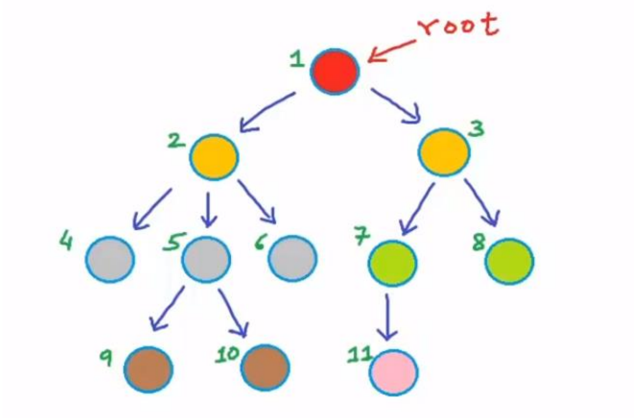
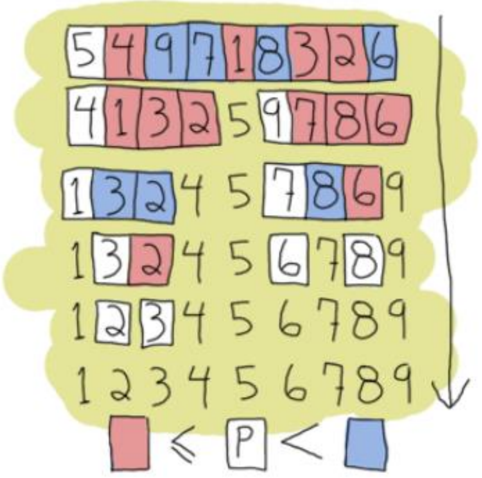
ECE 250 Data Structures & Algorithms



Trees



Ziqiang Patrick Huang

Electrical and Computer Engineering

University of Waterloo

WATERLOO | ENGINEERING

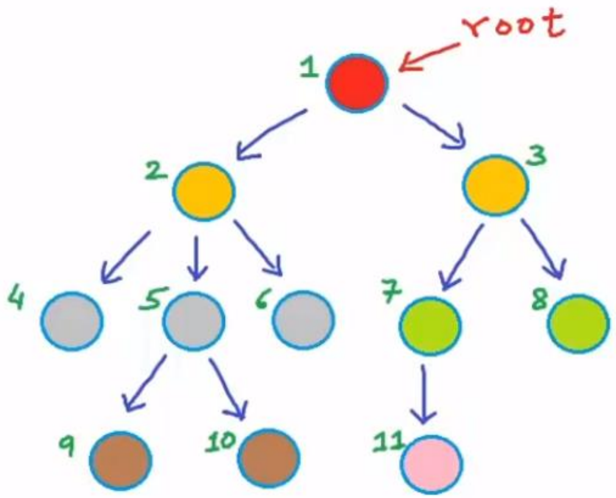
Data Structure Review

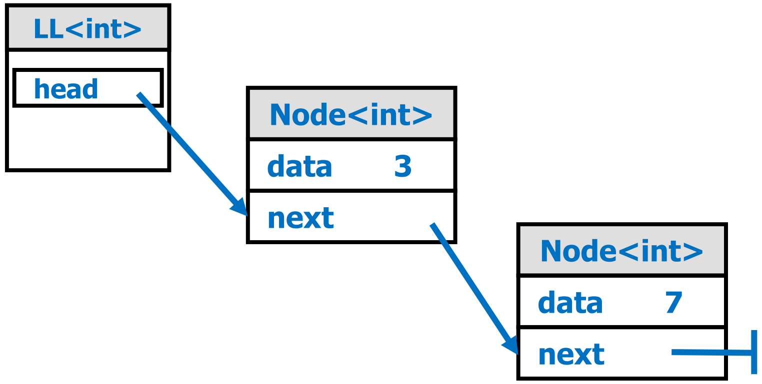
	Add to Front	Add to Back	Remove from Front	Remove from Back	Access	Search
Array	O(n)	O(1)*	O(n)	O(1)	O(1)	O(n)
Singly-Linked List w/Tail	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)
Doubly-Linked List w/Tail	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)

ADT Review

- Stacks, Queues, and Deques are limited by their ADT operations
- To access or search, we'd have to remove each of the data and re-add them afterwards
- These ADTs/data structures are meant for lightweight adding and removing, not searching for data

What about Trees?





A singly-linked is a tree!

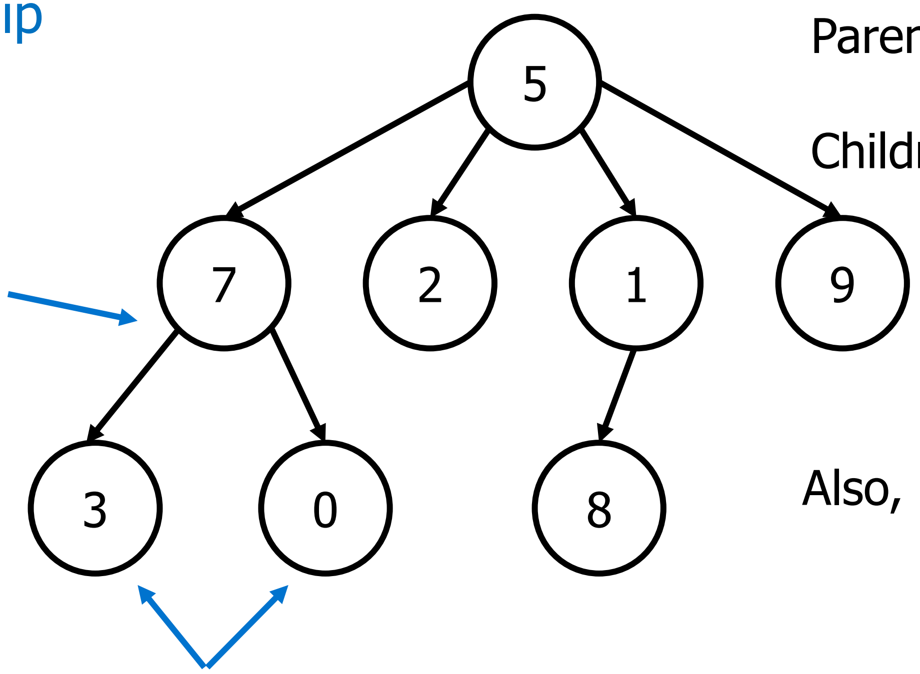
Tree Terminology: Parent-child Relationship

Hierarchical relationship

Parent (of a node): the node that points at it

Children (of a node): Nodes it directly points to

7 is the parent of 0 and 3



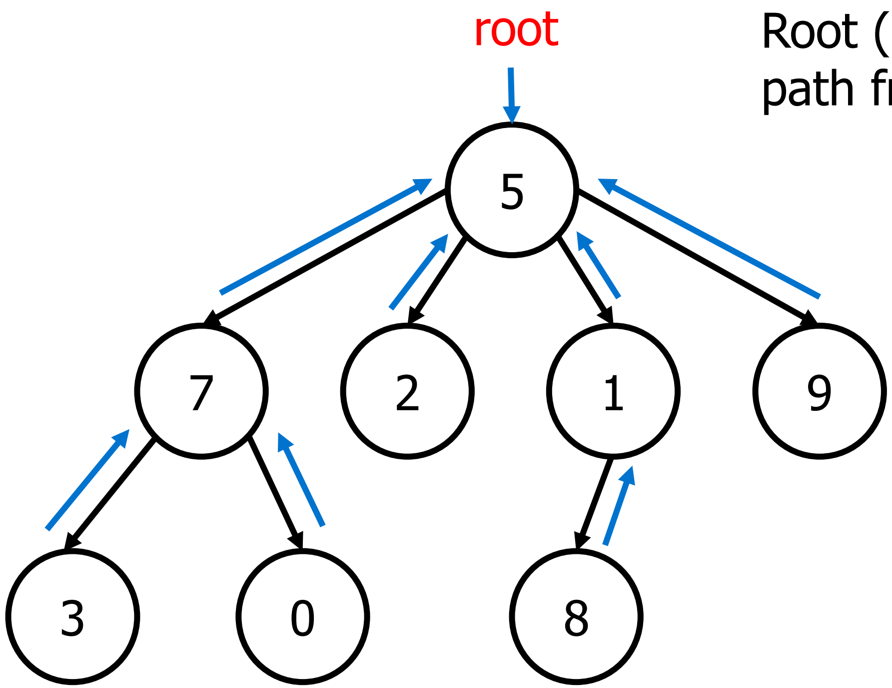
0 and 3 are children of 7

Also, 5 is the grandparent of 3, 0, and 8;

7, 2, 1 and 9 are siblings

3 and 0 are cousins of 8

Tree Terminology: Root

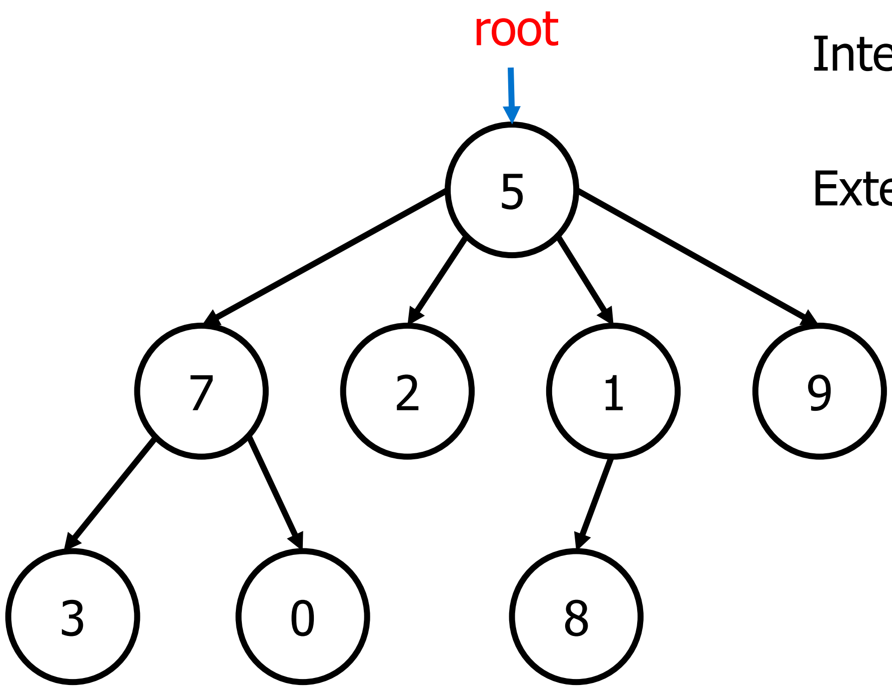


Root (of a tree): the node that exists a directed path from it to every other node in the tree

Tree Terminology: Internal vs External Nodes

Internal Nodes: 5, 7, 1

External Nodes: 3, 0, 2, 8, 9

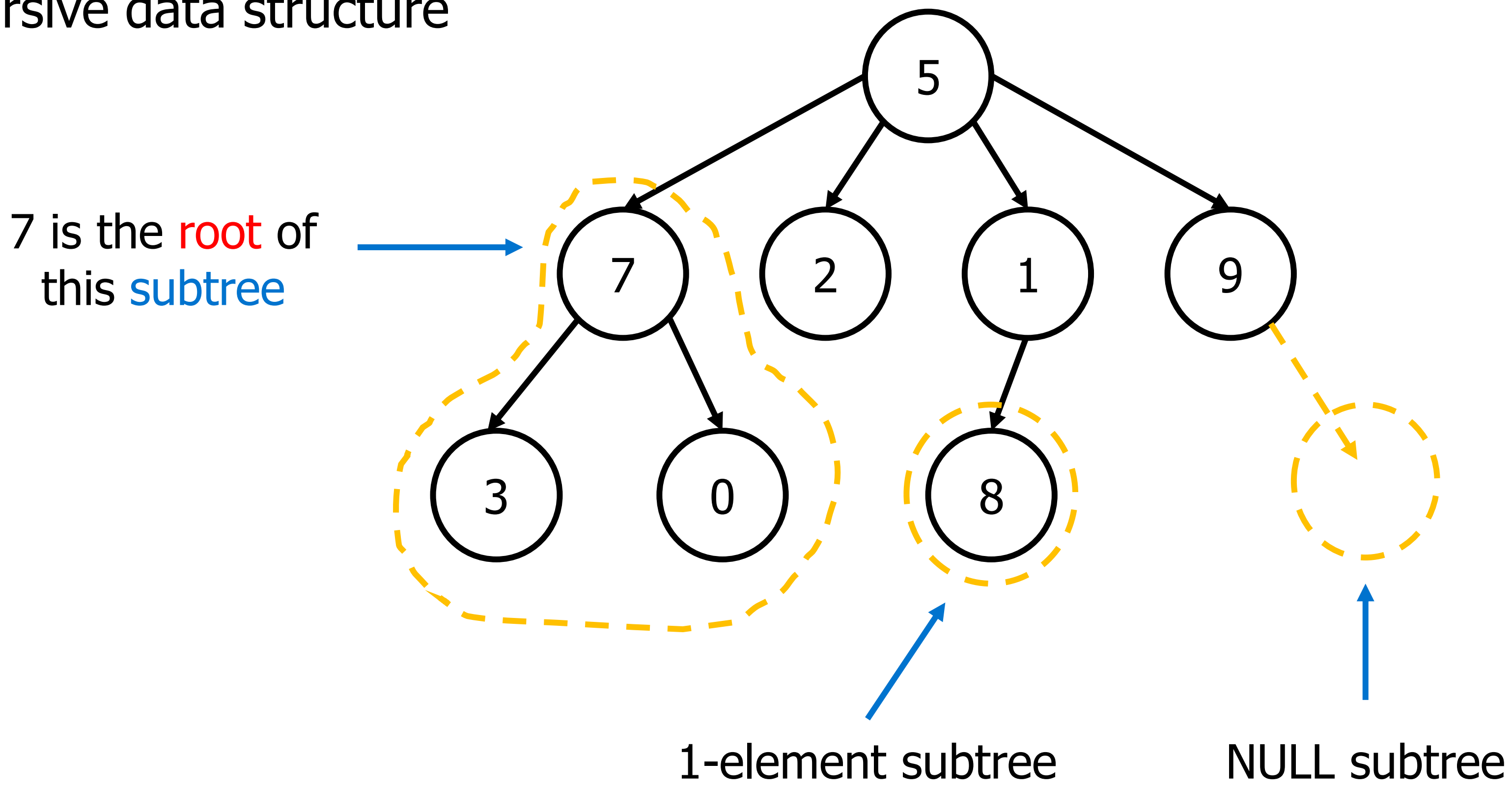


Internal Nodes: nodes that have children

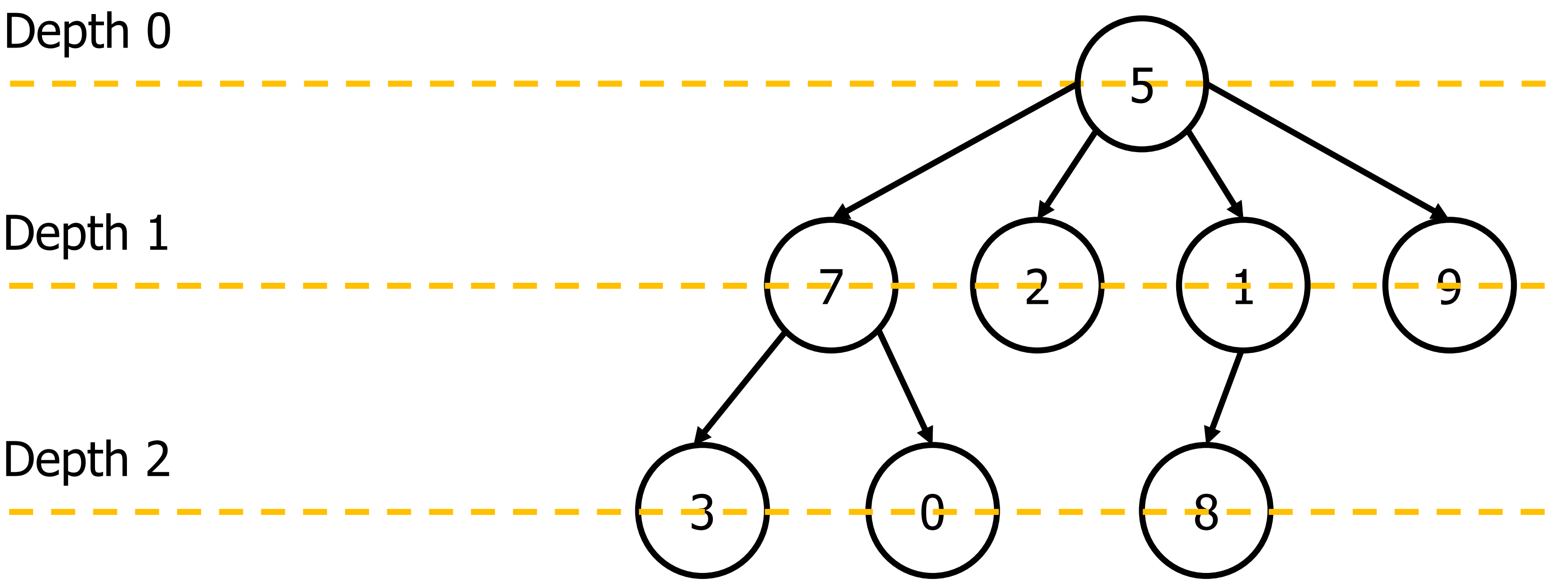
External Nodes (Leaf Nodes): otherwise

Tree Terminology: Subtrees

Recursive data structure



Tree Terminology: Depth



Some consider root has depth 1
(Just different conventions)

In this course, root has depth 0

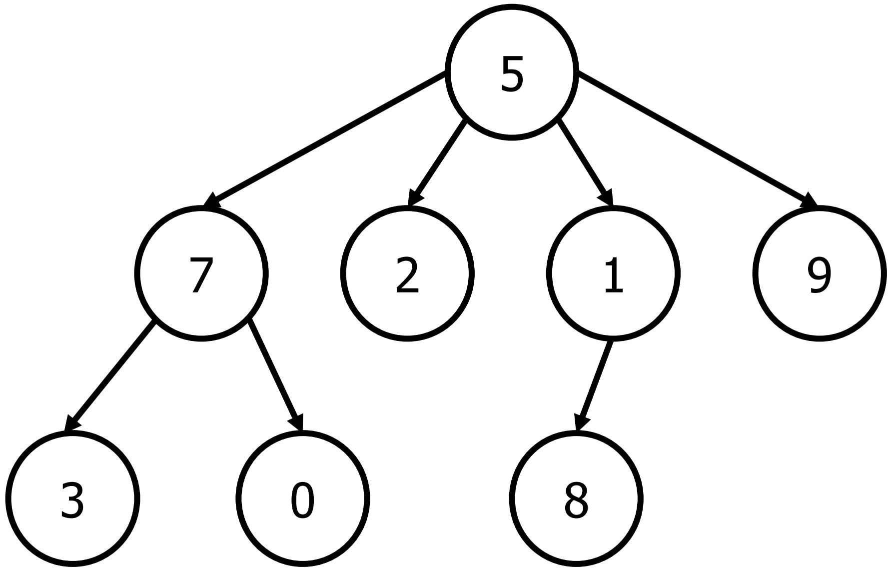
Depth (of a node): the length of the path from the root to that node

Tree Terminology: Height

Height 0: 3, 0, 2, 8, 9

Height 1: 7, 1

Height 2: 5



Height(leaf) = 0

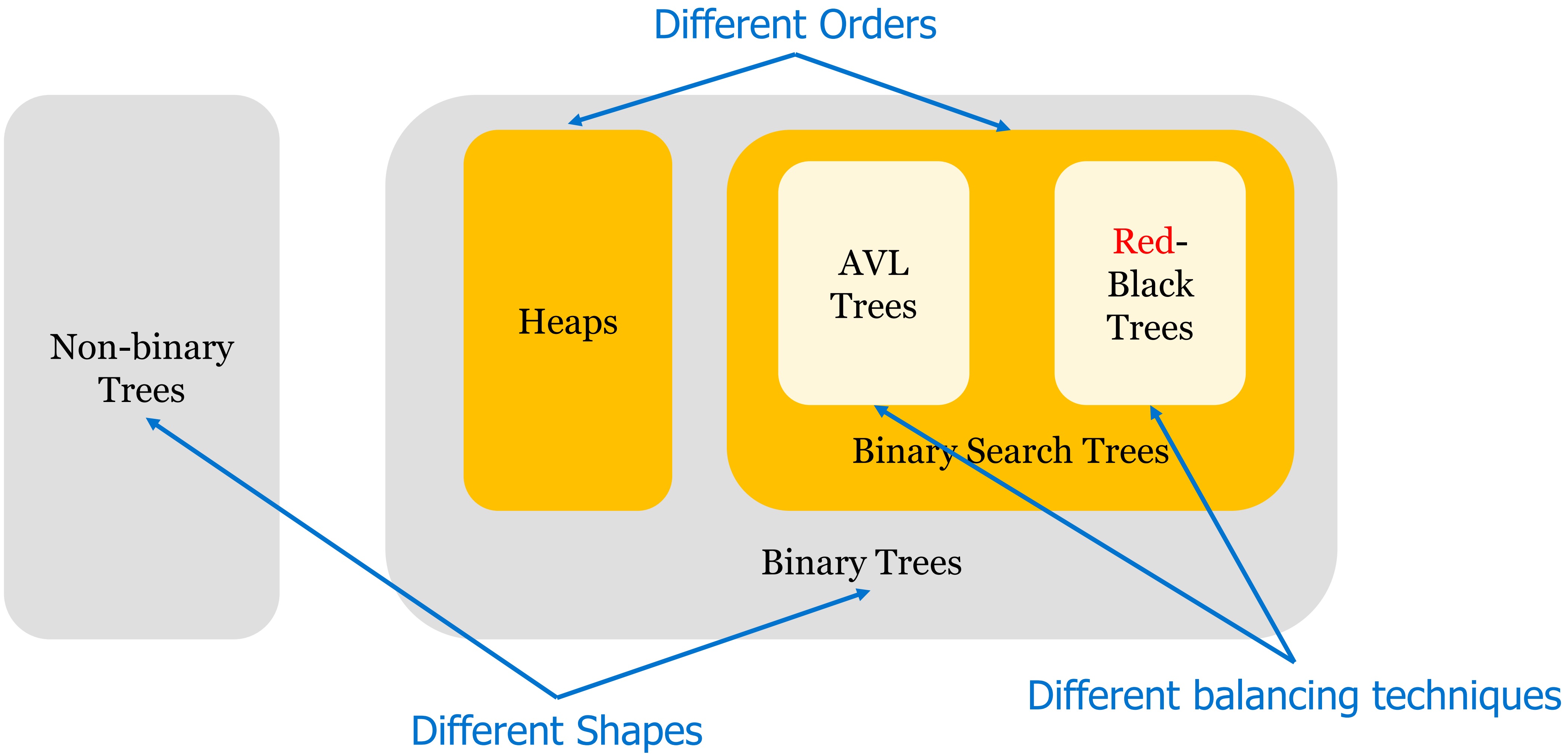
Height(node) = (Max. child height) + 1

Height (of a node): the maximum length path from it to a leaf node

Tree Terminology

- Trees: Connected linked structures with no cycles
  - A cycle is a path that starts and ends at the same node
  - E.g., a circular linked list is not a tree
- Often considered ADTs: can be implemented in multiple ways depending on the details of the type of tree
  - Mostly implemented using linked list-like structures (with nodes & pointers)
- Trees can be further categorized if we give them some other properties:
  - Shape: What is the structure of the nodes in the tree? (e.g., binary trees)
  - Order: How is the data arranged in the tree? (e.g., heap)

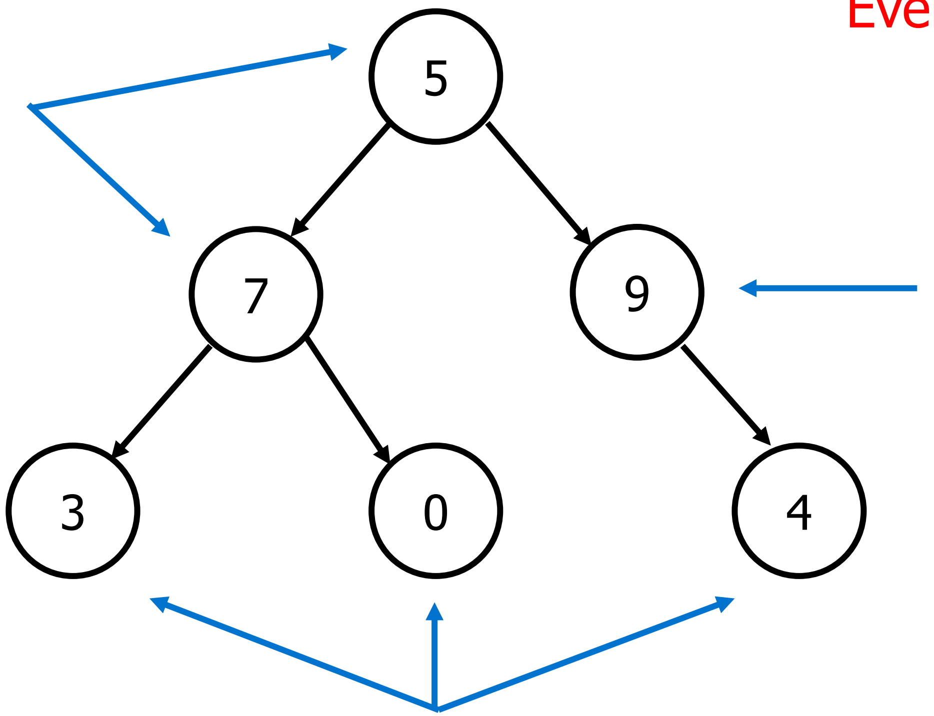
Classification of Trees



Binary Trees

Every node has at most 2 children

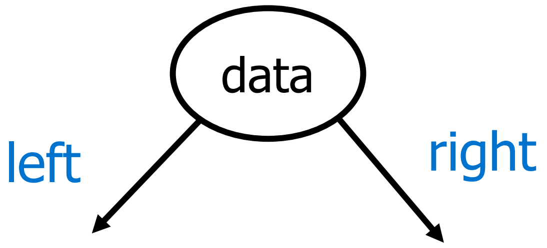
5 and 7 have 2 children
(the max.)



9 has only 1 child

3, 0 and 4 have no children
(the min.)

Binary Tree Node

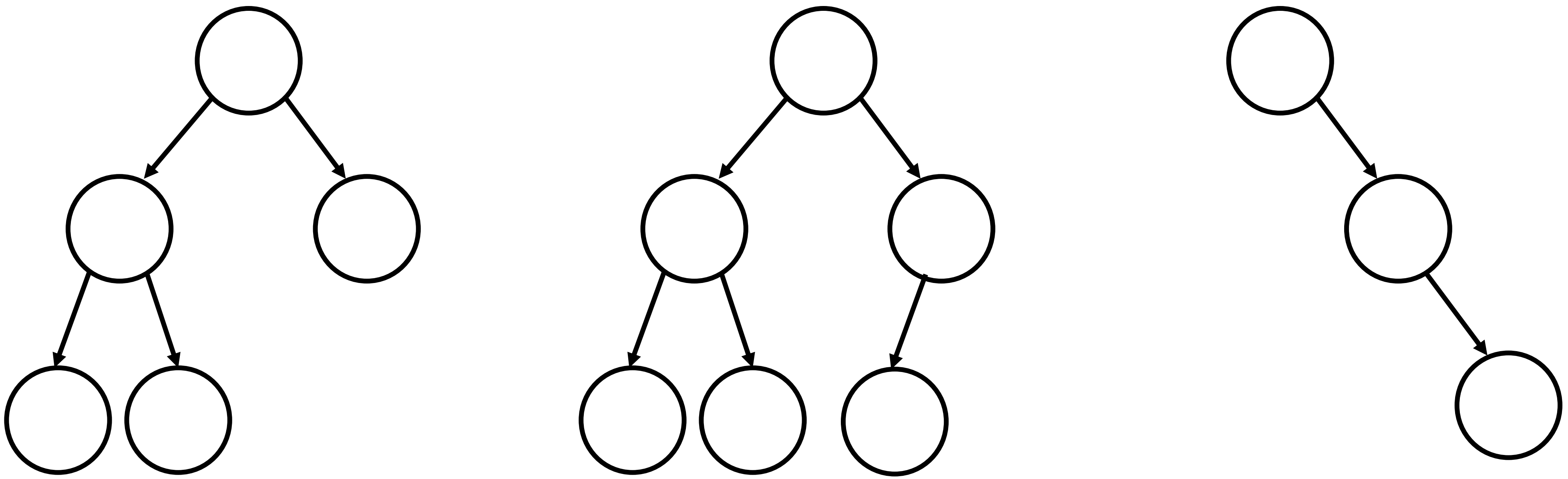


can point to an actual node or null

Other Potential Node Information:

- Parent
- Depth
- Height

Shape Property for Binary Tree



Full Tree
Every node must have 0 or 2 children

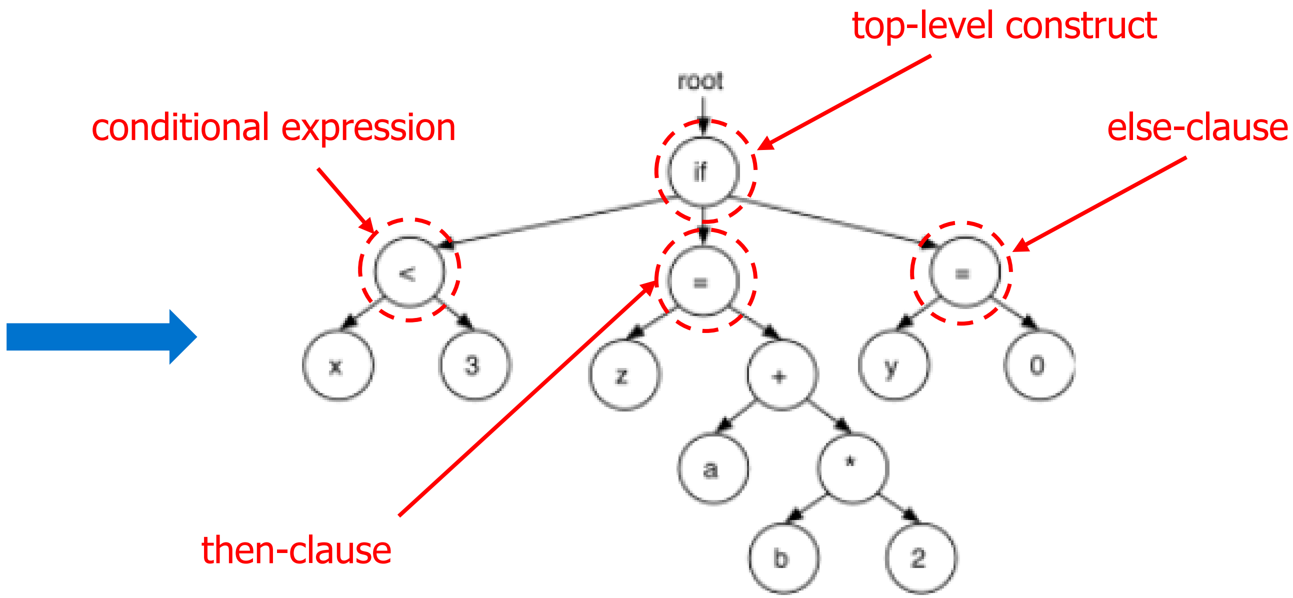
Complete Tree
Every level must be filled except for the last one, which is filled left to right

Degenerated Tree
All nodes have 1 child

Use of Non-binary Trees

- Example: Abstract syntax trees
  - Used in compilers for analyzing a program's grammatical structure
  - Abstract away certain details of the concrete syntax of the code
  - Focus on representing different language constructs (e.g., statements, expressions, declarations, operators, etc.)

```
if (x < 3) {
  z = a + b * 2;
}
else {
  y = 0;
}
```

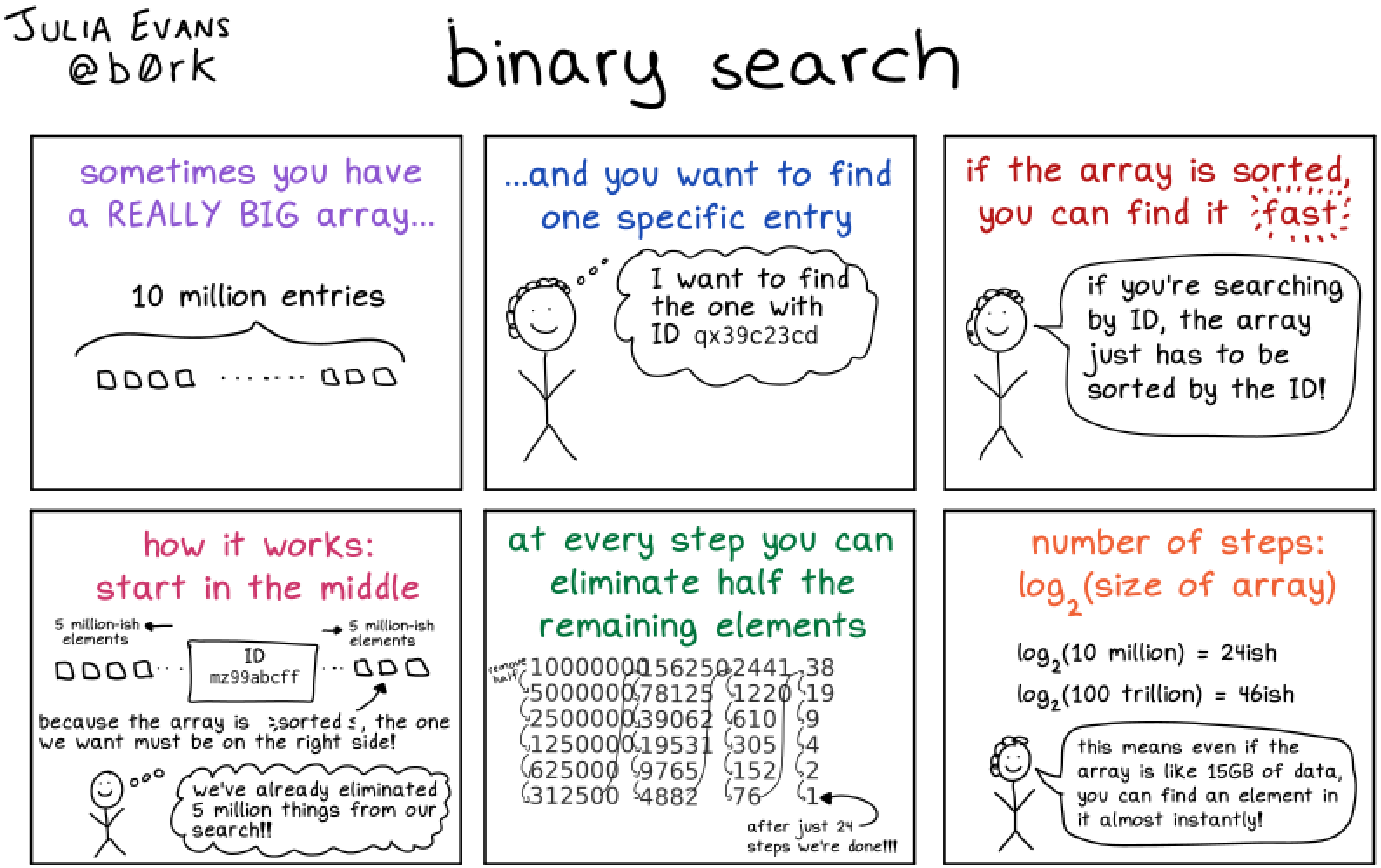


Binary Search

- Recall: Linked lists and arrays offer O(n) search
  - Good, but we can do better!

- Binary search → search in sub-linear time
  - Start with ordered data
  - Split the problem in half at each step → find in log(n) steps
  - O(log(n)) is much better than O(n)
    - E.g., log(1billion) ≈ 30

Binary Search on Arrays



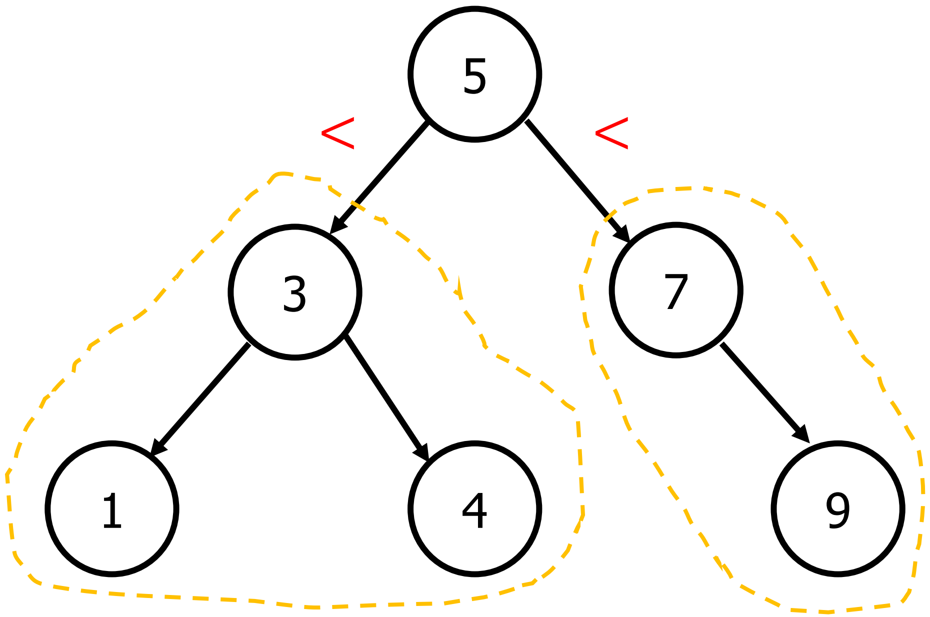
Array Binary Search

- Suppose we implement a Set of ints with a sorted array
  - Want to check contains in O(log(n)) time …

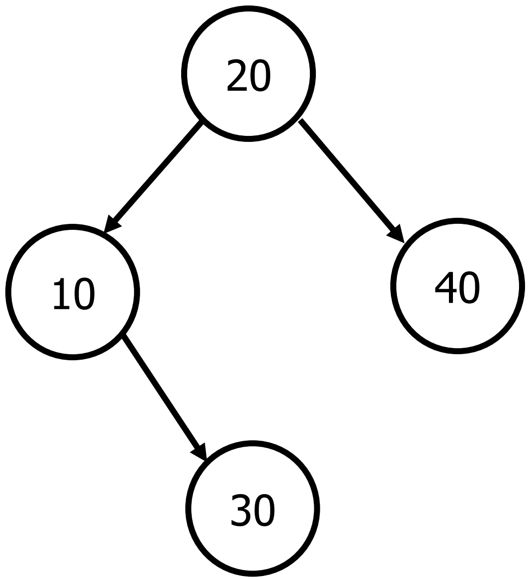
```
class IntSet {
  int * array;
  int arraySize;
  …
  bool contains (int x) {
    //exercise for you
  }
};
```

Binary Search Tree

- Binary Search Tree(BST): A binary tree in which everything to the left of any given node must be smaller than that node, and everything to the right must be greater than that node.



Valid BST



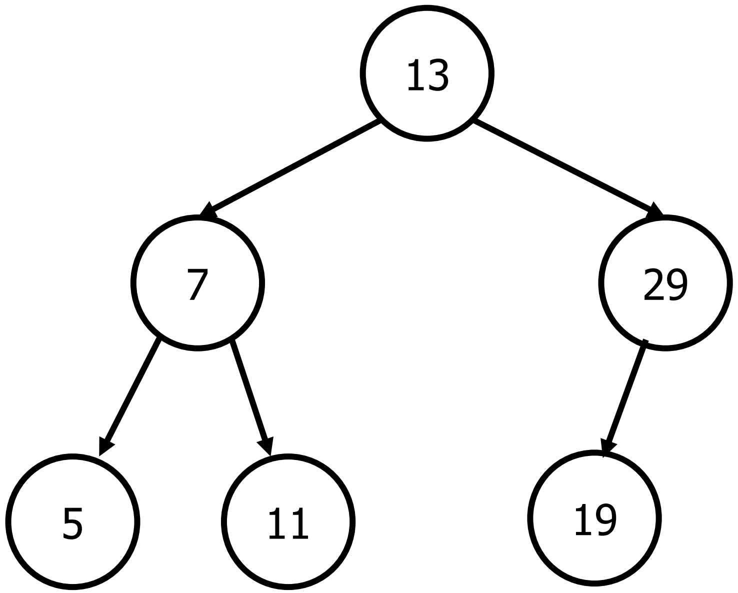
Invalid BST: 30 > 20

Tree Traversals

- Depth Traversals
  - Pre-order Traversal
  - In-order Traversal
  - Post-order Traversal

- Breadth Traversal
  - Level-order traversal

Start with In-order



- In-order: 5, 7, 11, 13, 19, 29
  - How do we come up with this?
  - Everyone take a moment to think out an algorithm …
    - Might help to imagine the tree without numbers

In-order Traversal Algorithm

- Check if trying to traverse empty tree?
  - If so, doing nothing
  - If not, then …
    - traverse left subtree (recurse left)
    - Print out my value
    - traverse right subtree (recurse right)

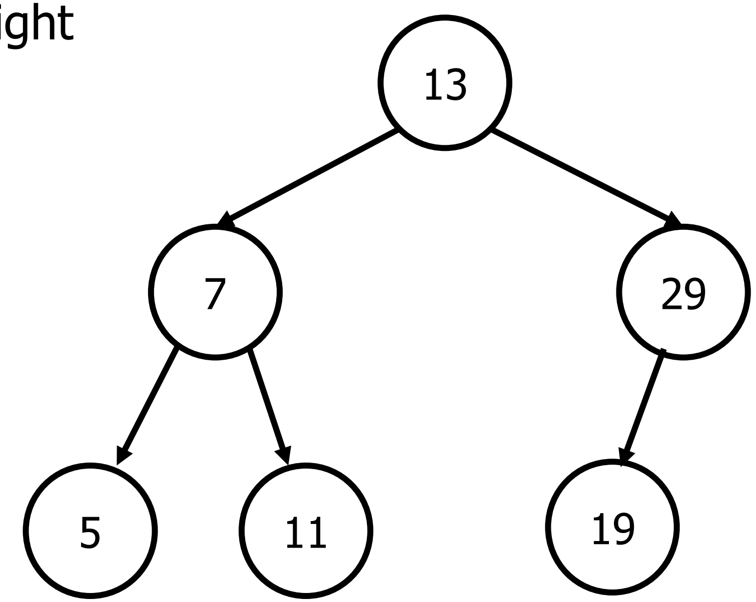
In-order Traversal in C++

```
void inorder (Node * curr) {
    if (curr == NULL) {
        //nothing
    }
    else {
        inorder(curr->left);
        cout << curr->data << endl;
        inorder(curr->right);
    }
}
```

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls



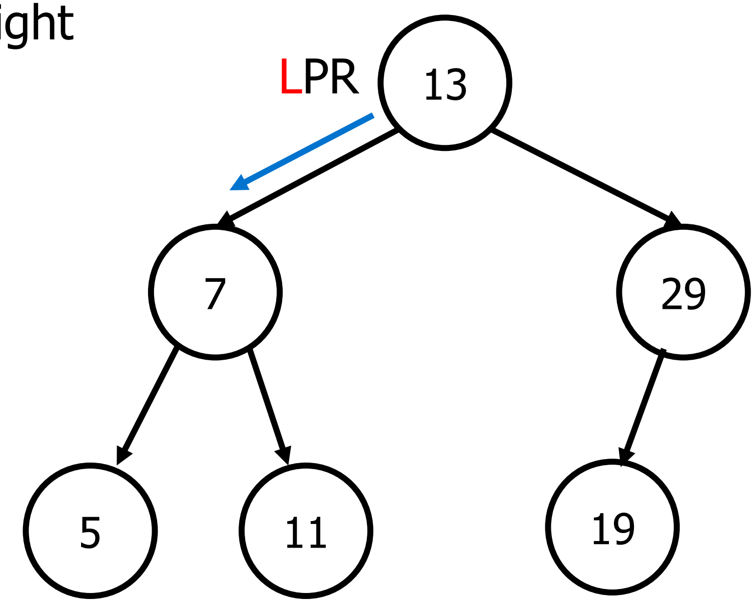
Traverse:

WATERLOO | ENGINEERING

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls



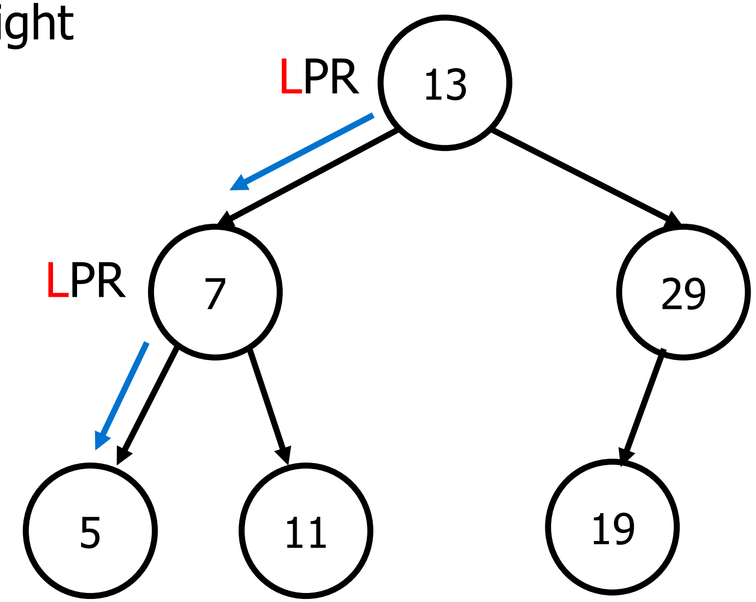
Traverse:

WATERLOO | ENGINEERING

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

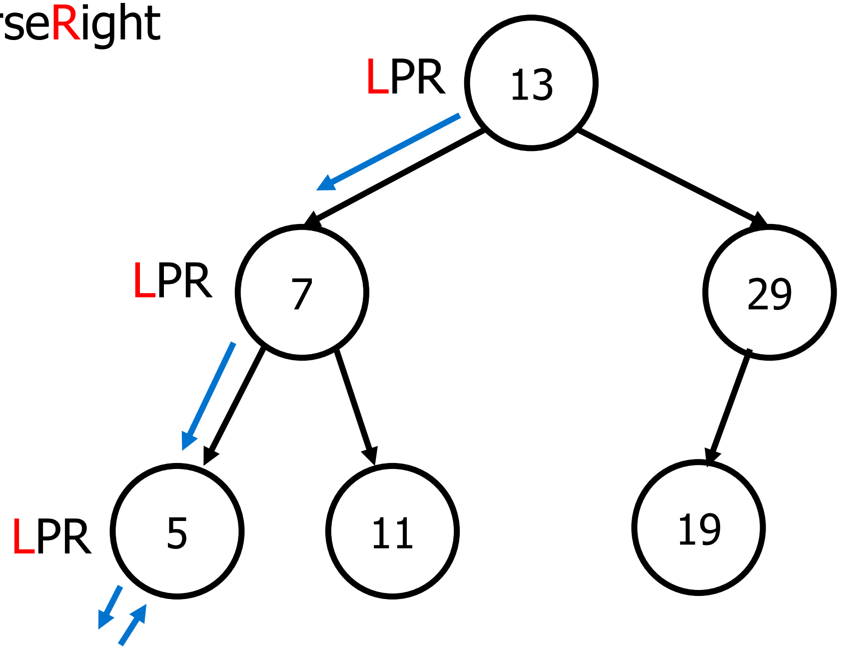


Traverse:

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls



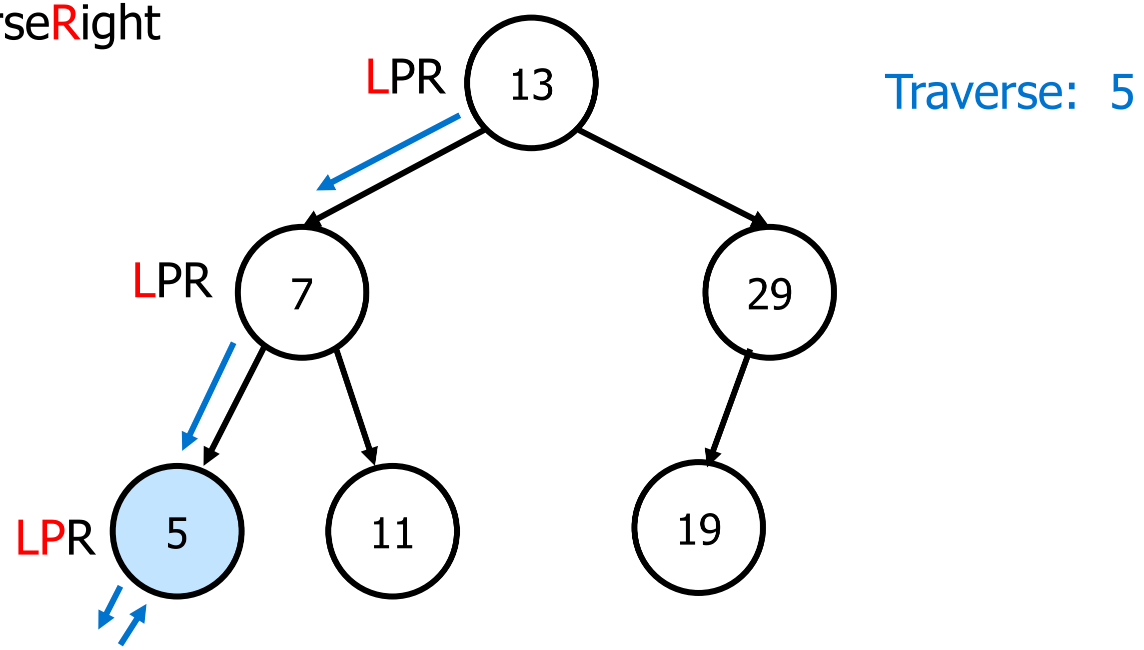
Traverse:

WATERLOO | ENGINEERING

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

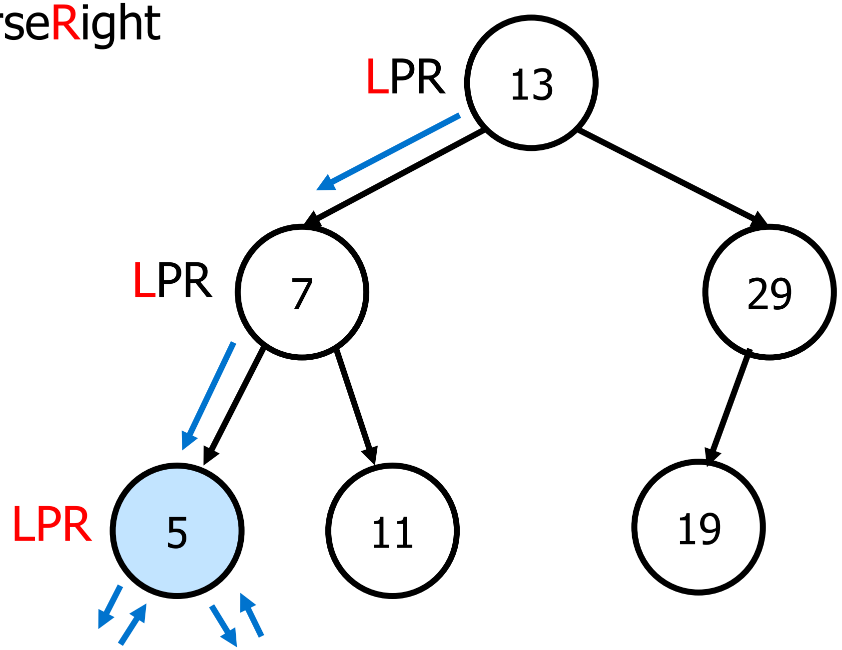
Arrow: direction of calls



In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

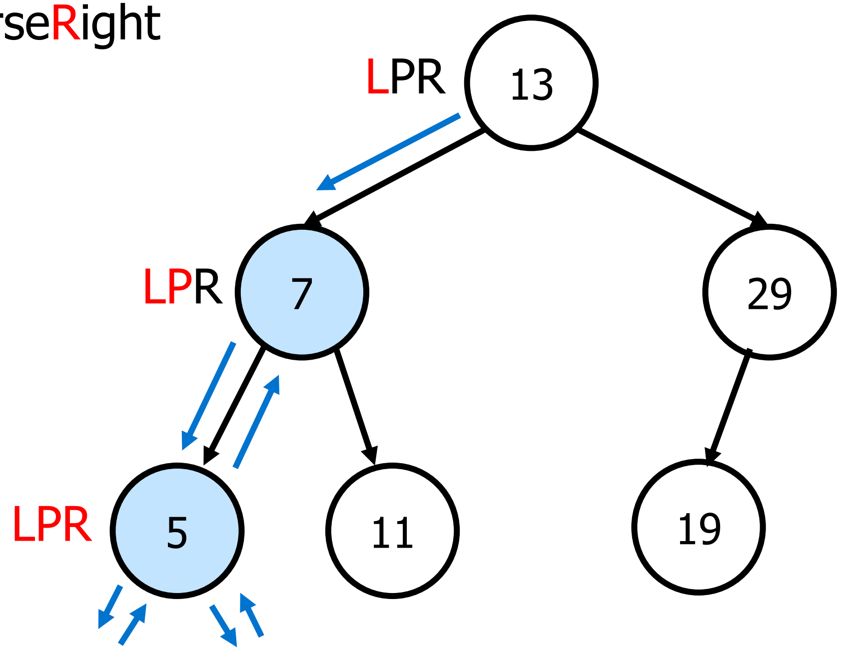


Traverse: 5

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

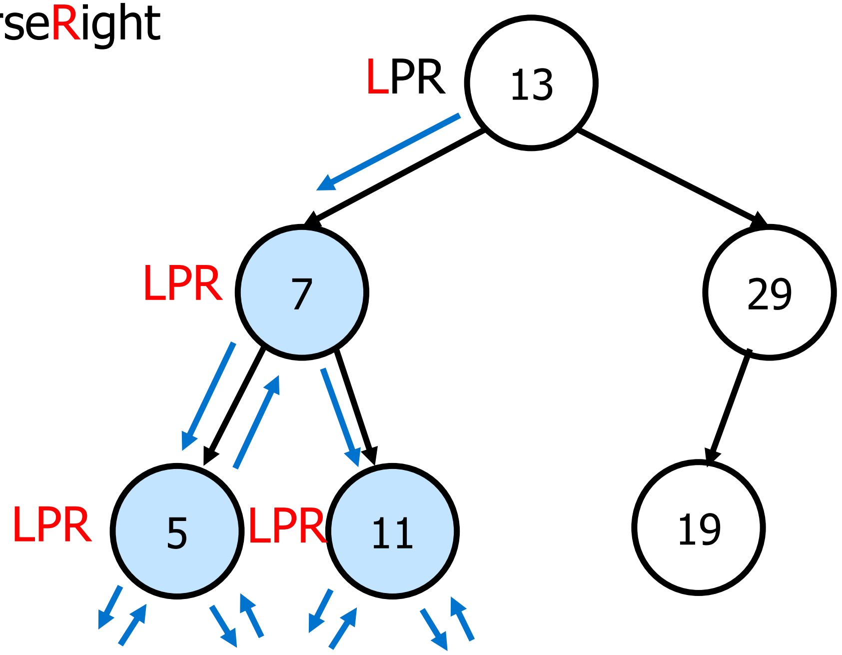


Traverse: 5, 7

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

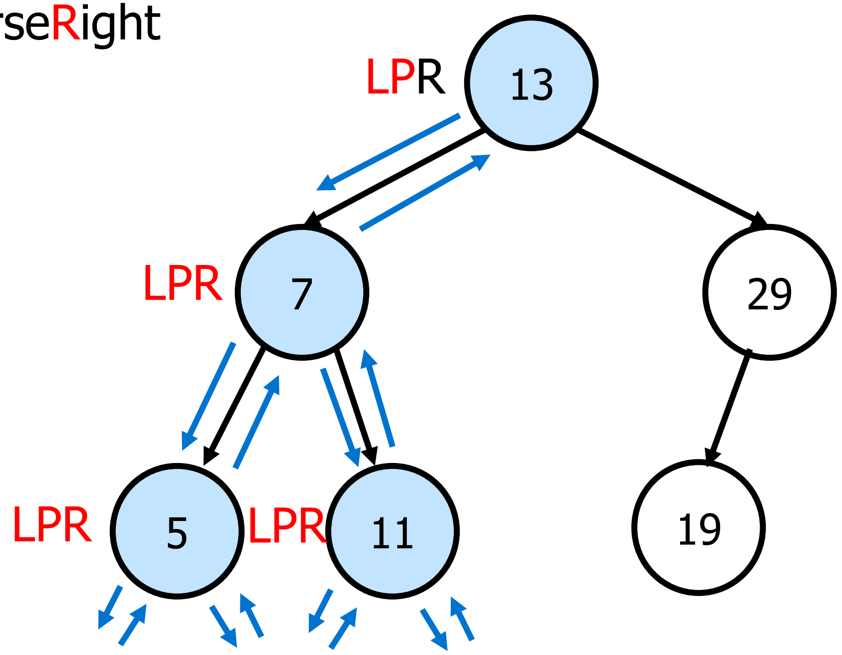


Traverse: 5, 7, 11

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

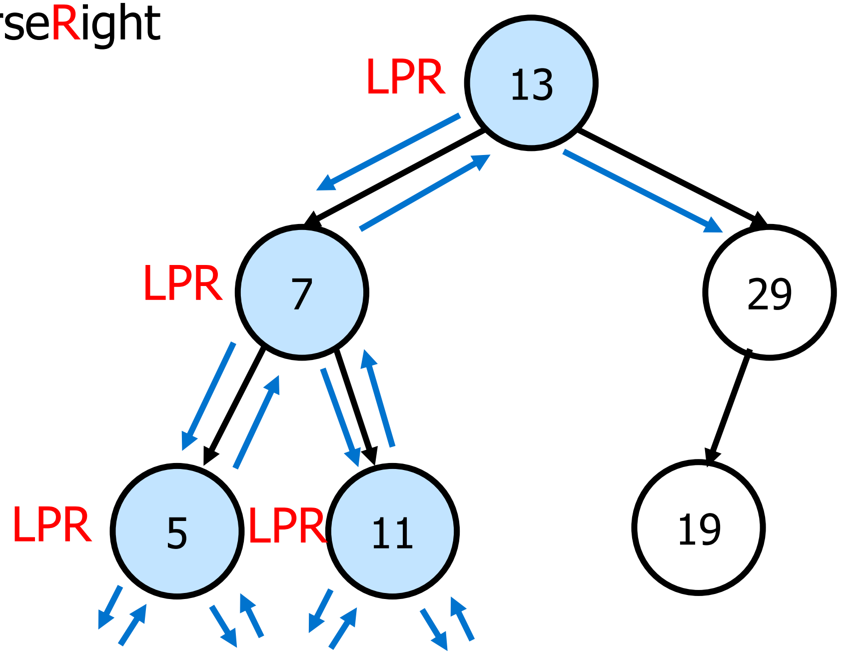


Traverse: 5, 7, 11, 13

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls



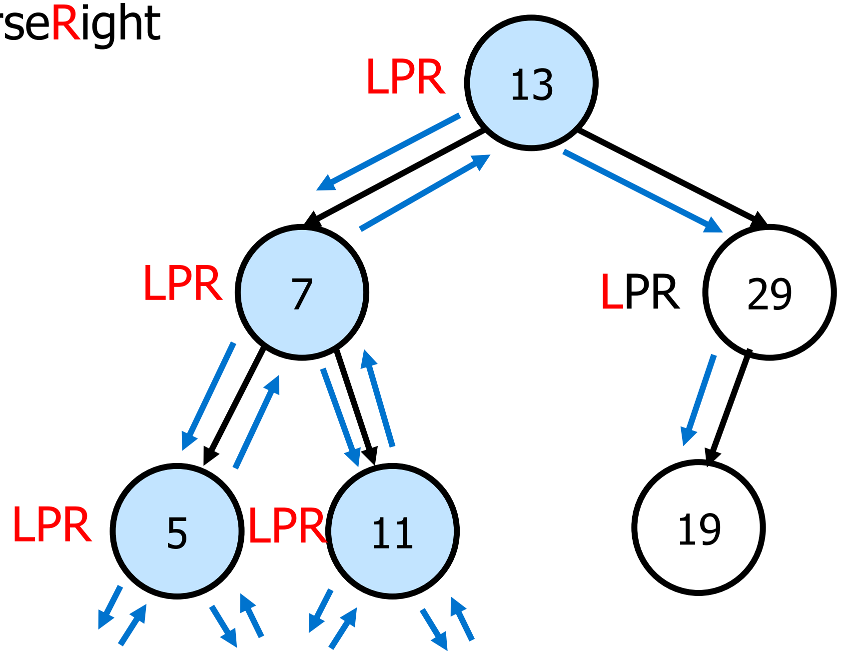
Traverse: 5, 7, 11, 13

WATERLOO | ENGINEERING

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

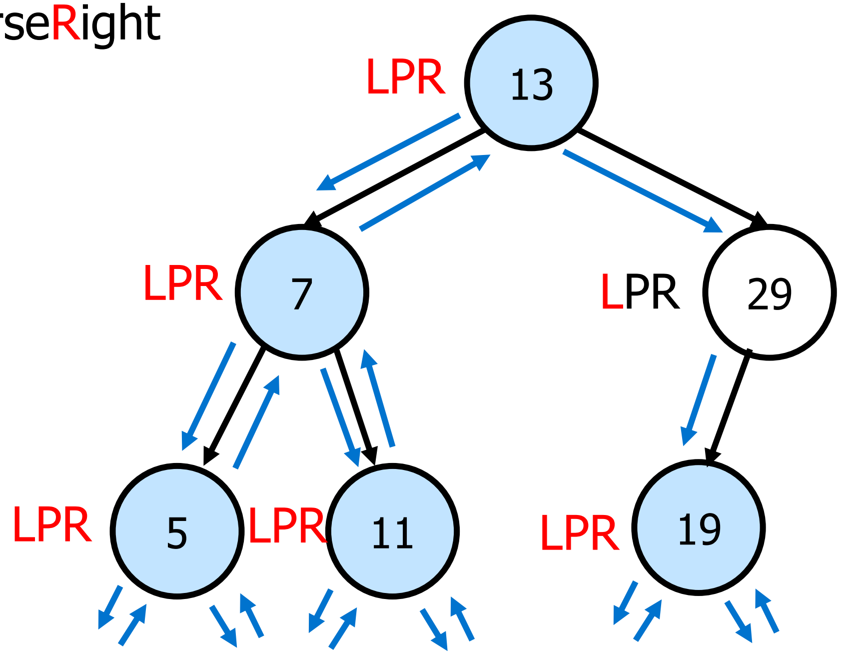


Traverse: 5, 7, 11, 13

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

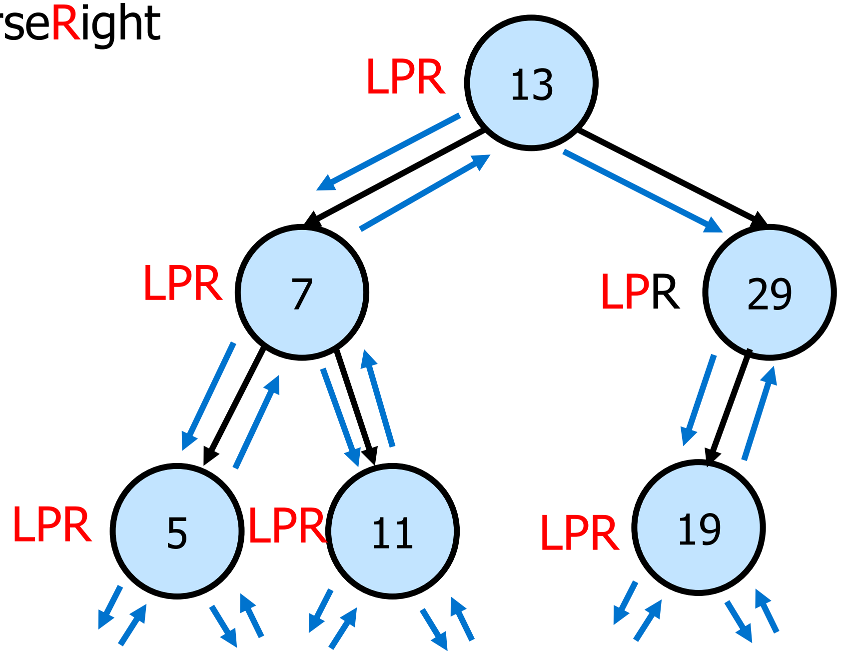


Traverse: 5, 7, 11, 13, 19

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls

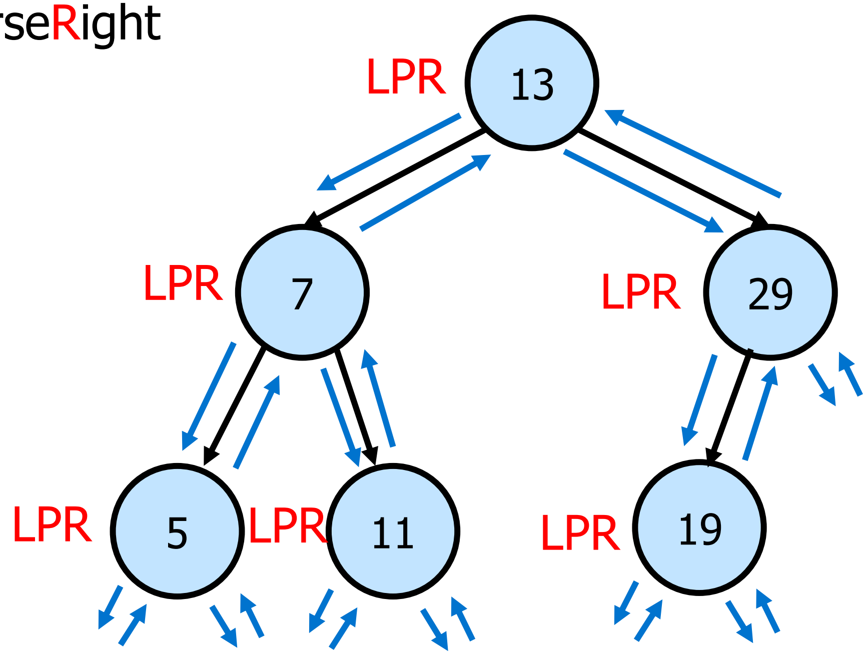


Traverse: 5, 7, 11, 13, 19, 29

In-order Recursive Tracing

LPR: RecurseLeft, Print, RecurseRight

Arrow: direction of calls



Traverse: 5, 7, 11, 13, 19, 29

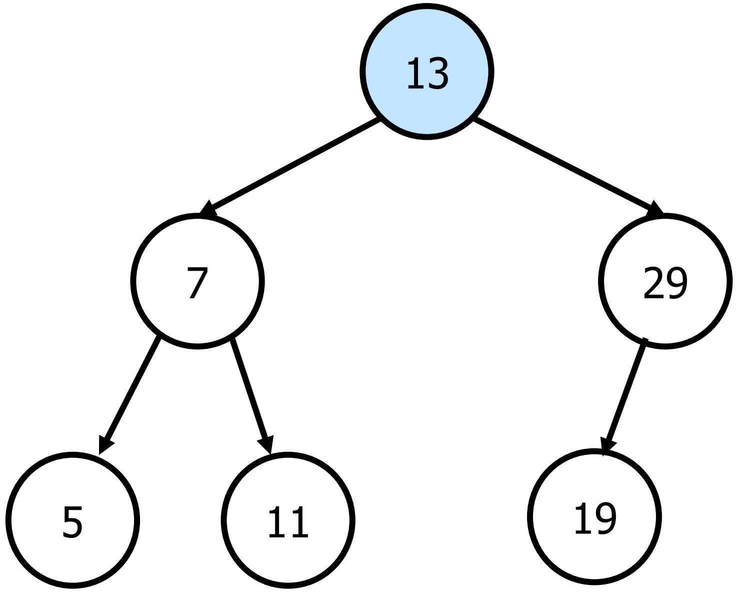
Iterative In-order Traversal Algorithm

- We can implement in-order traversal iteratively using a stack
  - In-order traversal: visit the left child, then the current node, and finally the right child
  - Algorithm:
    - Push the root onto the stack
    - Traverse as far left (of the root) as possible, pushing each node onto the stack
    - When we reach a leaf node, we pop a node from the stack, process it, and then move to its right child (if any)
    - Repeat until the stack is empty and all nodes are processed

Iterative In-order Traversal Algorithm

```
Create Stack s
Set curr as the root
while s is not empty or curr is not NULL:
      while curr is not NULL:
            s.push(curr)
            curr = curr->left
      curr = s.pop()
      process curr
      curr = curr->right
```

In-order Iterative Tracing

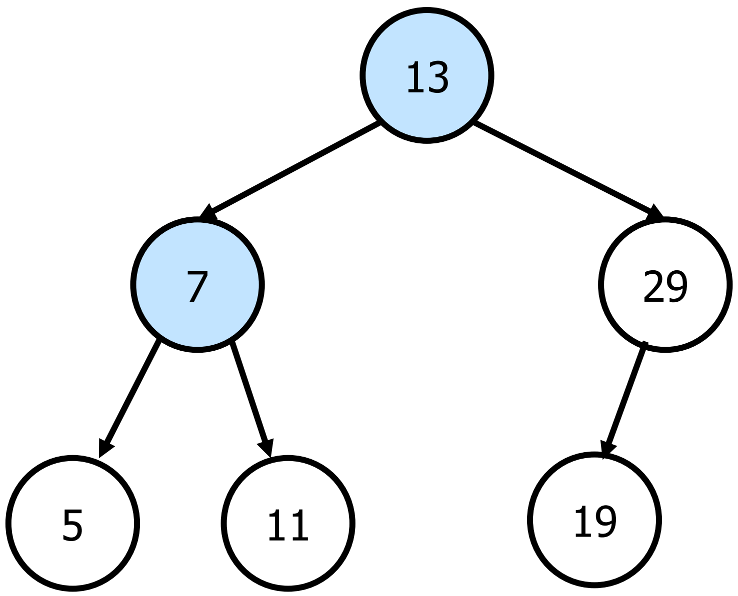


Traversal

Stack

13

In-order Iterative Tracing

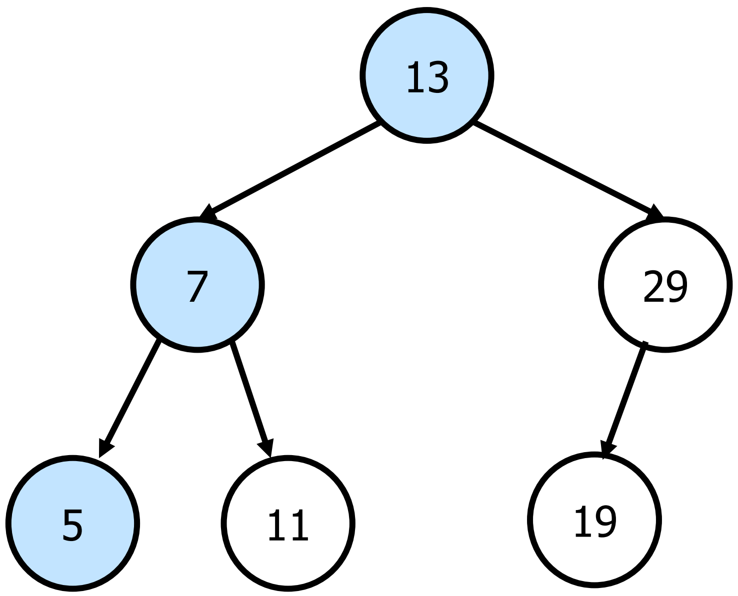


Traversal

Stack

7
13

In-order Iterative Tracing

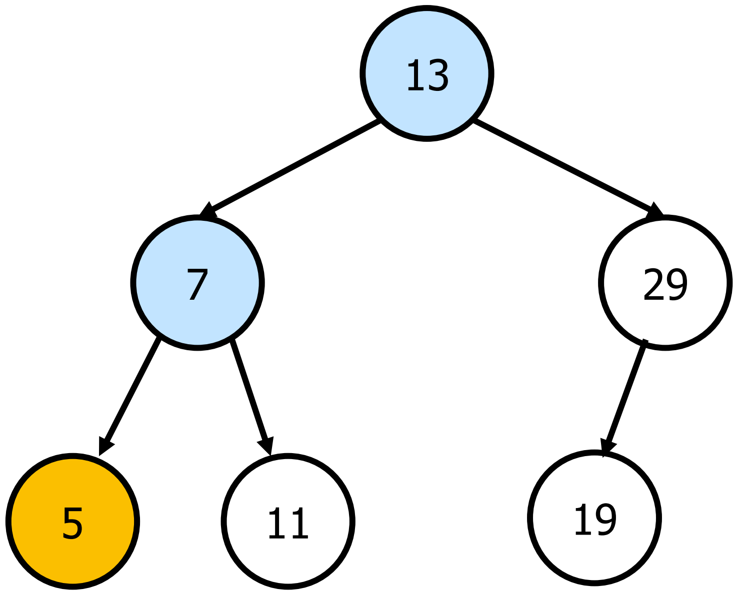


Traversal

Stack

5
7
13

In-order Iterative Tracing



Traversal

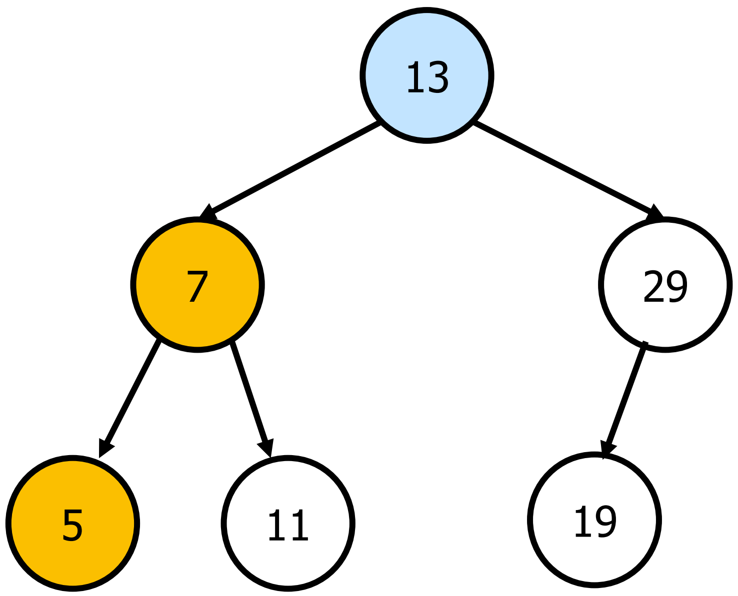
5

Stack

~~5~~
7
13

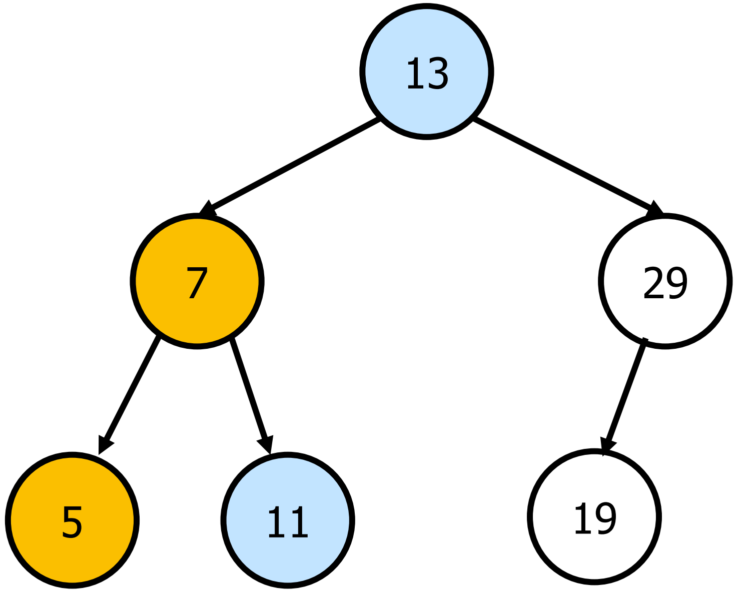
WATERLOO | ENGINEERING

In-order Iterative Tracing



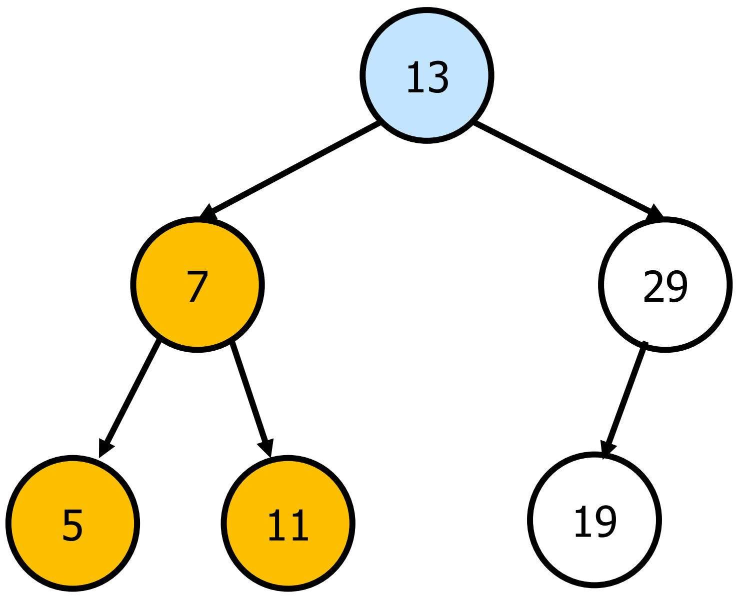
Traversal	Stack
5	~~5~~
7	~~7~~
	13

In-order Iterative Tracing



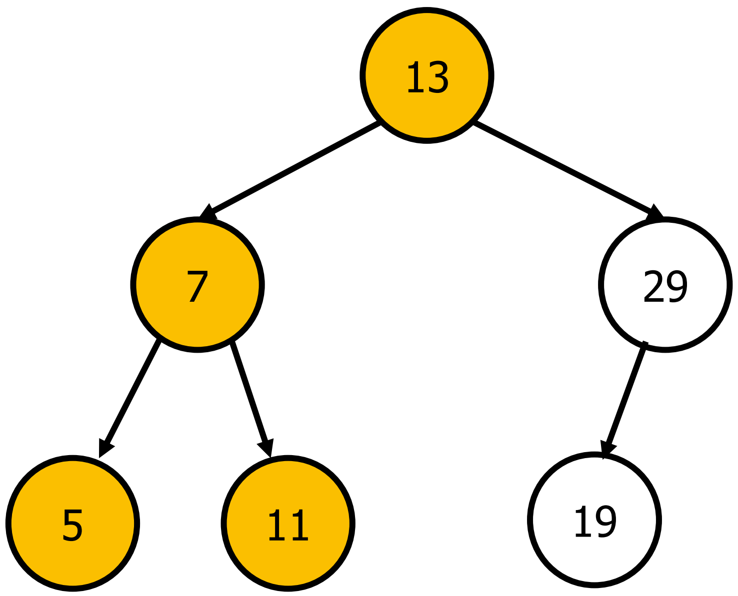
Traversal	Stack
5	
7	
	11
	~~5~~
	~~7~~
	13

In-order Iterative Tracing



Traversal	Stack
5	
7	
11	~~11~~
	~~5~~
	~~7~~
	13

In-order Iterative Tracing



Traversal

5

7

11

13

Stack

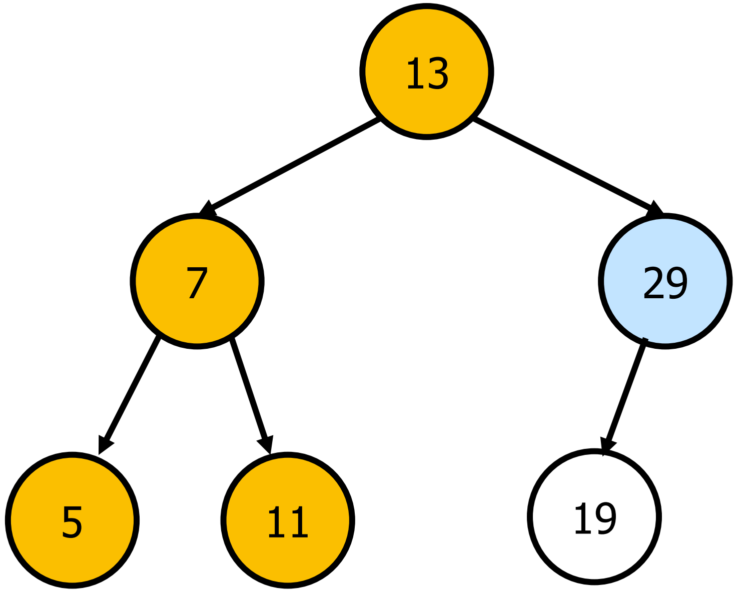
~~11~~

~~5~~

~~7~~

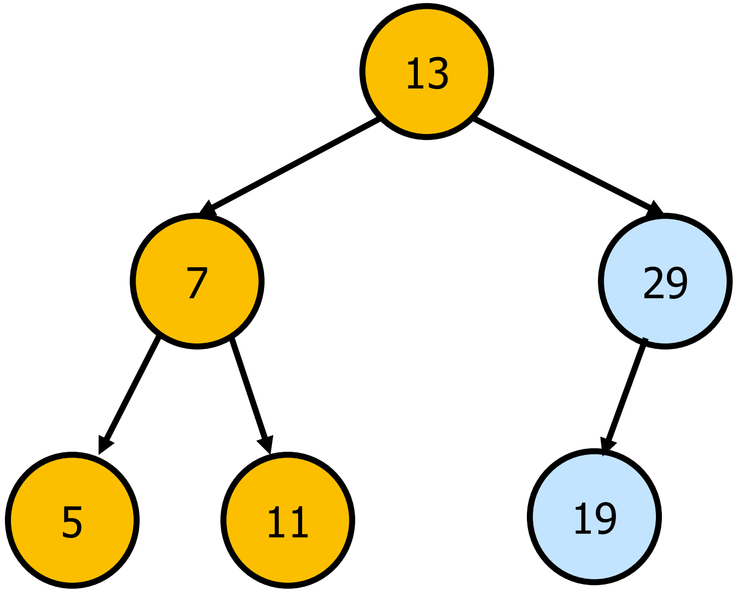
~~13~~

In-order Iterative Tracing



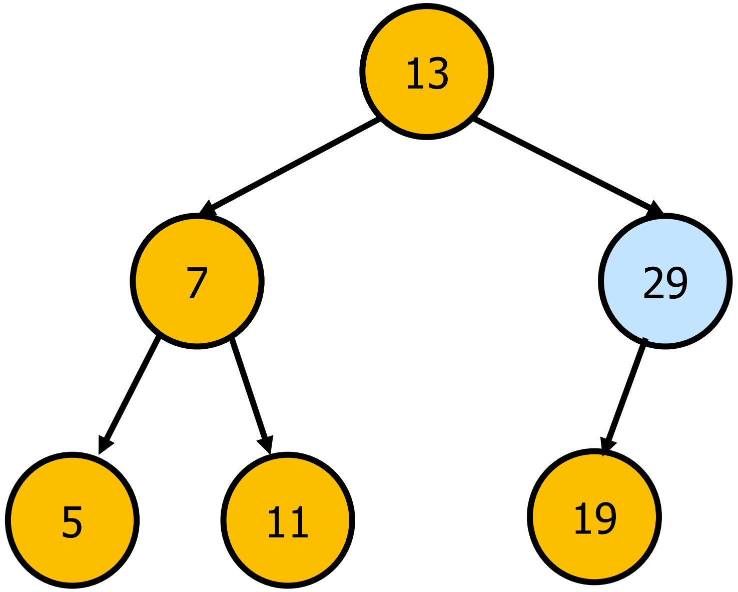
Traversal	Stack
5	
7	29
11	~~11~~
13	~~5~~
	~~7~~
	~~13~~

In-order Iterative Tracing



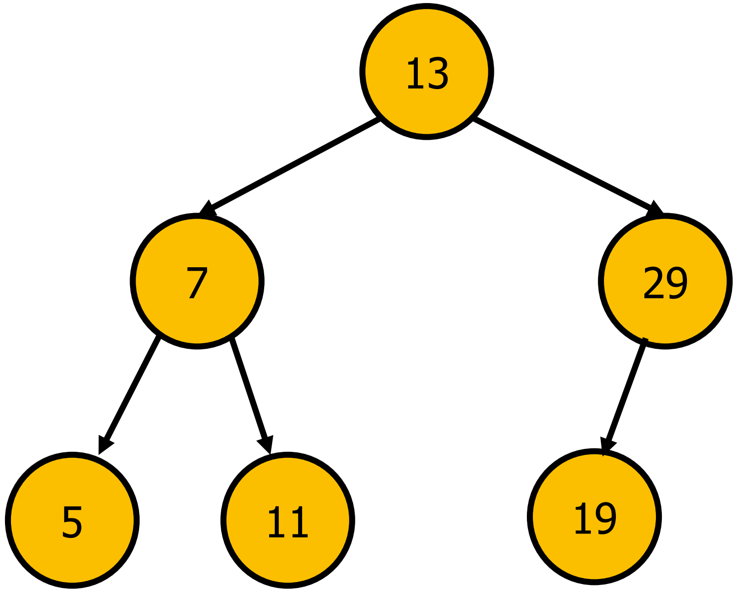
Traversal	Stack
5	19
7	29
11	~~11~~
13	~~5~~
	~~7~~
	~~13~~

In-order Iterative Tracing



Traversal	Stack
5	~~19~~
7	29
11	~~11~~
13	~~5~~
19	~~7~~
	~~13~~

In-order Iterative Tracing

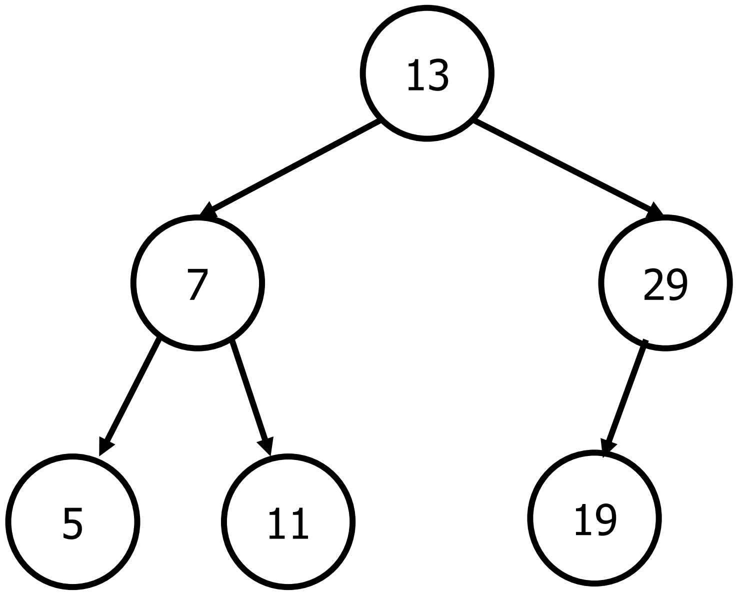


Traversal	Stack
5	~~19~~
7	~~29~~
11	~~11~~
13	~~5~~
19	~~7~~
29	~~13~~

Pre-order Traversal Algorithm

- Check if trying to traverse empty tree?
  - If so, doing nothing
  - If not, then …
    - Print out my value
    - traverse left subtree (recurse left)
    - traverse right subtree (recurse right)

Pre-order Traversal

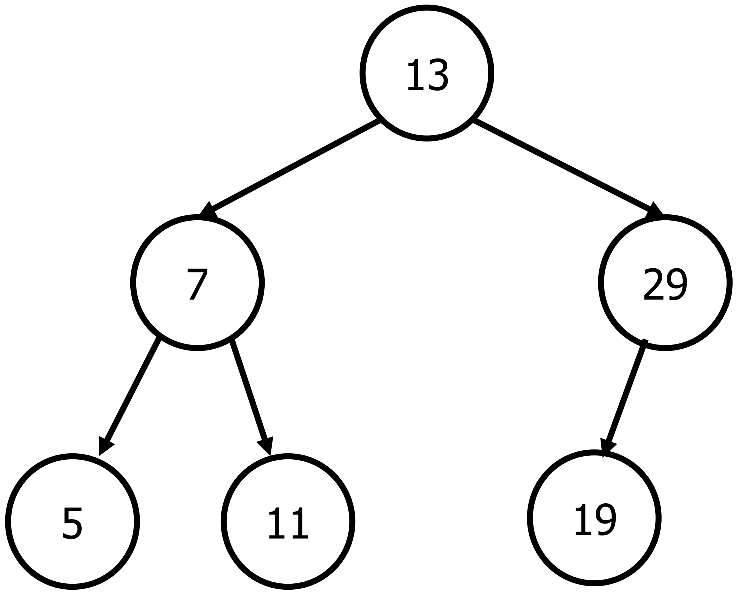


- Pre-order: 13, 7, 5, 11, 29, 19
  - Can anyone think why this might be useful?
  - Uniquely identifies a BST.
    - Save and restore: get exactly the same tree

Post-order Traversal Algorithm

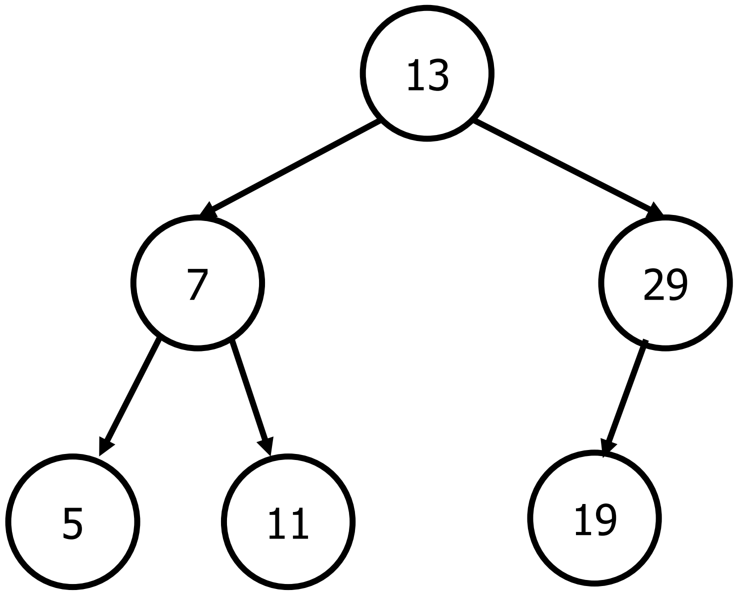
- Check if trying to traverse empty tree?
  - If so, doing nothing
  - If not, then …
    - traverse left subtree (recurse left)
    - traverse right subtree (recurse right)
    - Print out my value

Post-order Traversal



- Post-order: 5, 11, 7, 19, 29, 13
  - When is this useful?
    - Freeing the tree's memory?

Level-order Traversal

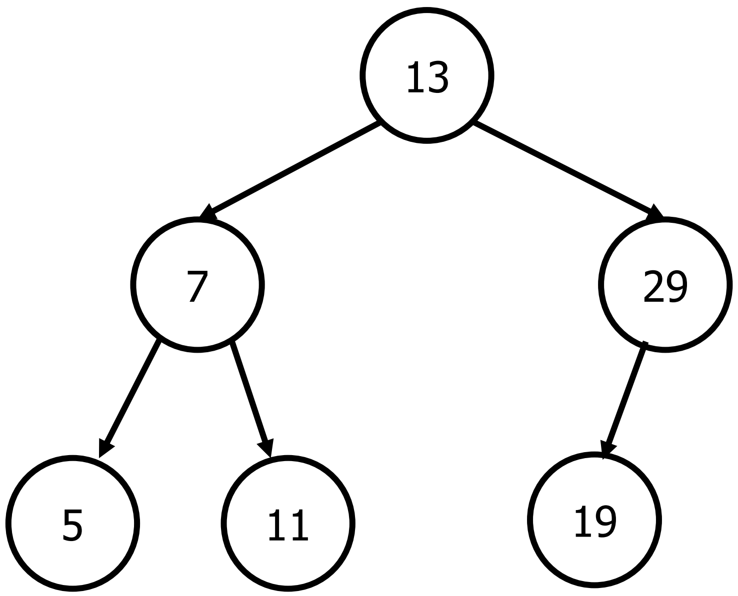


- Level-order: 13, 7, 29, 5, 11, 19
  - Useful when you want "hierarchical order"
  - How do we come up with this?

Level-order Traversal Algorithm

```
Create Queue q
Add root to q
while q is not empty:
     Node curr = q.dequeue()
     if curr->left is not NULL:
          q.enqueue(curr->left)
     if curr->right is not NULL:
          q.enqueue(curr->right)
```

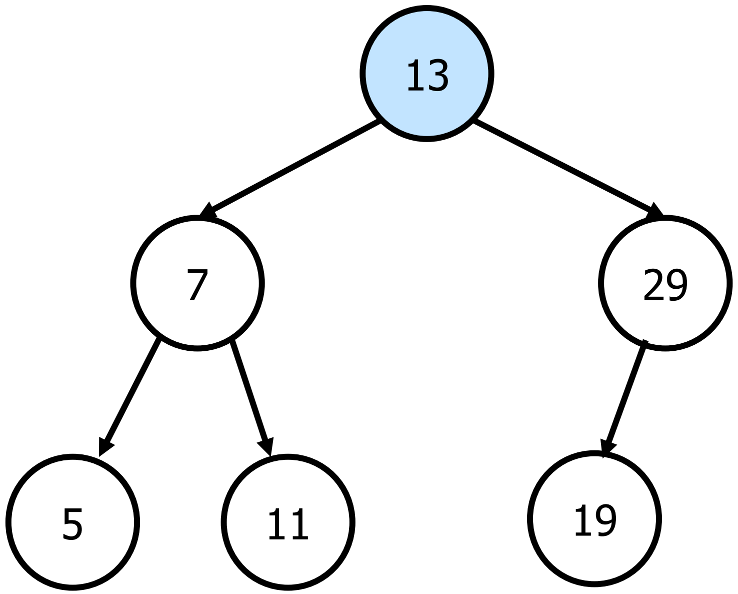
Level-order Iterative Tracing



Traversal

Queue

Level-order Iterative Tracing

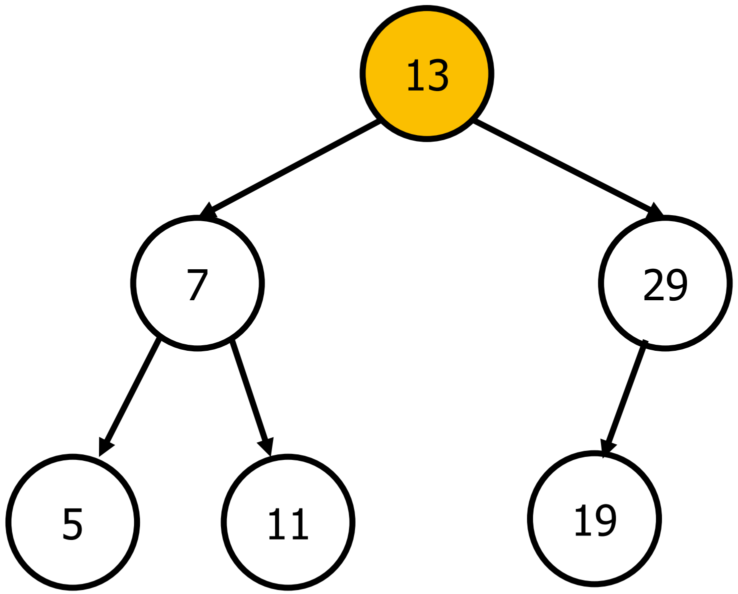


Traversal

Queue

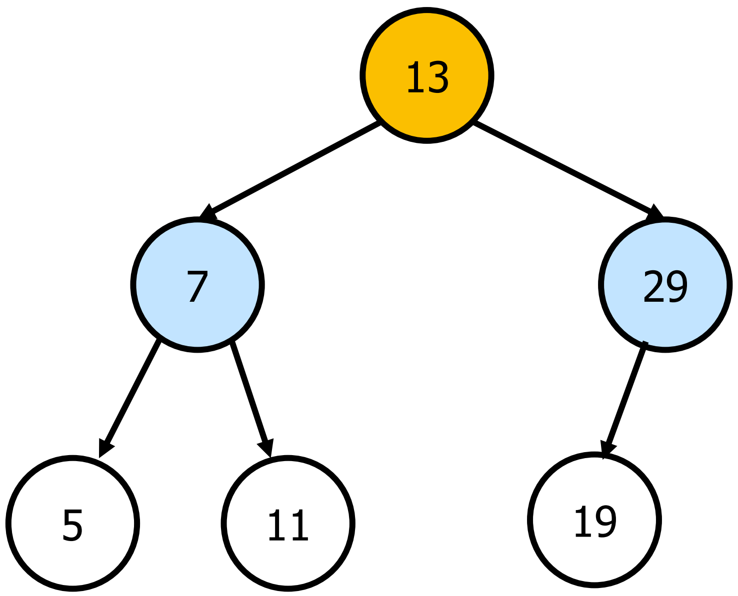
13

Level-order Iterative Tracing



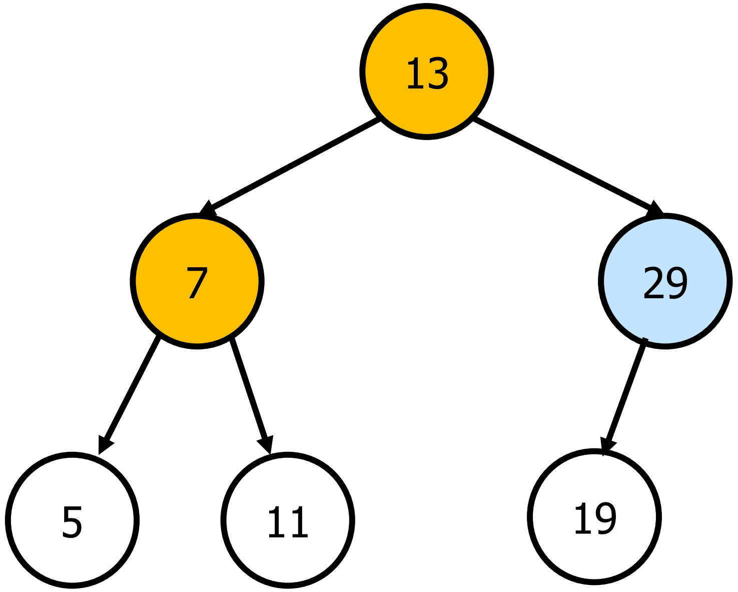
Traversal	Queue
13	~~13~~

Level-order Iterative Tracing



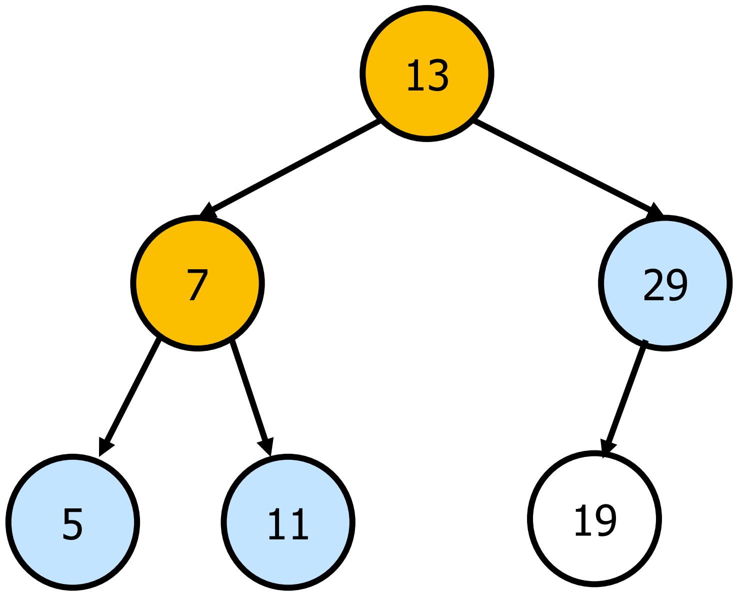
Traversal	Queue
13	~~13~~
	7
	29

Level-order Iterative Tracing



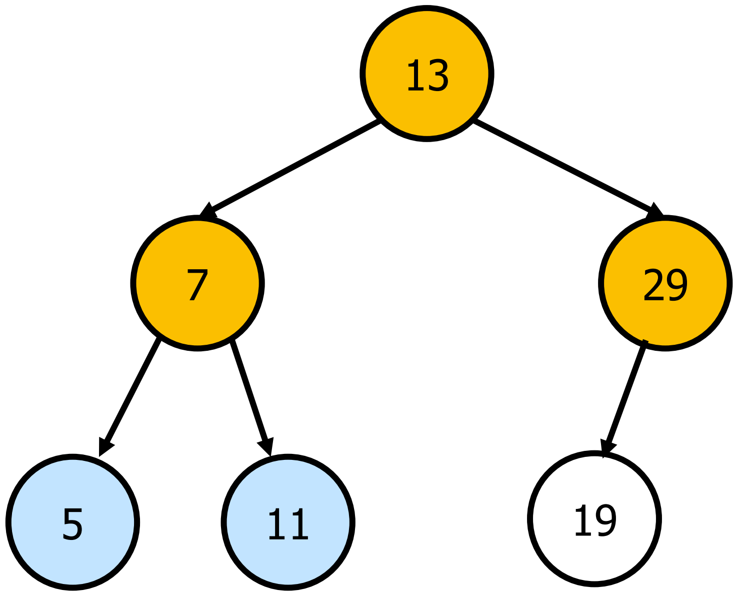
Traversal	Queue
13	~~13~~
7	~~7~~
	29

Level-order Iterative Tracing



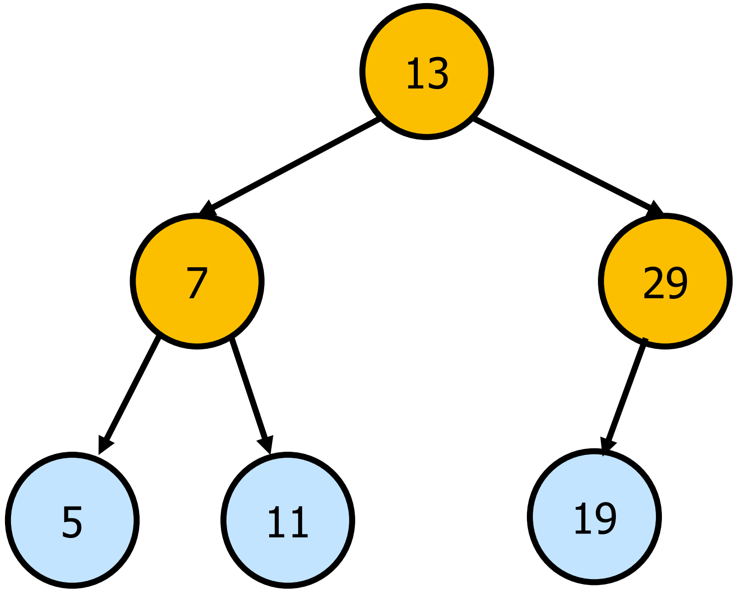
Traversal	Queue
13	~~13~~
7	~~7~~
	29
	5
	11

Level-order Iterative Tracing



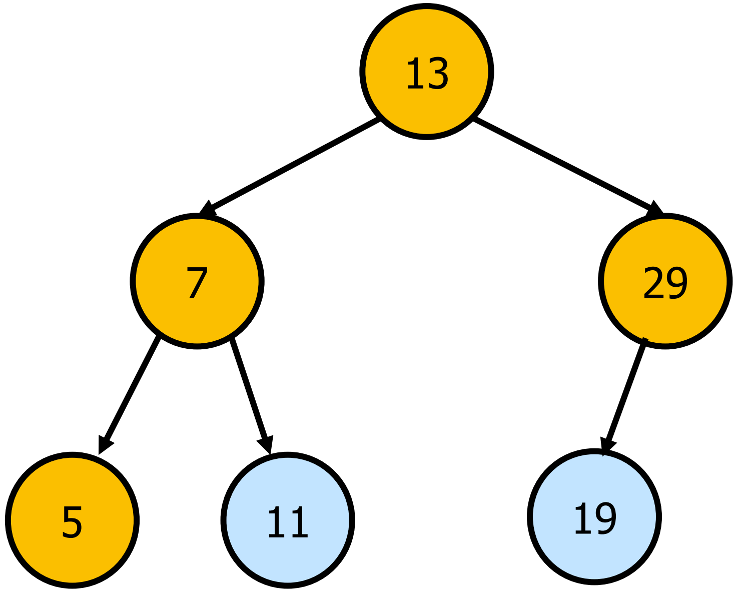
Traversal	Queue
13	~~13~~
7	~~7~~
29	~~29~~
	5
	11

Level-order Iterative Tracing



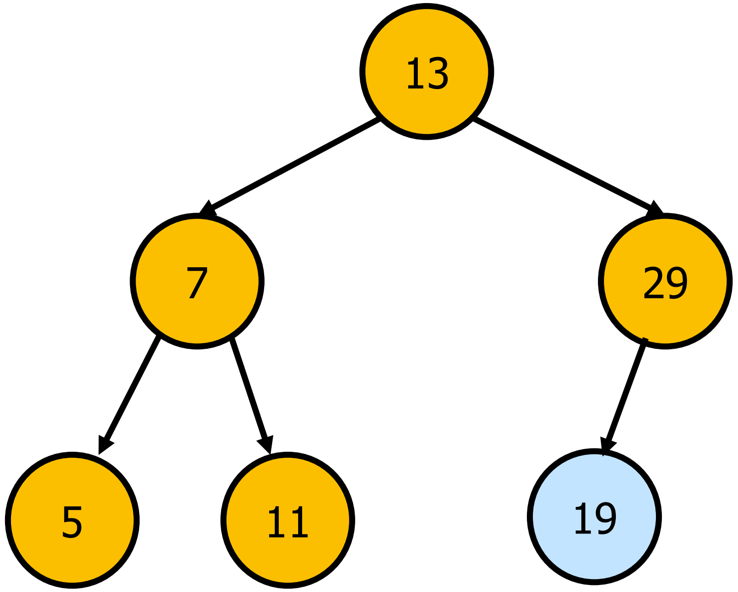
Traversal	Queue
13	~~13~~
7	~~7~~
29	~~29~~
	5
	11
	19

Level-order Iterative Tracing



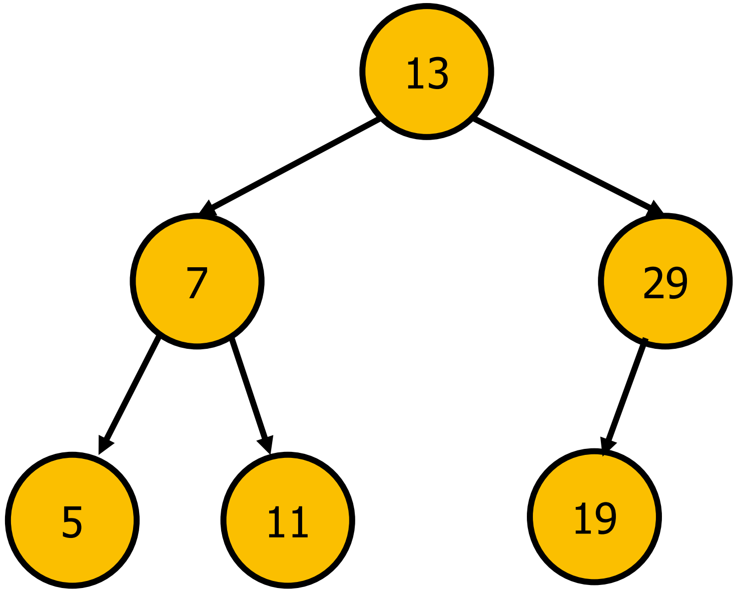
Traversal	Queue
13	~~13~~
7	~~7~~
29	~~29~~
5	~~5~~
	11
	19

Level-order Iterative Tracing



Traversal	Queue
13	~~13~~
7	~~7~~
29	~~29~~
5	~~5~~
11	~~11~~
	19

Level-order Iterative Tracing



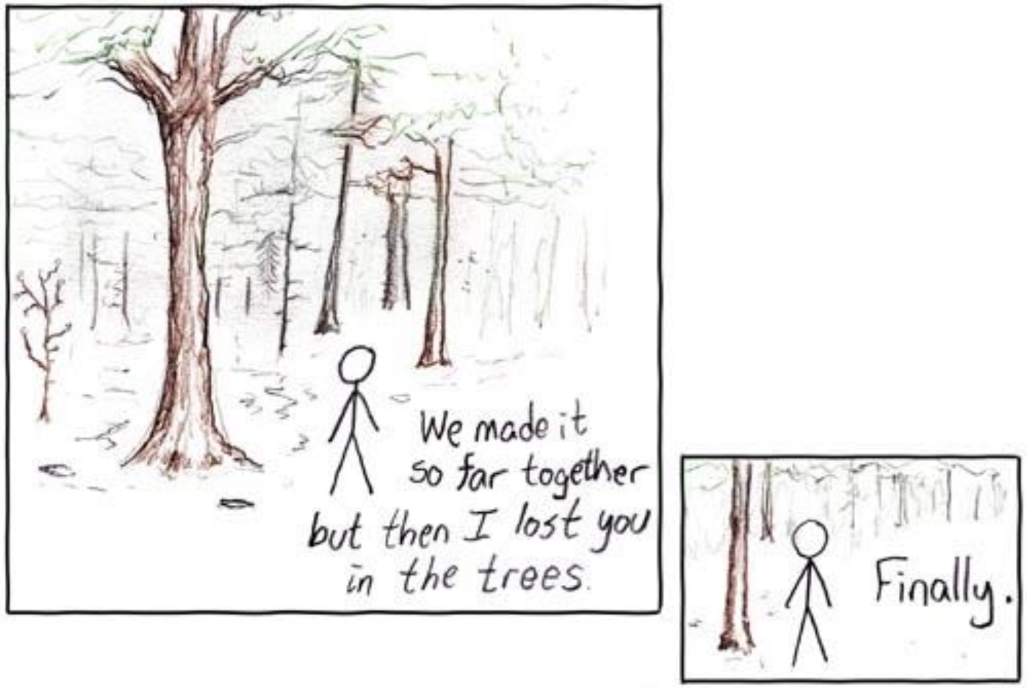
Traversal	Queue
13	~~13~~
7	~~7~~
29	~~29~~
5	~~5~~
11	~~11~~
19	~~19~~

Wrap Up

- In this lecture we talked about
  - Trees: binary trees, binary search trees
  - Different tree traversals
- Next up
  - BST operations & efficiency

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 4.1 – 4.3



source

Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)
  - Mary Hudachek-Buswell (Gatech)