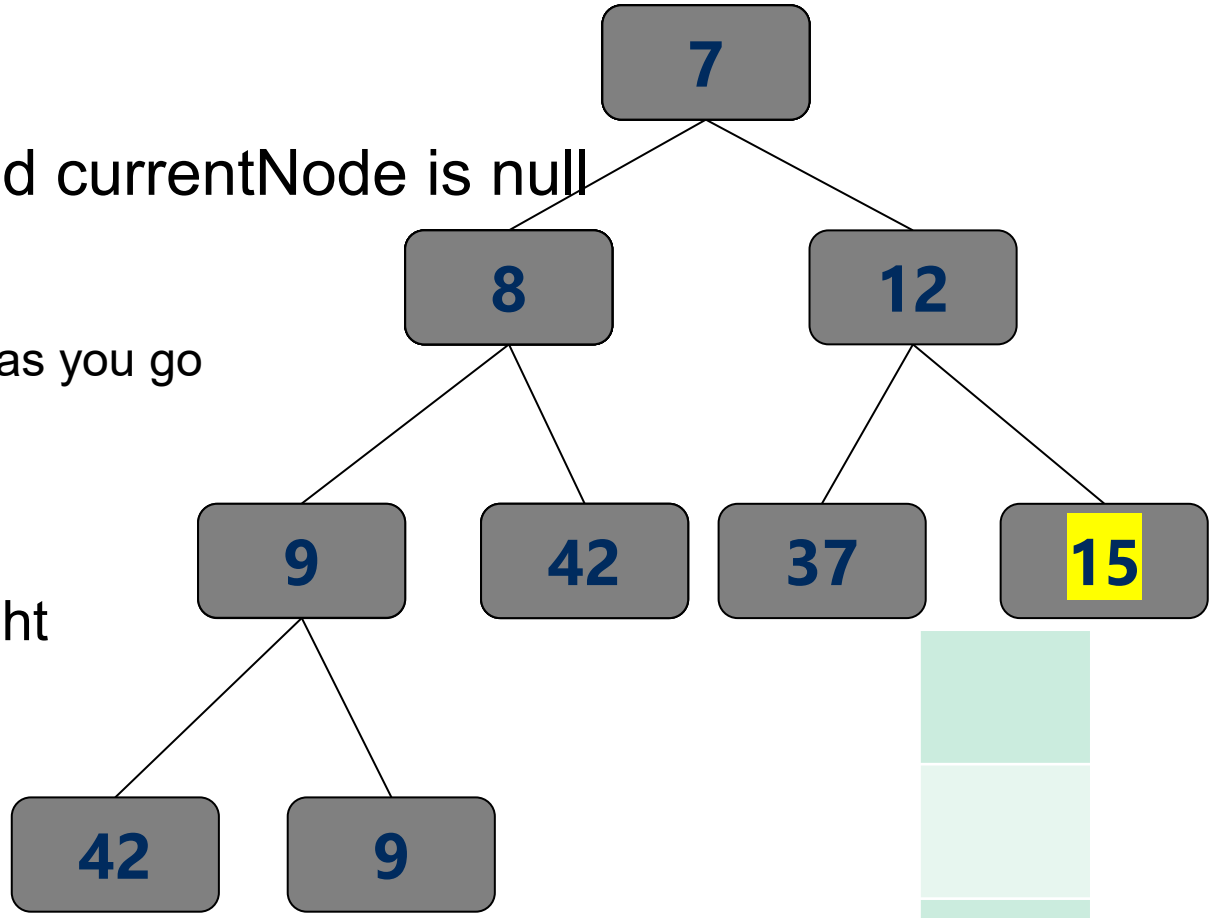


42, 9, 9, 8, 42, 7, 37, 12

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



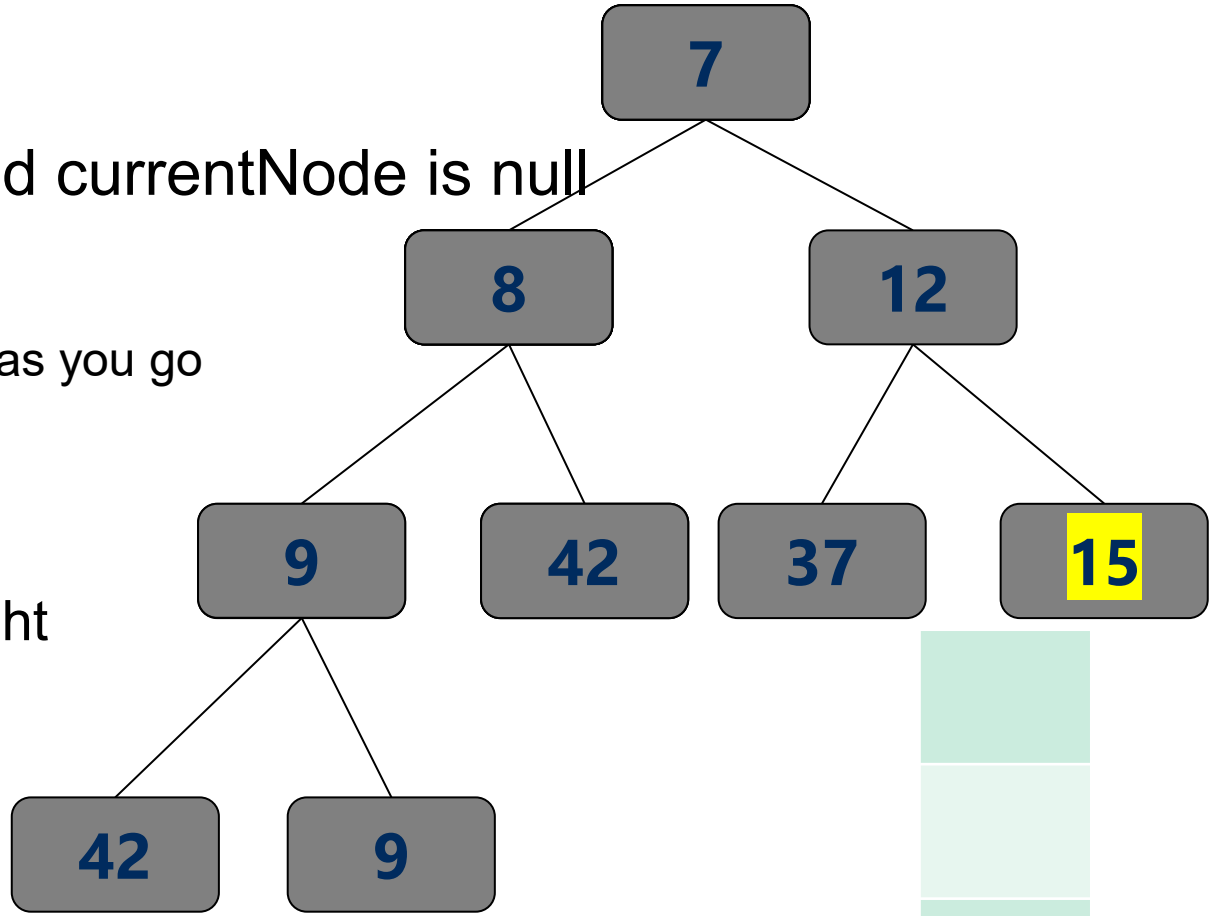
42, 9, 9, 8, 42, 7, 37, 12

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



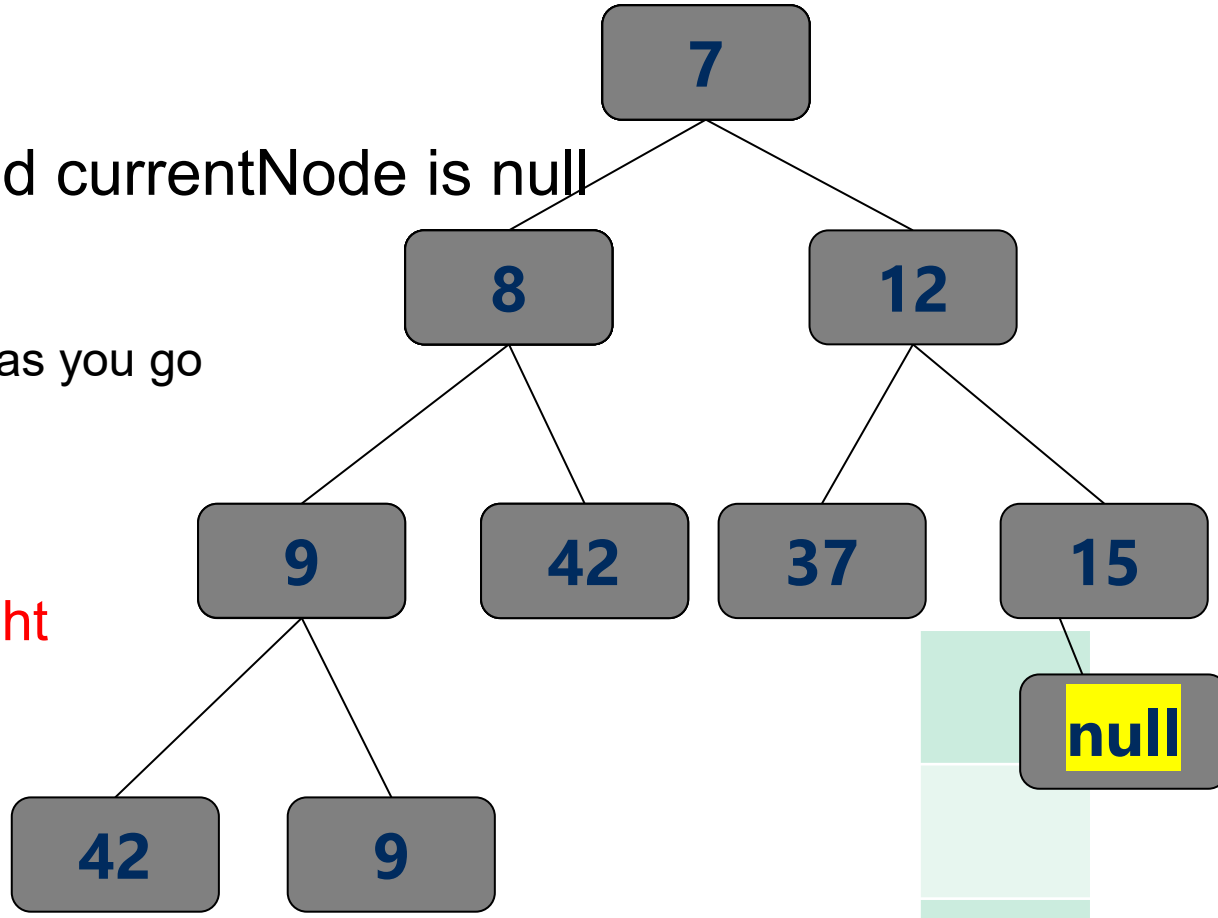
42, 9, 9, 8, 42, 7, 37, 12, 15

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right



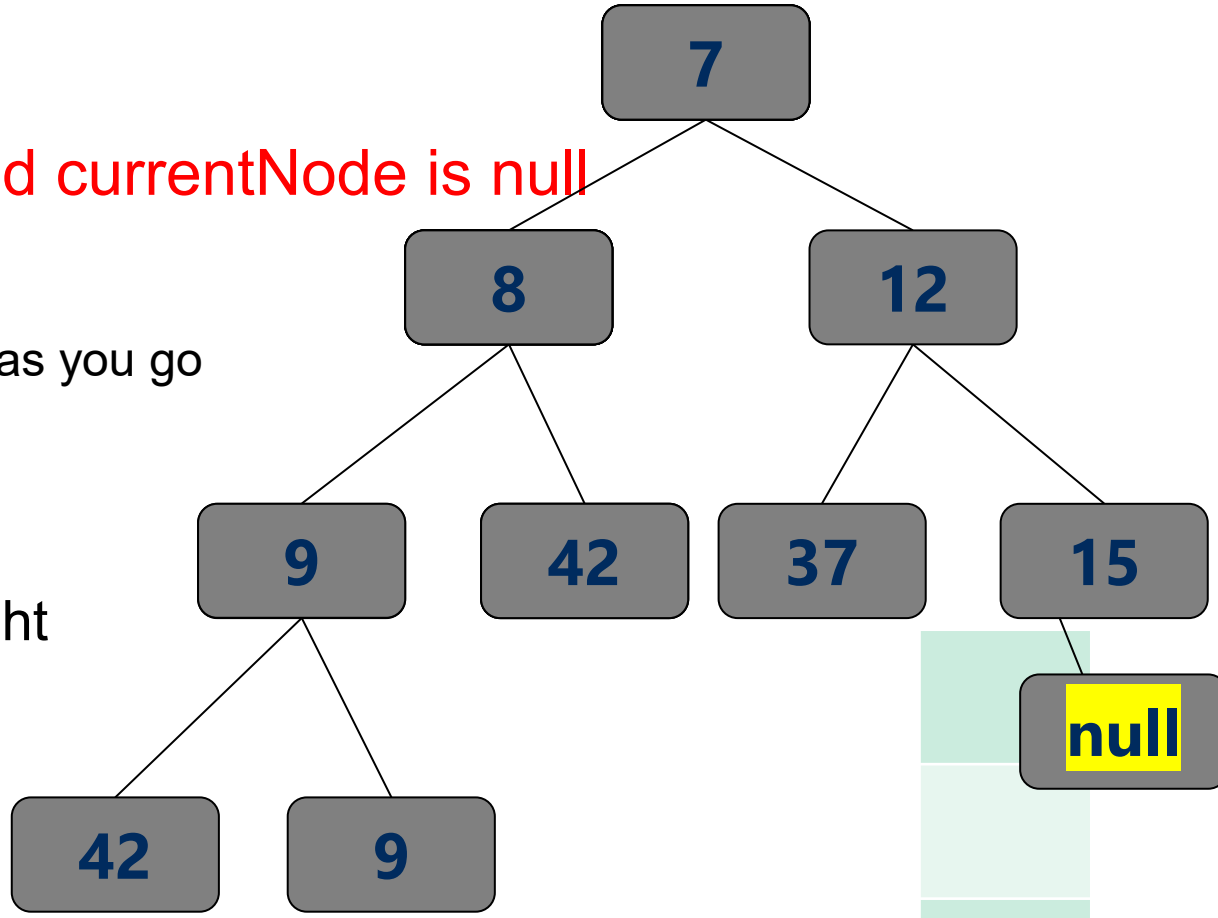
42, 9, 9, 8, 42, 7, 37, 12, 15

Stack

Muddiest Points

- **Q: Review iterative inorder traversal**

- currentNode = root
- repeat until stack is empty and currentNode is null
  - if currentNode != null
    - Move to leftmost node and push as you go
  - currentNode = pop()
  - visit currentNode
  - currentNode = currentNode.right

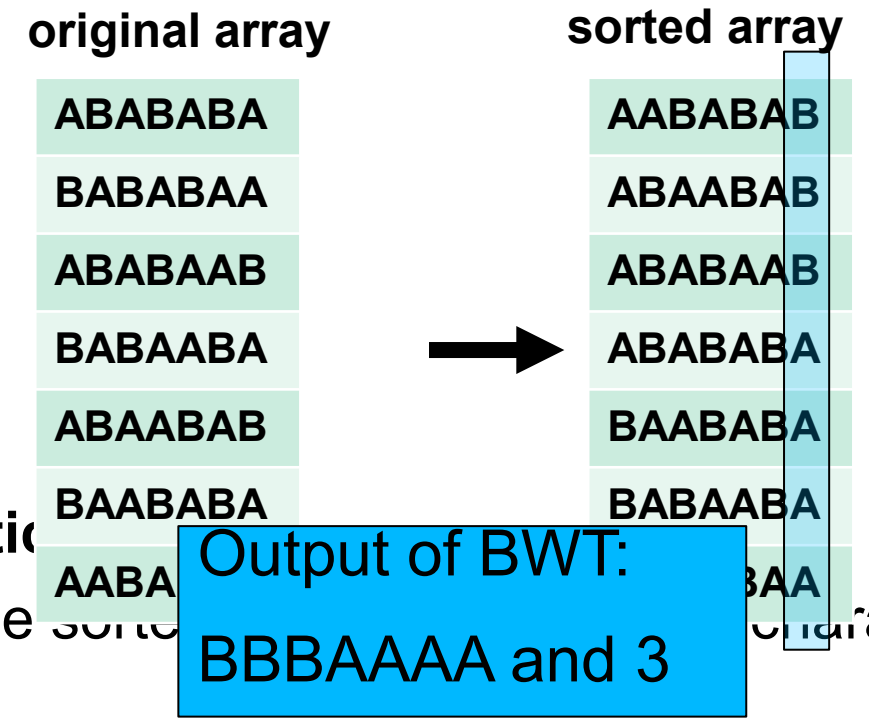


42, 9, 9, 8, 42, 7, 37, 12, 15

Stack

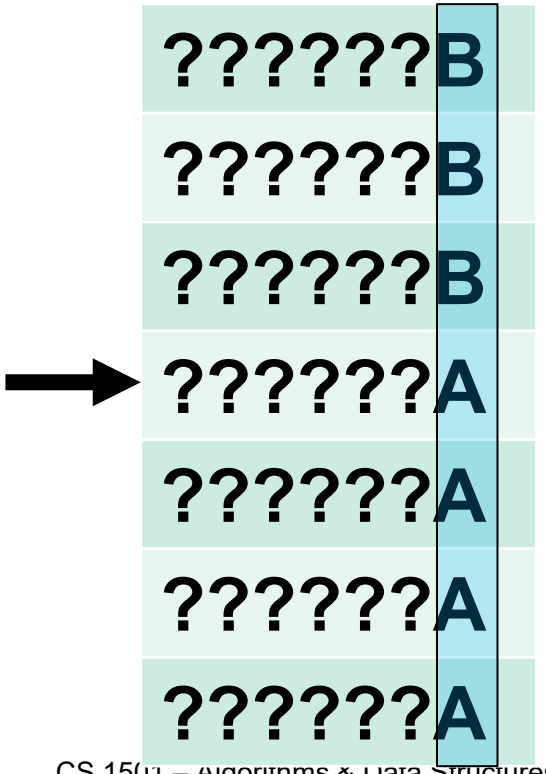
Burrows-Wheeler Transform Example

- ABABABA
- **Step 1: Build an array of 7 strings, each a circular rotation of the original by one character**
- ABABABA
- BABABAA
- ABABAAB
- BABAABA
- ABAABAB
- BAABABA
- AABABAB
- **Step 2: Sort the array alphabeti**
- **Notice that** the first column of the sorte… characters as the last column
  - all columns have the same set of letters
- **Step 3: Output the last column of the sorted array and the index of the input string in the sorted array**



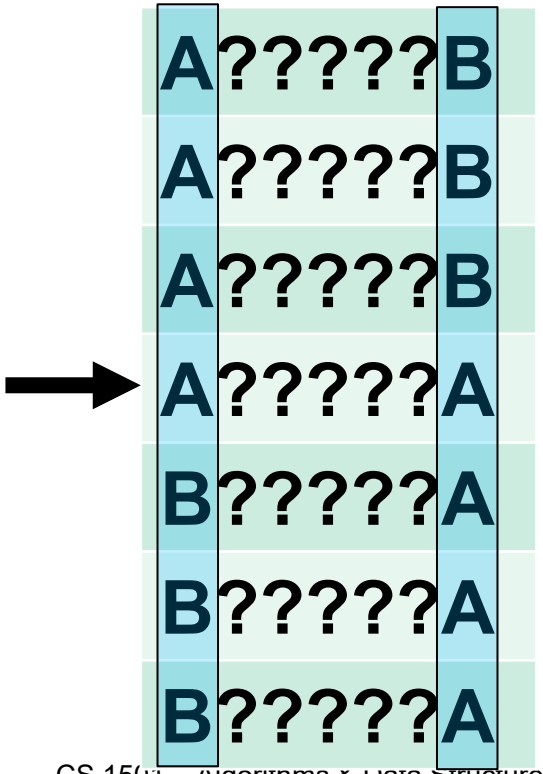
Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
  - BBBAAAA → AAAABBB
  - The first column of the sorted array has the same characters as the last column
    - but in sorted order



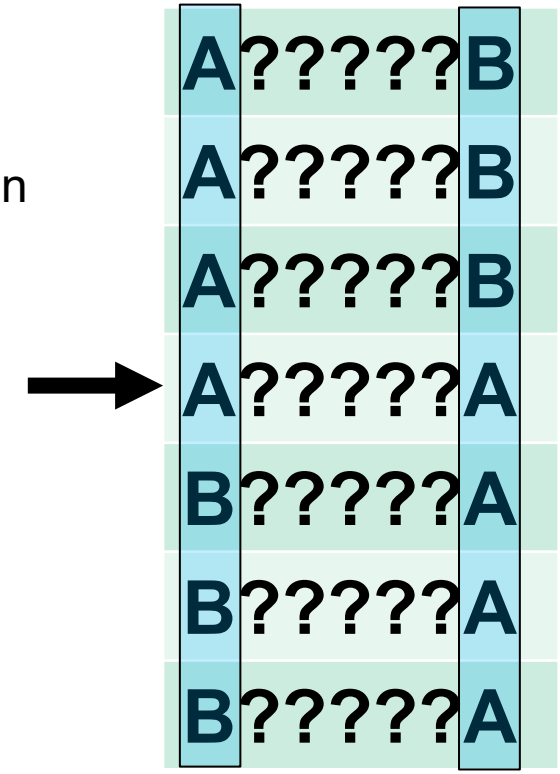
Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 1: Sort the encoded string**
  - BBBAAAA → AAAABBB
  - This gives us the first column of the sorted array



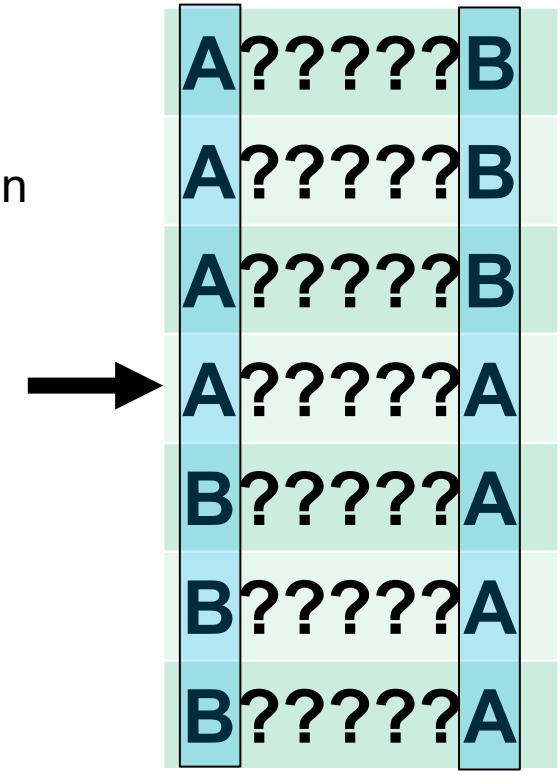
Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



Burrows-Wheeler Transform Decoding

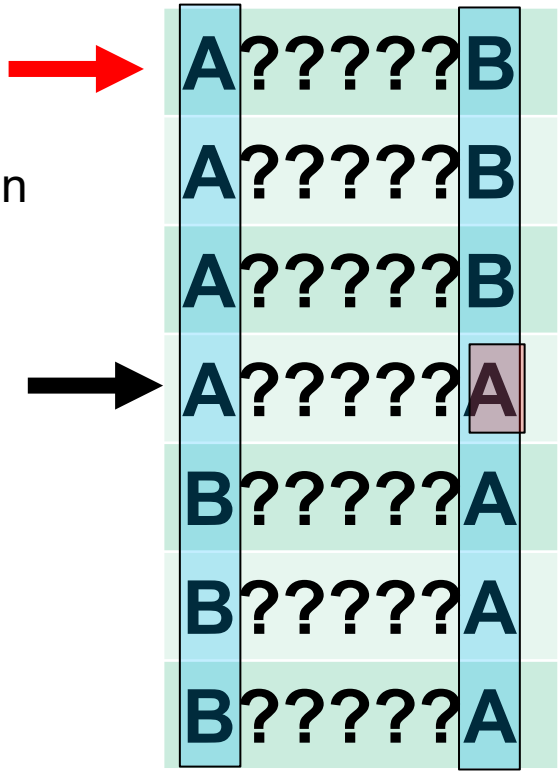
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
-
-
-
-
-
-
-

Burrows-Wheeler Transform Decoding

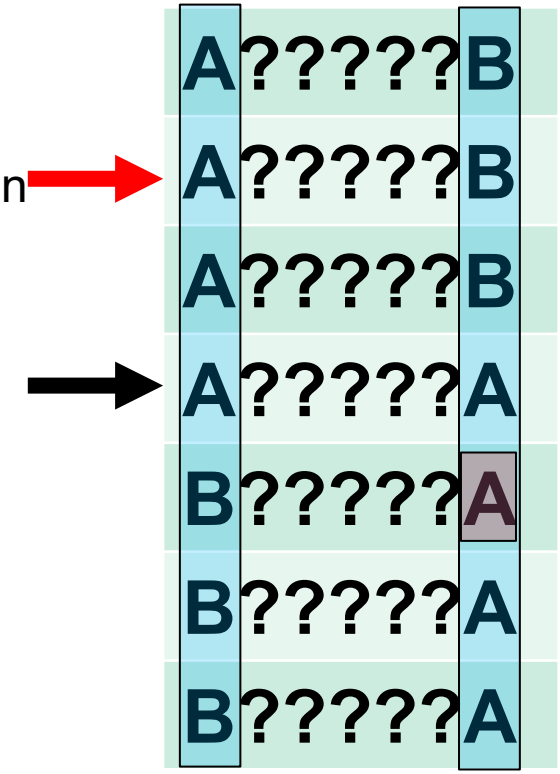
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
-
-
-
-
-
-

Burrows-Wheeler Transform Decoding

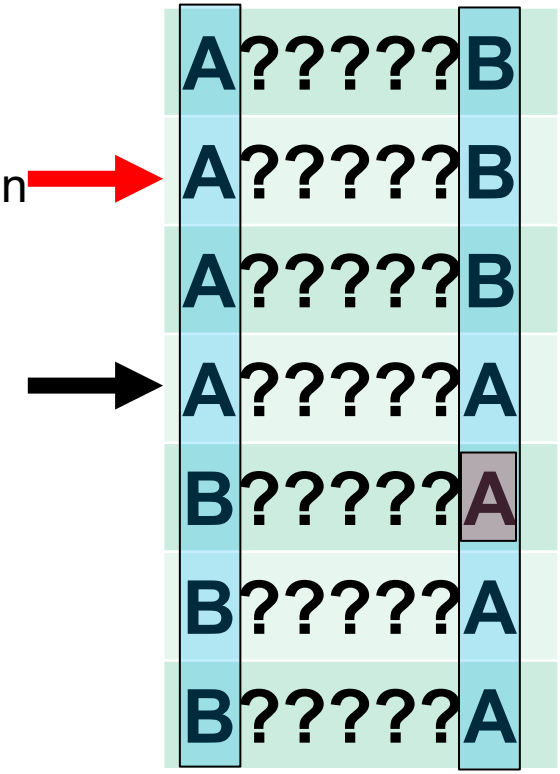
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
-
-
-
-
-

Burrows-Wheeler Transform Decoding

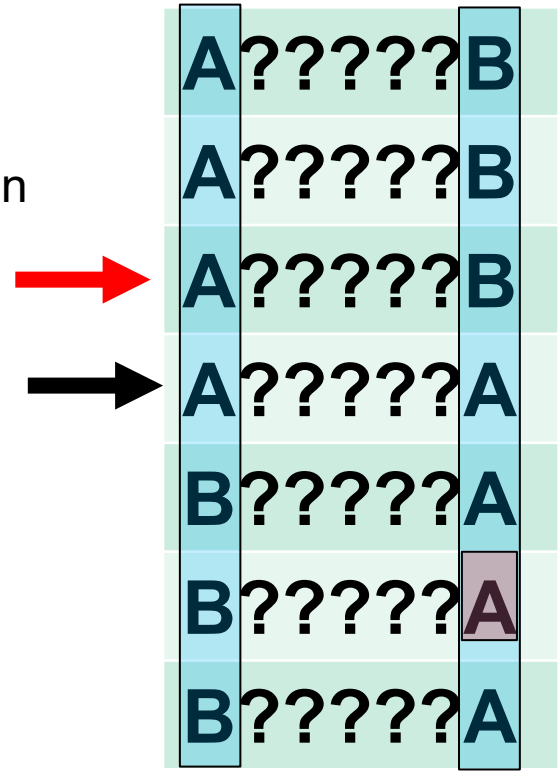
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
-
-
-
-
-

Burrows-Wheeler Transform Decoding

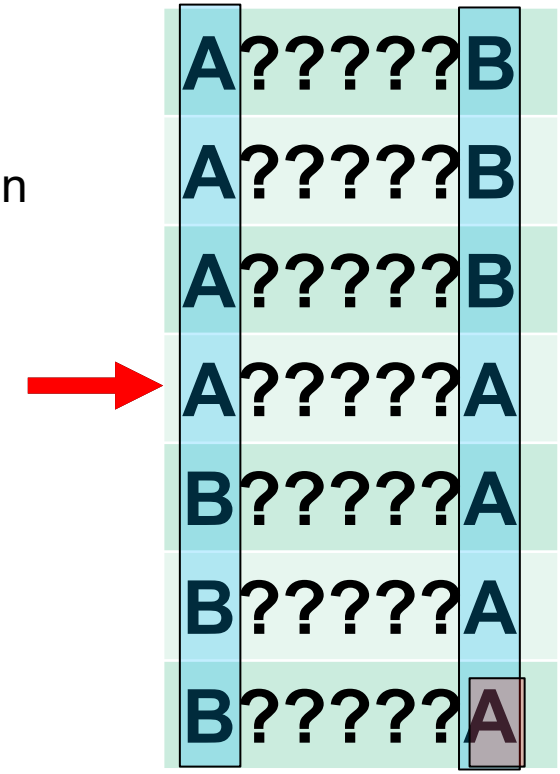
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
5
-
-
-
-

Burrows-Wheeler Transform Decoding

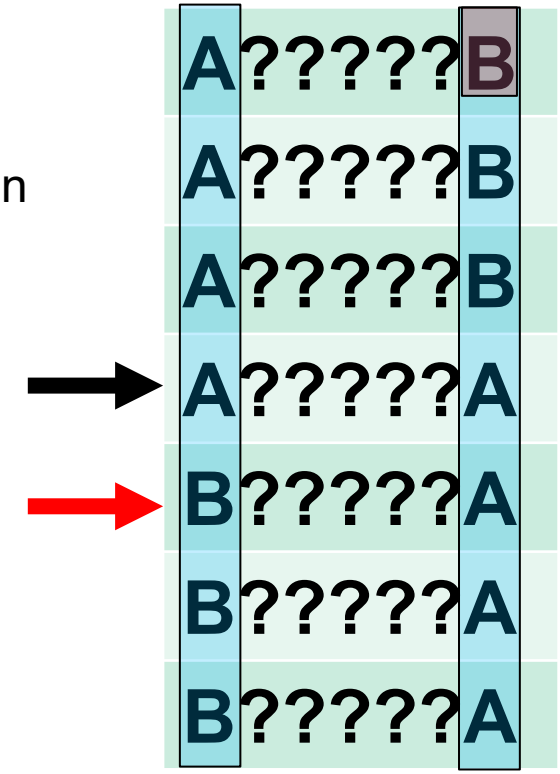
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
5
6
-
-
-

Burrows-Wheeler Transform Decoding

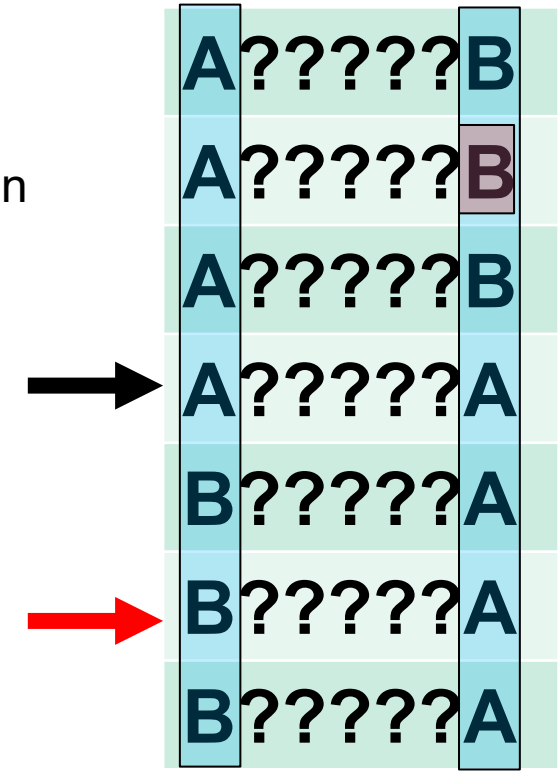
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
5
6
0
-
-

Burrows-Wheeler Transform Decoding

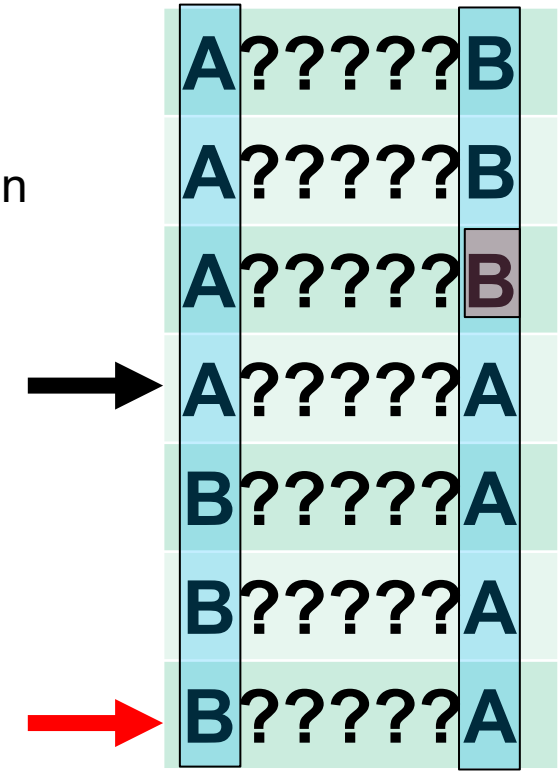
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
5
6
0
1
-

Burrows-Wheeler Transform Decoding

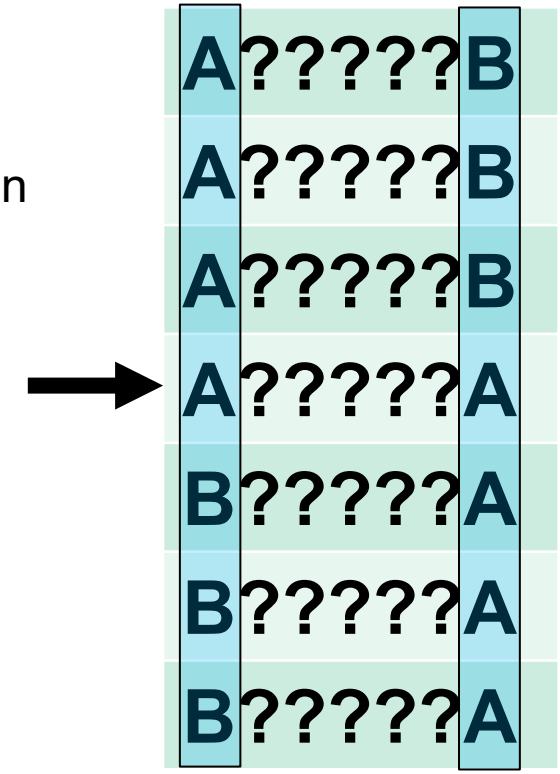
- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column



next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 2: Fill an array next[]**
  - defined for each entry in the sorted array
  - tells us the index in sorted array of the next string in the original array
  - Scan through the first column
    - for each row *i* holding character *c*
    - next[i] = first unassigned index of *c* in the last column
- Why does that work?
  - first character of a string becomes the last character in the next string in the original order

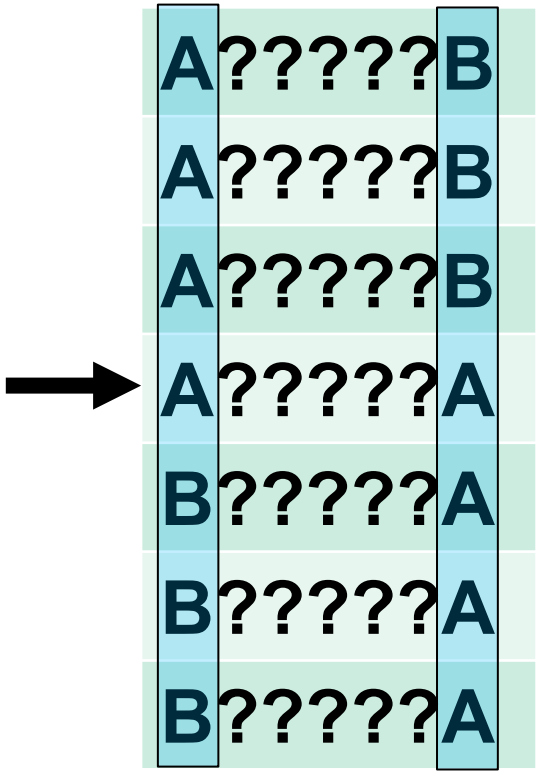


next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?

A??????

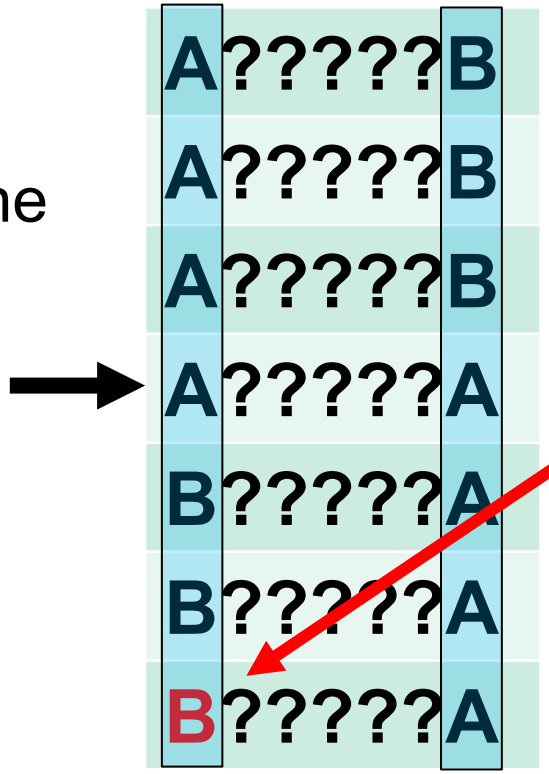


next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[3]

AB?????

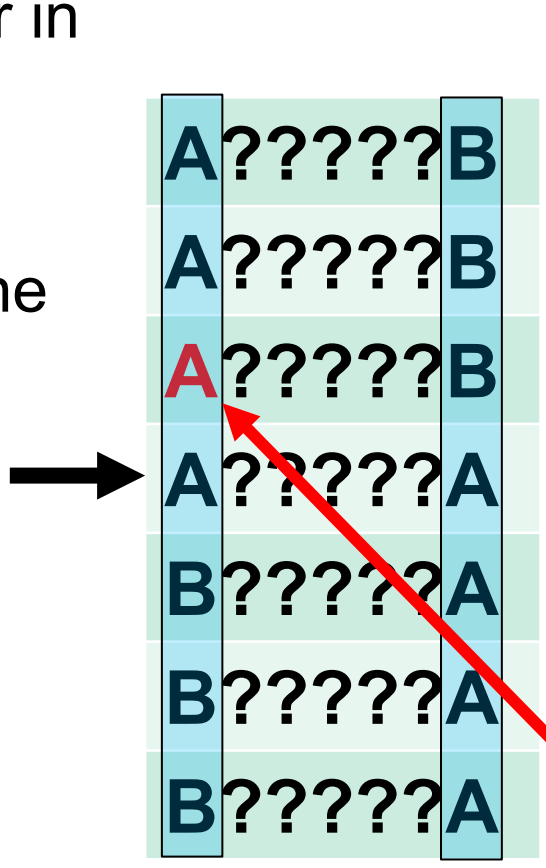


next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[6]

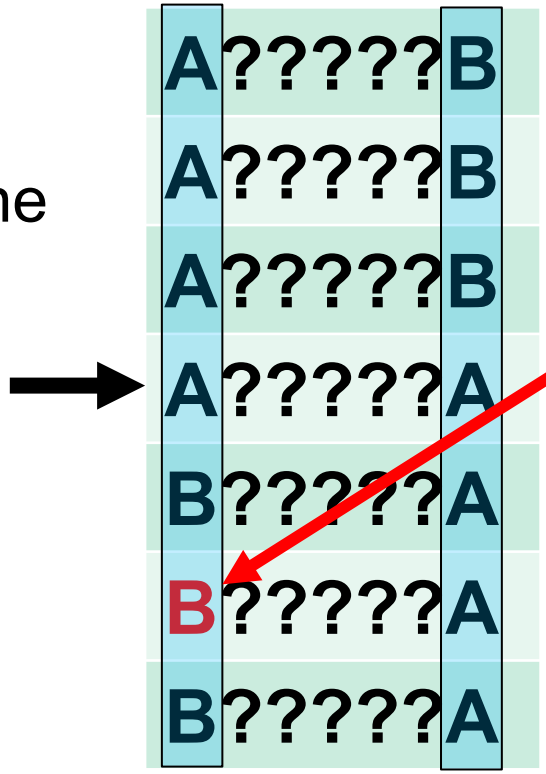
ABA????



Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[2]

ABAB???

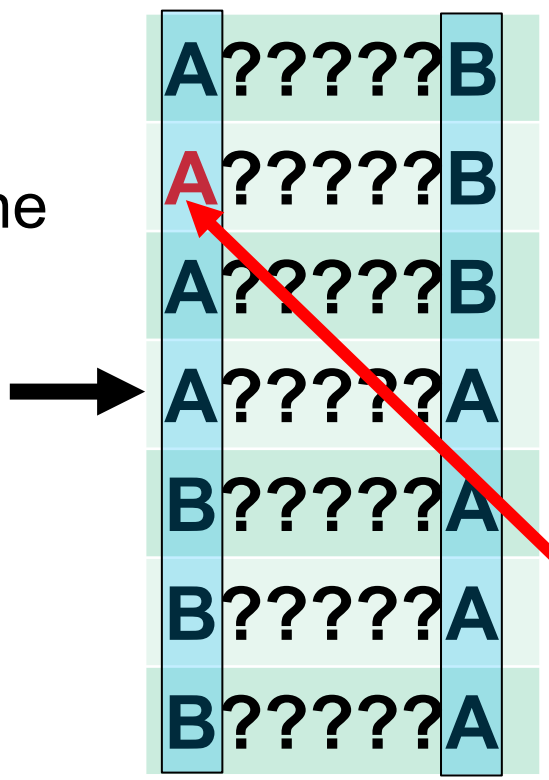


next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[5]

ABABA??



next
3
4
5
6
0
1
2

Burrows-Wheeler Transform Decoding

- Output of BWT:
  - BBBAAAA and 3
- How can we recover ABABABA?
- **Step 3: Recover the input string using the next[] array**
- We can conclude that A is the first character in the input string
  - why?
- The next character is the first character of the next string in the original order
  - first character in string at next[5]

	next
A?????B	3
A?????B	4
A?????B	5
A?????A	6
B?????A	0
B?????A	1
B?????A	2

ABABABA

Problem of the Day

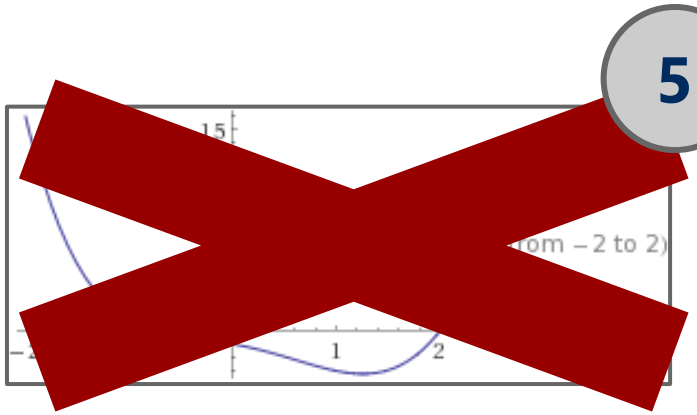
- **Input**: A file containing LinkedIn (LI) Connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - e.g., 1st connection?, 2nd connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These accounts are called ***articulation points***

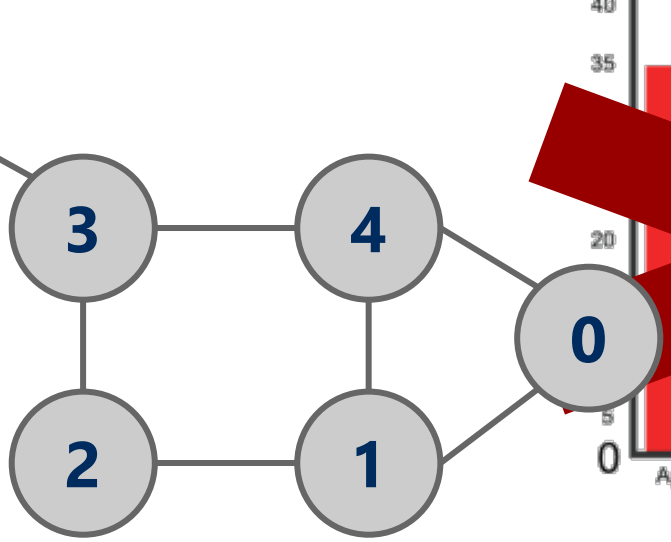
Which Data Structure to use?

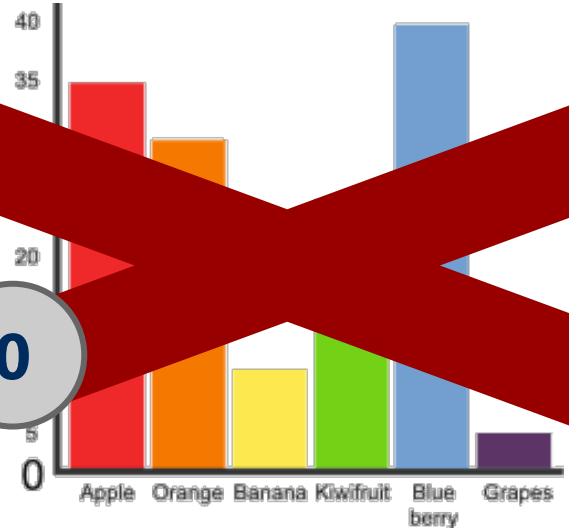
Let's think first about how to structure the data that we have in memory.

- Account1: Connection1, Connection2, …
- Account2: Connection1, Connection2, …
- …

Graphs!

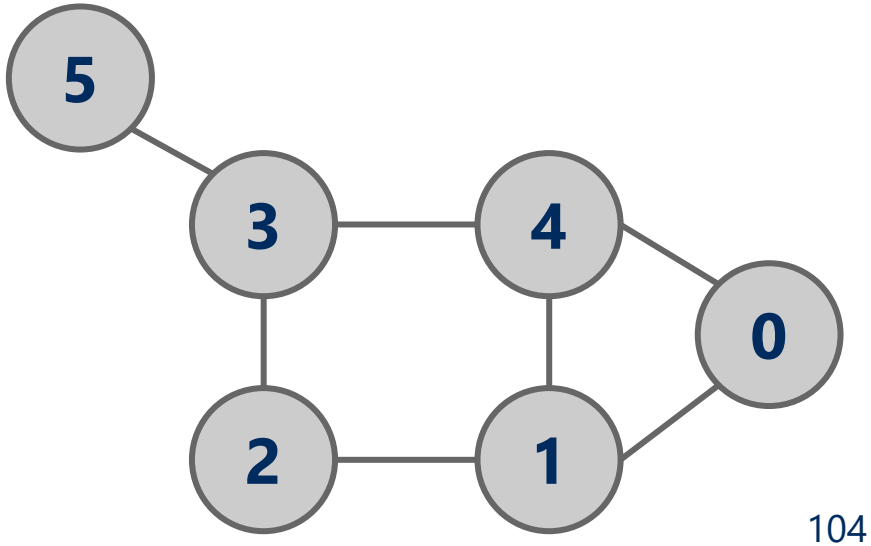

1.5
1
2


5
3
4
0
2
1


40
35
20
5
0
Apple
Orange
Banana
Kiwifruit
Blue berry
Grapes

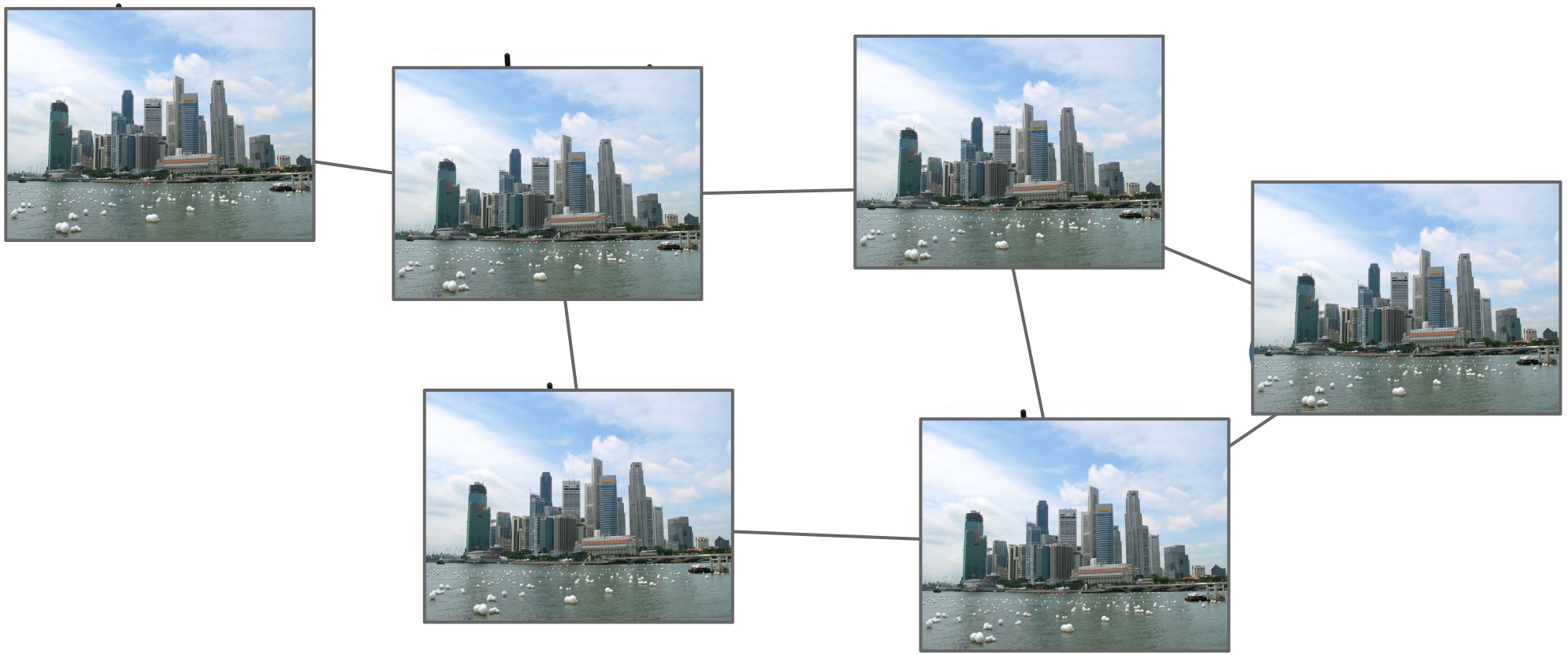
Graphs

- A graph G = (V, E)
  - Where V is a set of vertices
  - E is a set of edges connecting vertex pairs
- Example:
  - V = {0, 1, 2, 3, 4, 5}
  - E = {(0, 1), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4), (3, 5)}



Why?

- Can be used to model many different scenarios

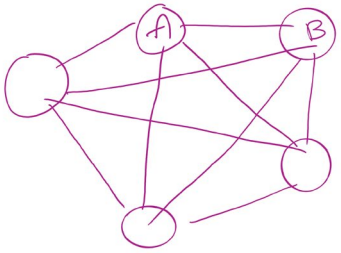


Some definitions

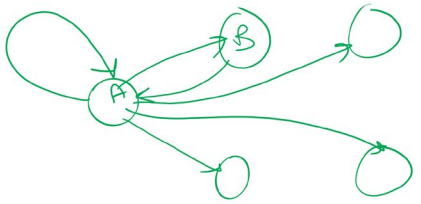
- Undirected graph
  - Edges are unordered pairs: (A, B) == (B, A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B, A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

Graph sizes

- Let v = |V|, and e = |E|
- Given v, what are the minimum/maximum sizes of e?
  - Minimum value of e?
    - Definition doesn't necessitate that there are any edges…
    - So, 0
  - Maximum of e?
    - Depends…
      - Are self edges allowed?
      - Directed graph or undirected graph?
    - In this class, we'll assume directed graphs have self edges while undirected graphs do not



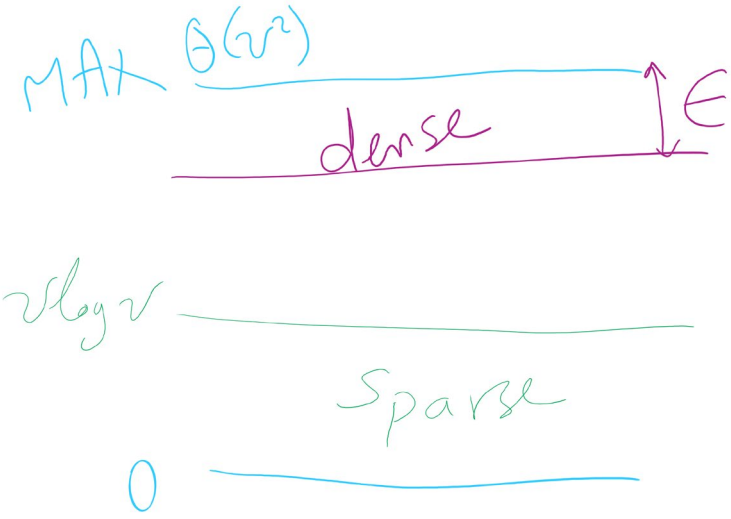
$$\frac{V(V-1)}{2}$$



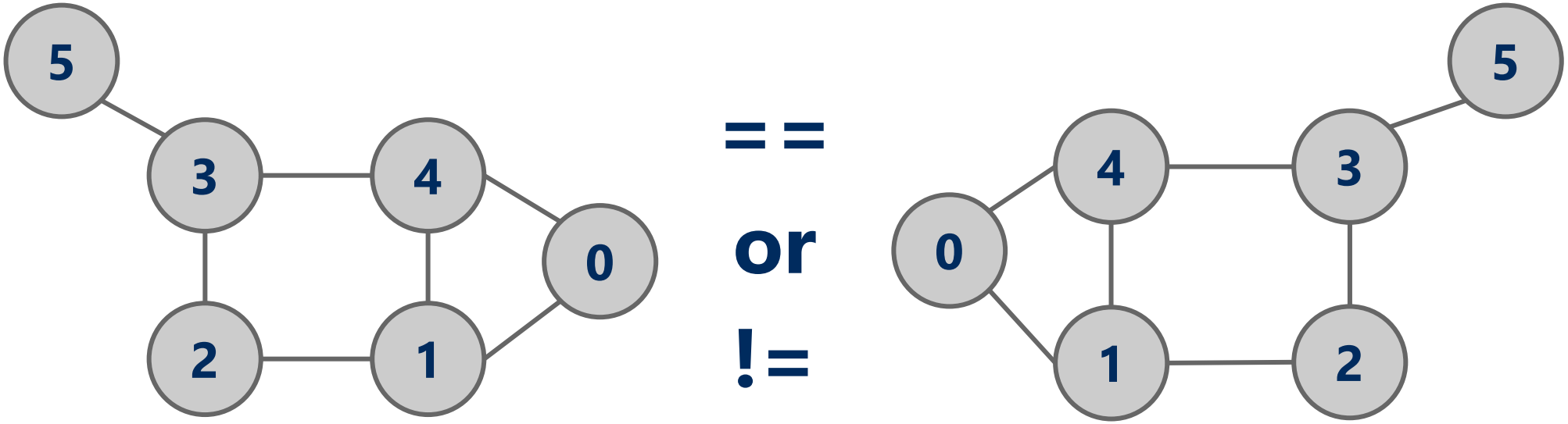
$$V * V = V^2$$

More definitions

- A graph is considered *sparse* if:
  - e <= v lg v
- A graph is considered *dense* as it approaches the maximum number of edges
  - I.e., e == MAX - ε
- A *complete* graph has the maximum number of edges



Question:



- ?

Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Assume we're going to be analyzing static graphs
    - I.e., no insert and remove
  - So what operations should we consider?

Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if (i, j) ∈ E
  - M[i][j] = 0 if (i, j) ∉ E

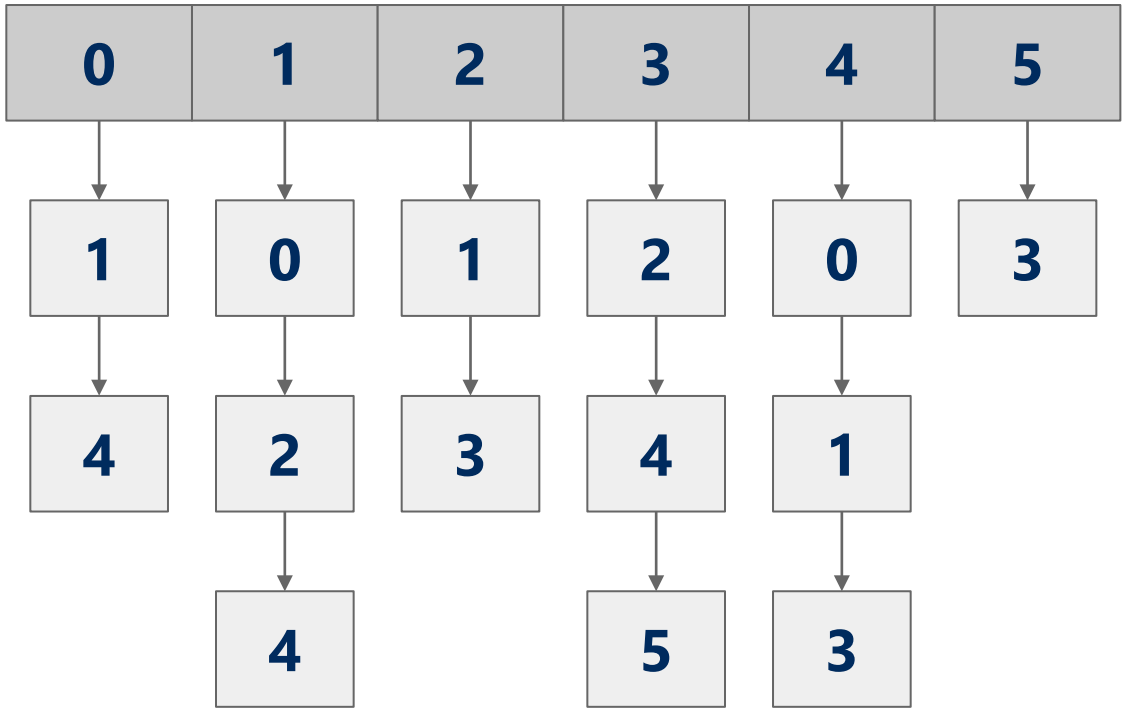
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0

Adjacency matrix analysis

- Runtime?
- Space?

Adjacency lists

- Array of neighbor lists
  - A[i] contains a list of the neighbors of vertex i



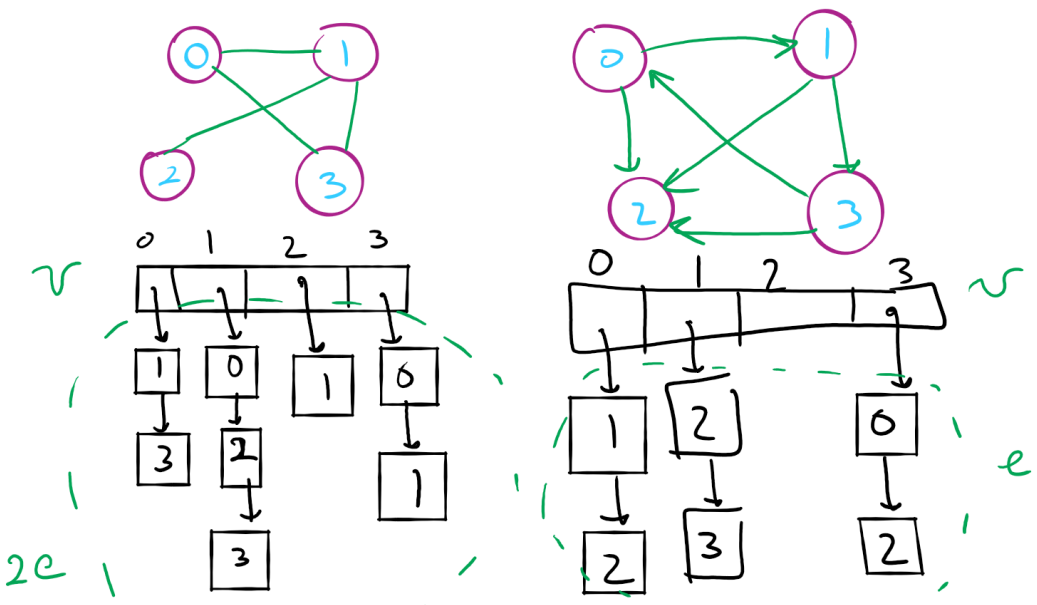
Adjacency list analysis

- Runtime?
- Space?

Comparison

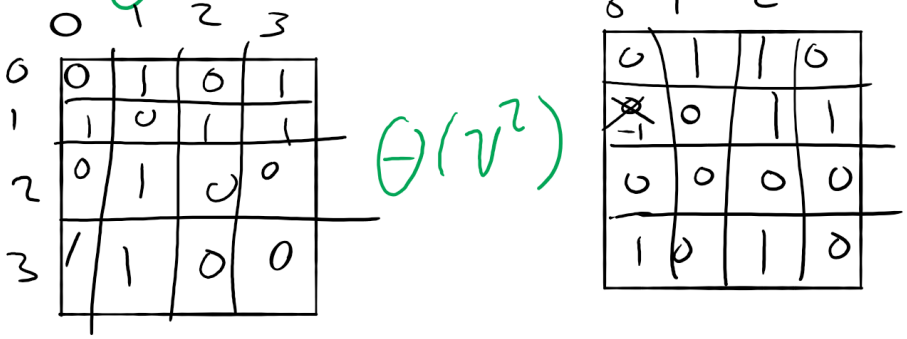
- Where would we want to use adjacency lists vs adjacency matrices?
  - What about the list of vertices/list of edges approach?

Graph Representation Example 2



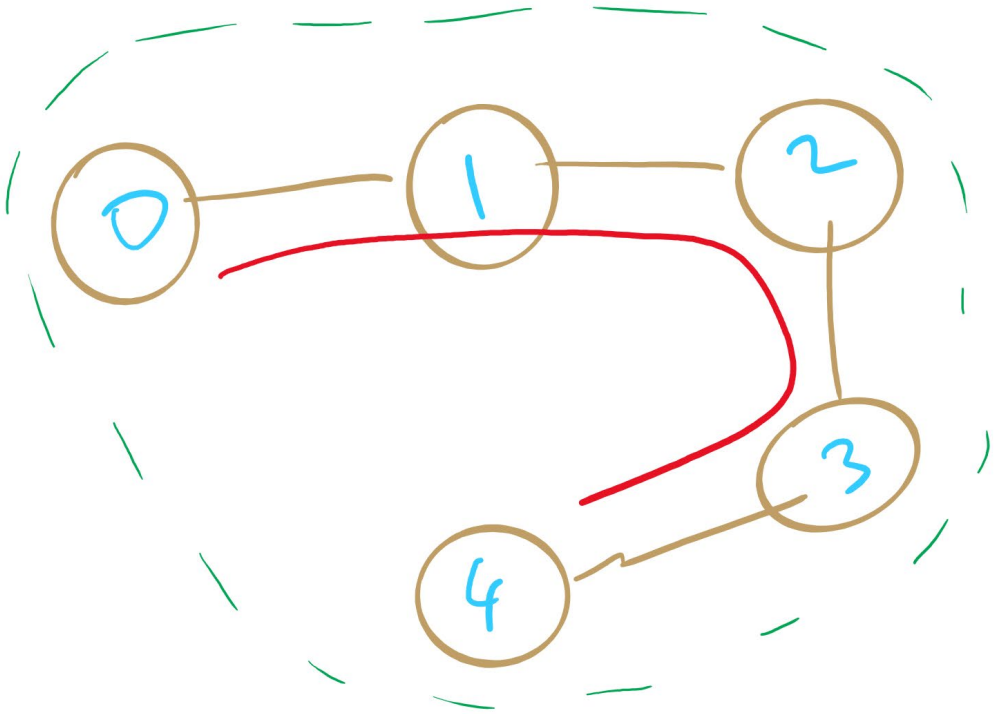
$\Theta(V+2e) = \Theta(V+e)$ $\Theta(V+e)$

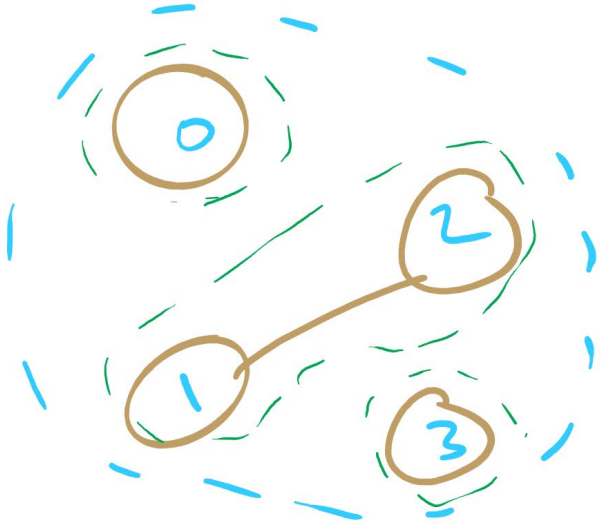
ArrayList<LinkedList<Node>> adjLists;



$\Theta(V^2)$

Sparse Graphs

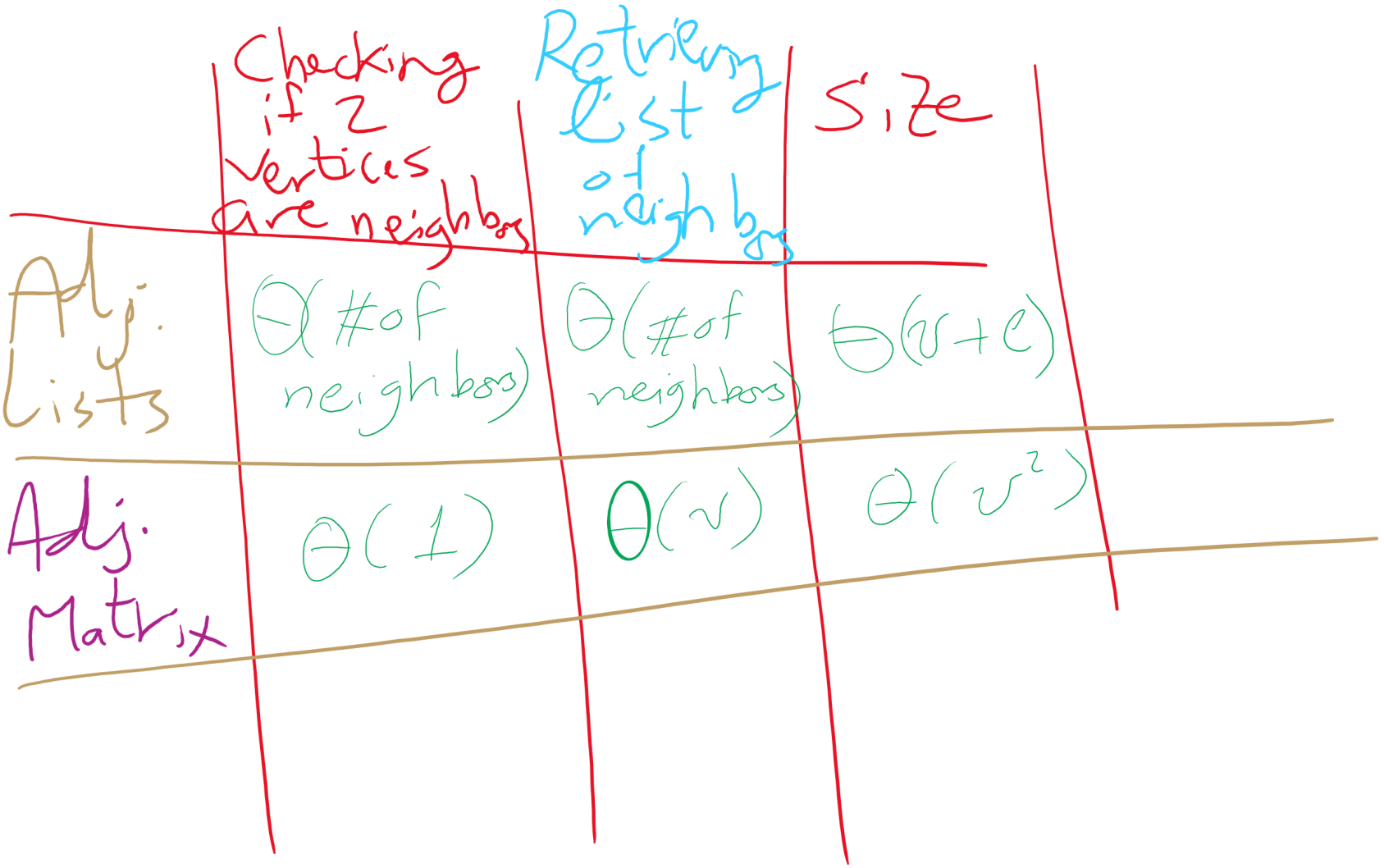




Comparison

- Where would we want to use adjacency lists vs adjacency matrices?
  - What about the list of vertices/list of edges approach?

Adjacency Matrix vs. Adjacency Lists



Even more definitions

- Path
  - A sequence of adjacent vertices
- Simple Path
  - A path in which no vertices are repeated
- Simple Cycle
  - A simple path with the same first and last vertex
- Connected Graph
  - A graph in which a path exists between all vertex pairs
- Connected Component
  - Connected subgraph of a graph
- Acyclic Graph
  - A graph with no cycles
- Tree
  - ?
  - A connected, acyclic graph
    - Has exactly v-1 edges

Graph traversal

- What is the best order to traverse a graph?
- Two primary approaches:
  - Breadth-first search (BFS)
    - Search all directions evenly
      - I.e., from i, visit all of i's neighbors, then all of their neighbors, etc.
    - Would help us compute the distance between two vertices
      - Remember our problem of the day?
  - Depth-first search (DFS)
    - "Dive" as deep as possible into the graph first
    - Branch when necessary
    - Would help us find articulation points
      - Remember our problem of the day?

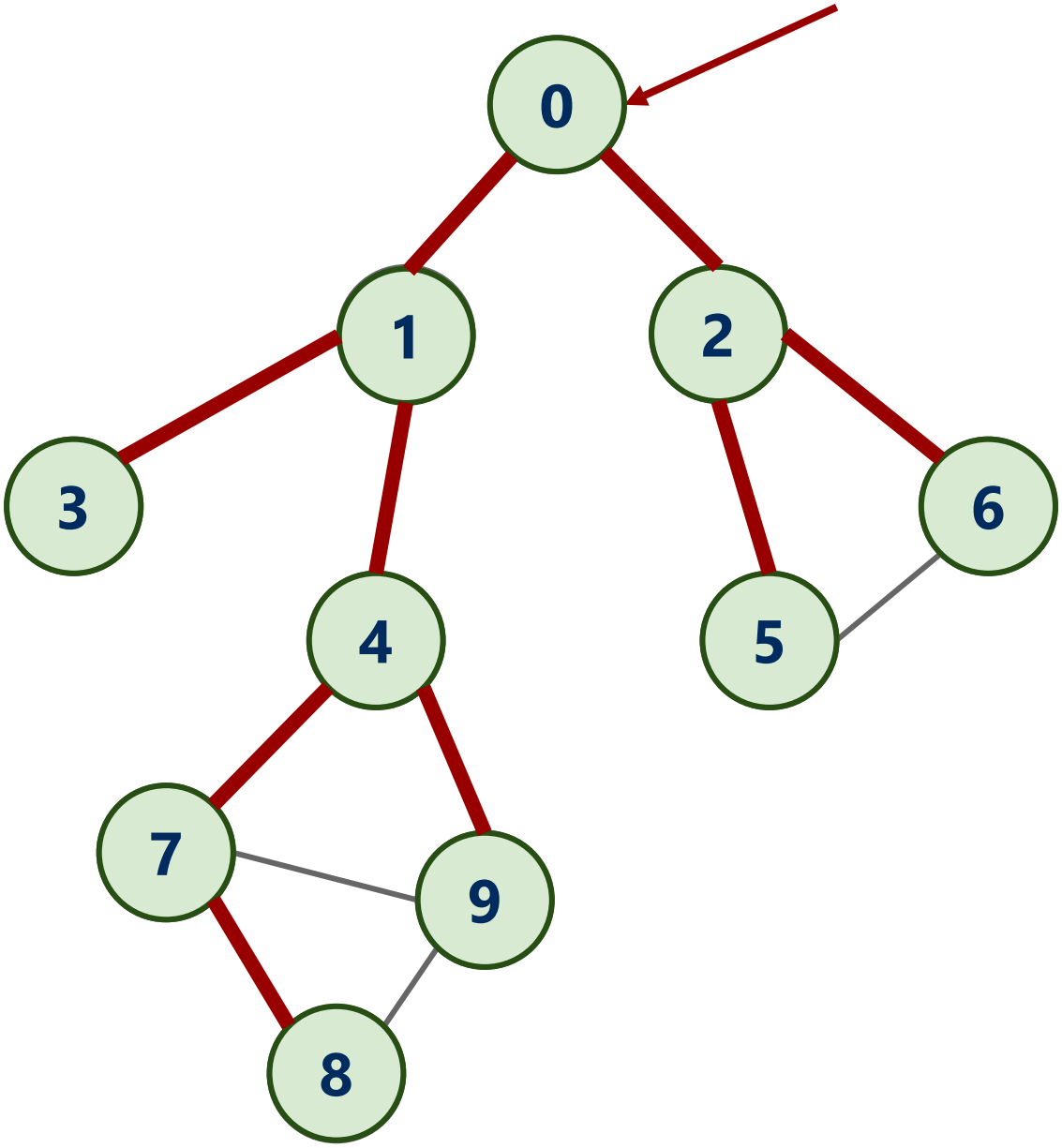
BFS

- Can be easily implemented using a queue
  - For each vertex visited, add all of its neighbors to the Q (if not previously added)
    - Vertices that have been seen (i.e., added to the Q) but not yet visited are said to be the *fringe*
  - Pop head of the queue to be the next visited vertex
- See example

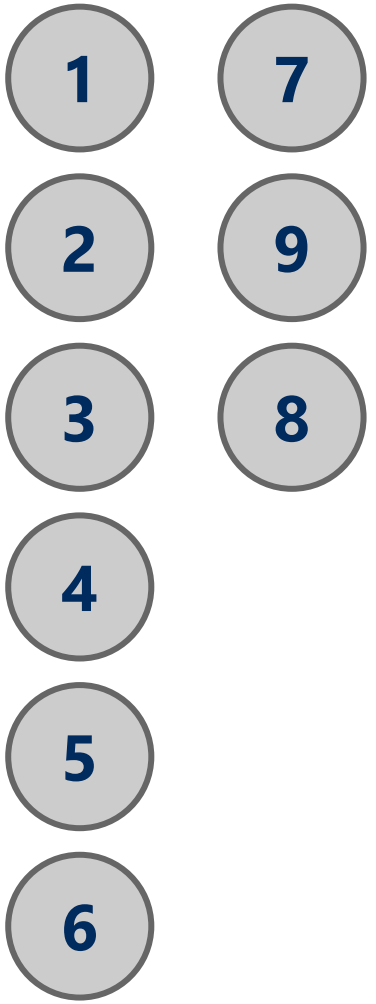
BFS Pseudo-code

```
BFS(vertex v) {
    insert v into Q
    while(Q is not empty) {
        w = pop head of Q
        visit w
        for each unseen neighbor x of w
            parent[x]=w; distance[x]= distance[w]+1
            add x to Q
    }
}
```

BFS example



Q



Shortest paths

- BFS traversals can further be used to determine the *shortest path* between two vertices

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

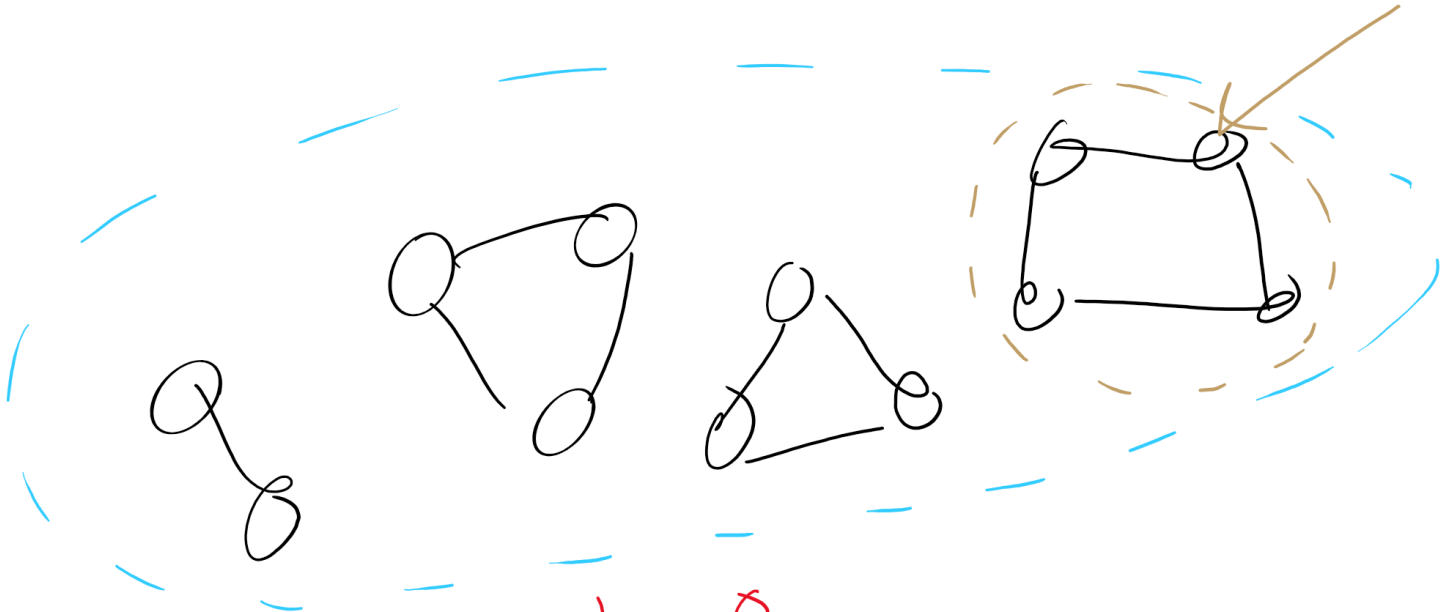
Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

BFS would be called from a wrapper function

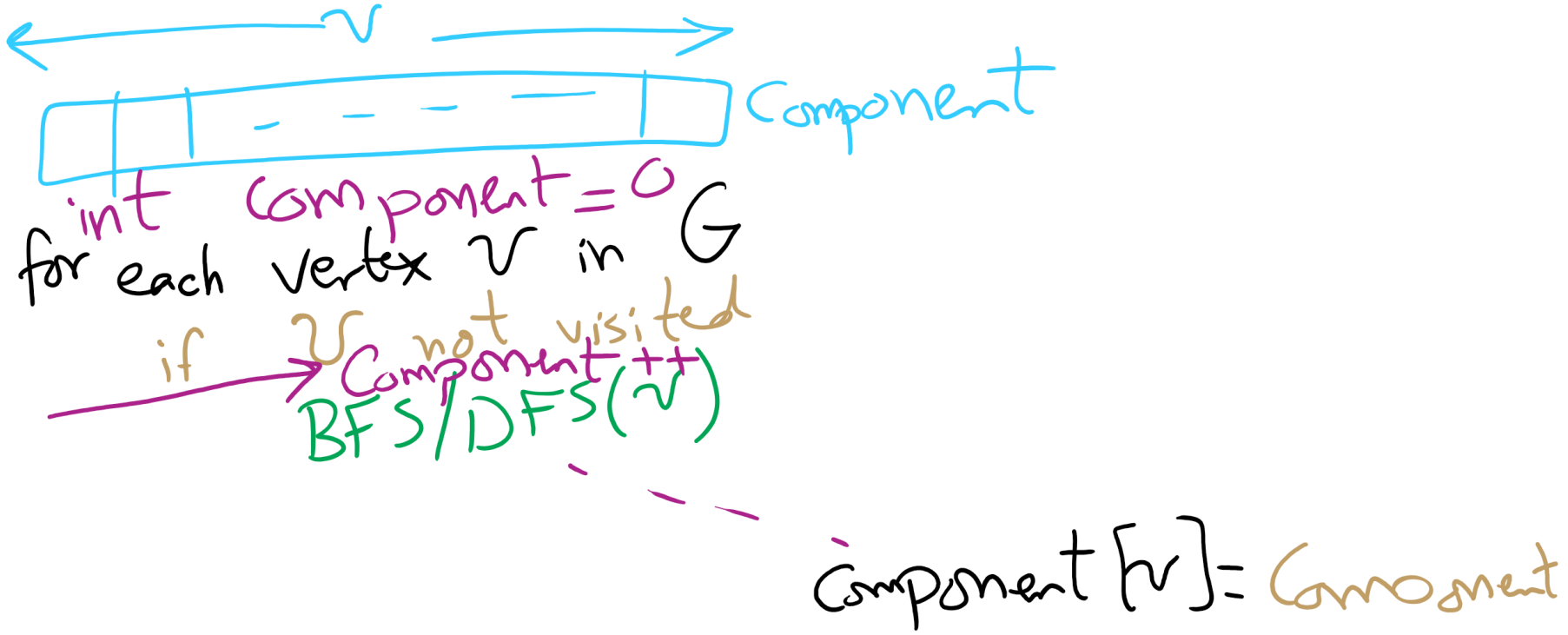
- If the graph is connected:
  - bfs() is called only once and returns a *spanning tree*
- Else:
  - A loop in the wrapper function will have to continually call bfs() while **there are still unseen vertices**
  - Each call will yield a spanning tree for a connected component of the graph

Wrapper function and connected components



```
int component = 0
for each vertex v
    if v is unseen
        DFS(v) / BFS(v)
        component++
```

Wrapper function for BFS



Problem of the Day

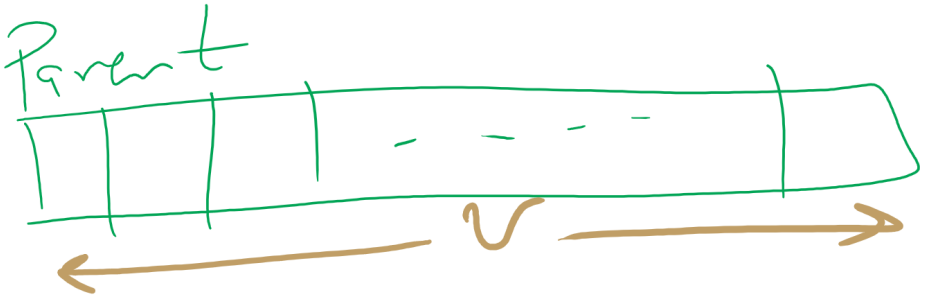
- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

DFS

- Already seen and used this throughout the term
  - For tries…
  - For Huffman encoding…
- Can be easily implemented recursively
  - For each vertex, visit first (in some arbitrary order) unseen neighbor
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try next unseen neighbor after backtracking

DFS Pseudo-code

```
DFS(Vertex v) {
  // visit v
  Mark v as seen
  for each unseen neighbor w
    Parent[w] = v
    DFS(w)
  // visit v
}
```



DFS example 2