

# Data Structures

## Tree Traversal

CS 225

September 16, 2026

Brad Solomon



UNIVERSITY OF  
**ILLINOIS**  
URBANA - CHAMPAIGN

Department of Computer Science



# Learning Objectives

Review and expand on foundational tree terminology

Discuss the tree ADT

Explore tree implementation details

# Binary Tree

Join Code: 225

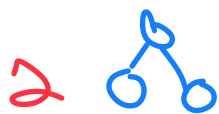


Lets review our understanding of binary tree terminology!

**How many leaves are there in a perfect tree of height  $h$ ?**

7 0 1 node = = height 0

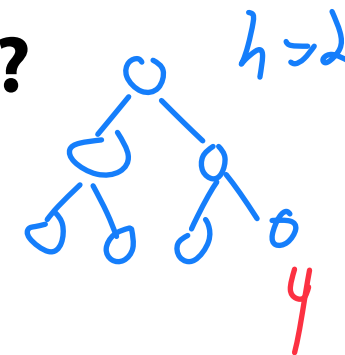
A)  $h$  leaves



height = 1

C)  $2^{h-1}$  leaves

4 2%



B)  $2^h$  leaves

5 5%

D)  $h^2$  leaves



# Binary Tree

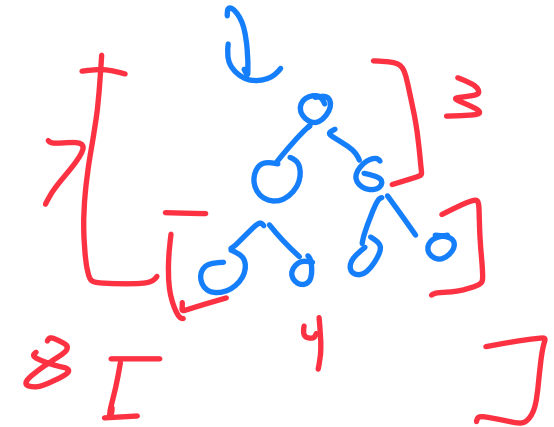
Lets review our understanding of binary tree terminology!

**What is the upperbound on # of nodes in a height  $h$  complete tree?**

is the # of nodes in a perfect tree

A)  $2^{h+1} + 1$  nodes

C)  $2^h + 1$  nodes



B)  $2^h - 1$  nodes

100%

D)  $2^{h+1} - 1$  nodes

70%



# Binary Tree

Lets review our understanding of binary tree terminology!

**What is the lowerbound on # of nodes in a height h complete tree?**

$$2^{h+1} - 1 \rightarrow 2^{h+1-1} - 1$$

$$2^h - 1 + 1$$

Diagram showing a node with two children, labeled with  $h=h-1$  and  $h$ .

A)  $2^{h+1} + 1$  nodes

C)  $2^h + 1$  nodes

40%

B)  $2^h$  nodes

60%

D)  $2^{h+1} - 1$  nodes

# Binary Tree: Practicing Proofs

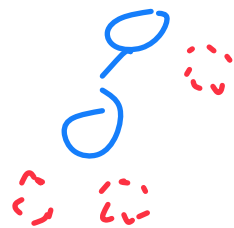
**Theorem:** If there are  $n$  objects in our representation of a binary tree, then there are  $n+1$  NULL pointers.

# Binary Tree: Practicing Proofs

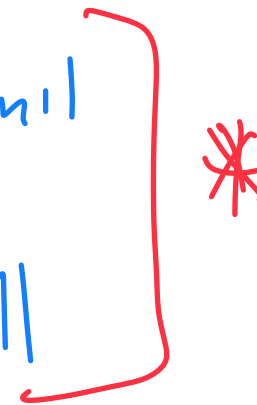
**Theorem:** If there are  $n$  objects in our representation of a binary tree, then there are  $n+1$  NULL pointers.

Base Case:

Null empty tree



0 nodes, 1 null  
1 node, 2 null  
2 nodes, 3 null



# Binary Tree: Practicing Proofs

**Theorem:** If there are  $n$  objects in our representation of a binary tree, then there are  $n+1$  NULL pointers.

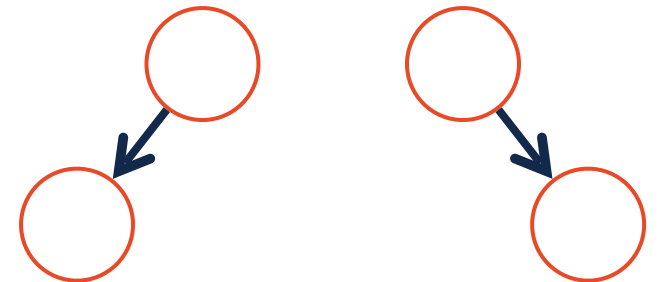
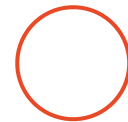
Base Case:

Let  $F(n)$  be the max number of NULL pointers in a tree of  $n$  nodes

$N=0$  has one NULL

$N=1$  has two NULL

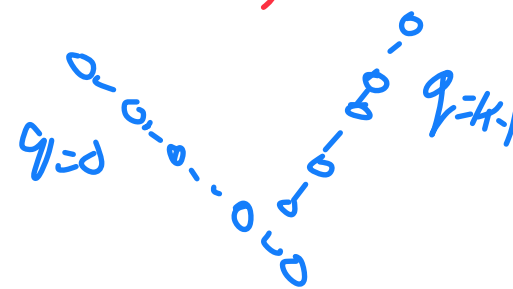
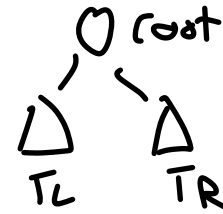
$N=2$  has three NULL



**Theorem:** If there are  $n$  objects in our representation of a binary tree, then there are  $n+1$  NULL pointers.

Induction Step: Assume claim true for up to  $K-1$  nodes, prove  $K$

By def  $T = \text{root}, T_L, T_R$  ( $K$  nodes total)



Let  $q$  be # nodes in  $T_L$ :  $0 \leq q \leq K-1$

$\hookrightarrow$  By IH  $T_L$  has  $q+1$  null values

$T_R$  has  $K-q-1$  nodes

$\hookrightarrow$  By IH  $T_R$  has  $(K-q-1)+1$  null values

So my total Nulls we  $\underbrace{K-q}_{\text{from } T_R} + \underbrace{q+1}_{\text{from } T_L} = K+1$   $\smile$

**Theorem:** If there are  $n$  objects in our representation of a binary tree, then there are  $n+1$  NULL pointers.



**IS:** Assume claim is true for  $|T| \leq k - 1$ , prove true for  $|T| = k$

By def,  $T = r, T_L, T_R$ . Let  $q$  be the # of nodes in  $T_L$

Since  $r$  exists,  $0 \leq q \leq k - 1$ . By IH,  $T_L$  has  $q + 1$  NULL

All nodes not in  $r$  or  $T_L$  exist in  $T_R$ . So  $T_R$  has  $k - q - 1$  nodes

$k - q - 1$  is also smaller than  $k$  so by IH,  $T_R$  has  $k - q$  NULL

Total number of NULL is the sum of  $T_L$  and  $T_R$ :  $q + 1 + k - q = k + 1$

# Alternate proof (# of null ptrs)

**Theorem:** If there are  $n$  objects in our representation of a binary tree, then there are  $n+1$  NULL pointers.

Proof -

We have  $n$  objects  $\Rightarrow 2n$  pointers.

Each pointer either points to (exactly 1) node or null.

Each node has exactly 1 incoming pointer (from its parent)

There are  $(n-1)$  children in total  $\Rightarrow$  total  $(n-1)$  pointers pointing to them

So, there are  $2n - (n-1) = (n + 1)$  nullptrs.

# Tree ADT

Insert

Remove

Traverse

Find

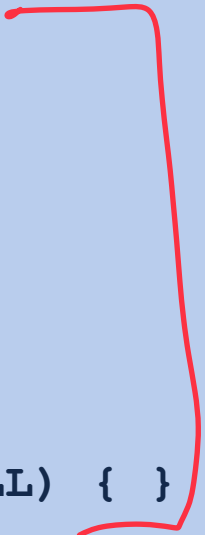
Constructor

My standard interface expectations

one exception (relevant when non-linear)  
↳ Iterator!

## List.h

```
1 #pragma once
2
3 template <typename T>
4 class List {
5     public:
6         /* ... */
7     private:
8         class ListNode {
9             T & data;
10
11             ListNode * next;
12
13
14             ListNode(T & data) :
15                 data(data), next(NULL) { }
16         };
17
18
19         ListNode *head_;
20         /* ... */
21 };
22
23
```



## Tree.h

```
1 #pragma once
2
3 template <typename T>
4 class BinaryTree {
5     public:
6         /* ... */
7     private:
8         class TreeNode {
9             T & data;
10
11             TreeNode * left;
12             TreeNode * right;
13
14             TreeNode(
15
16             )
17
18         };
19
20         TreeNode *root_;
21         /* ... */
22 };
23
```

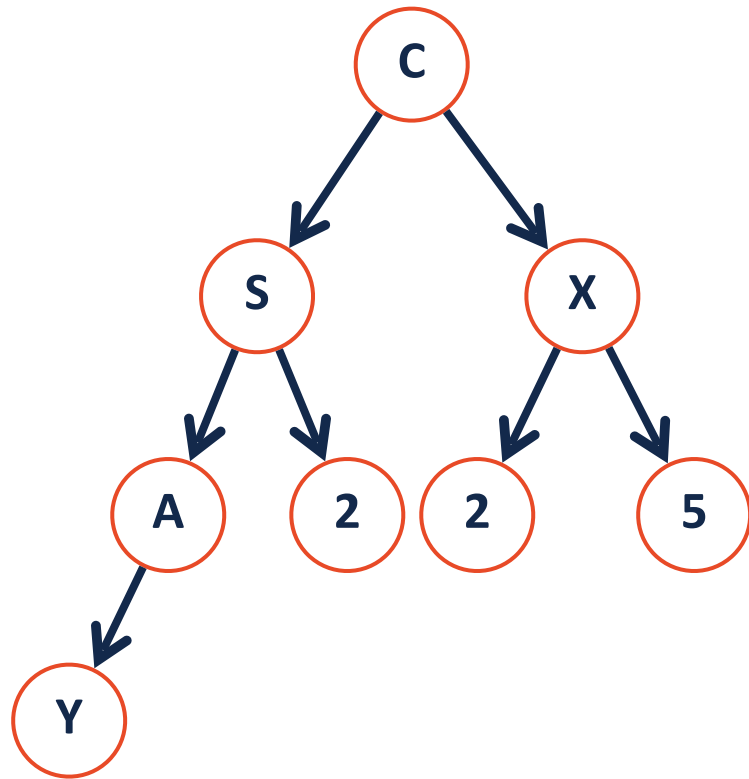
## List.h

```
1 #pragma once
2
3 template <typename T>
4 class List {
5     public:
6         /* ... */
7     private:
8         class ListNode {
9             T & data;
10
11             ListNode * next;
12
13
14             ListNode(T & data) :
15                 data(data), next(NULL) { }
16         };
17
18
19
20
21     ListNode *head_;
22     /* ... */
23 };
```

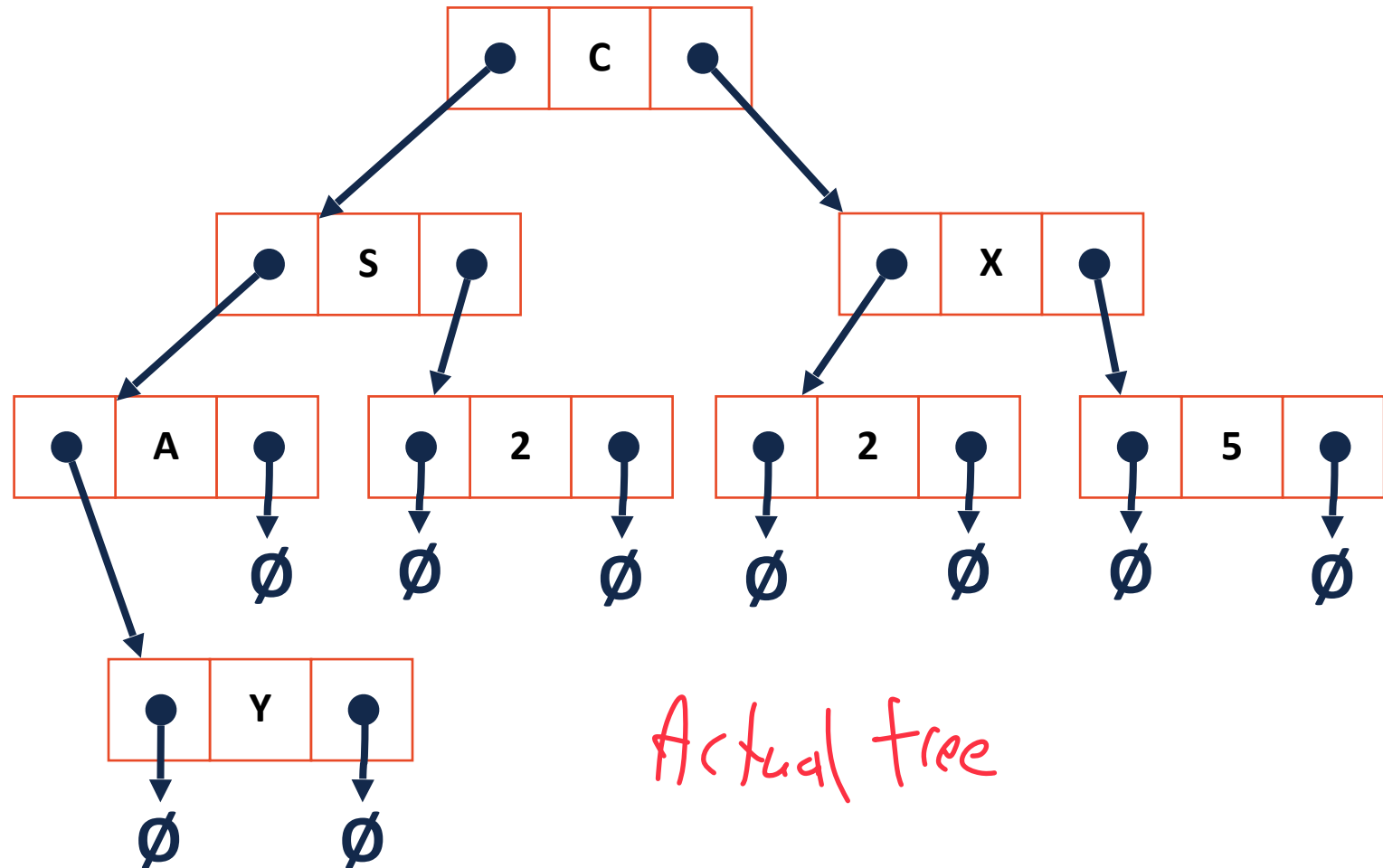
## Tree.h

```
1 #pragma once
2
3 template <typename T>
4 class BinaryTree {
5     public:
6         /* ... */
7     private:
8         class TreeNode {
9             T & data;
10
11             TreeNode * left;
12
13             TreeNode * right;
14
15             TreeNode(T & data) :
16                 data(data), left(NULL),
17                 right(NULL) { }
18         };
19
20
21     TreeNode *root_;
22     /* ... */
23 };
```

# Visualizing trees

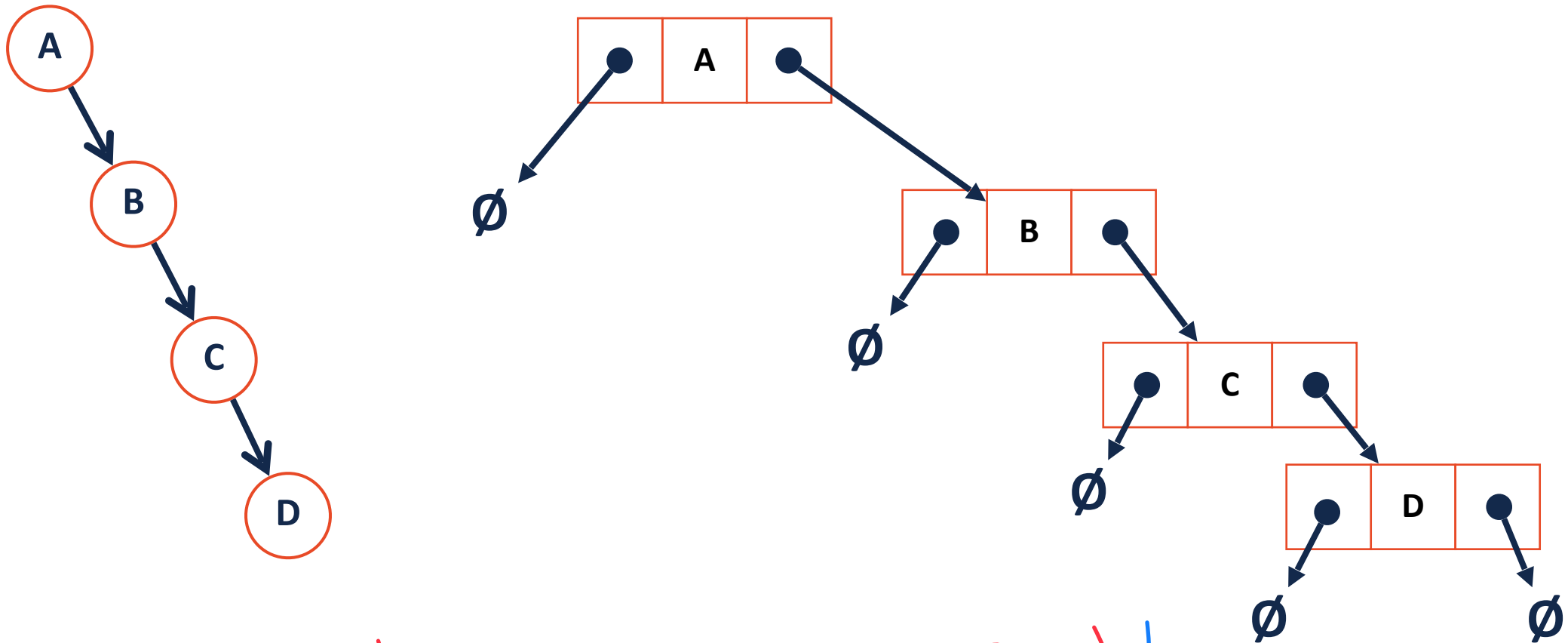


What we  
show on  
slide



Actual tree

# Tree Insert / Remove acts like Linked List

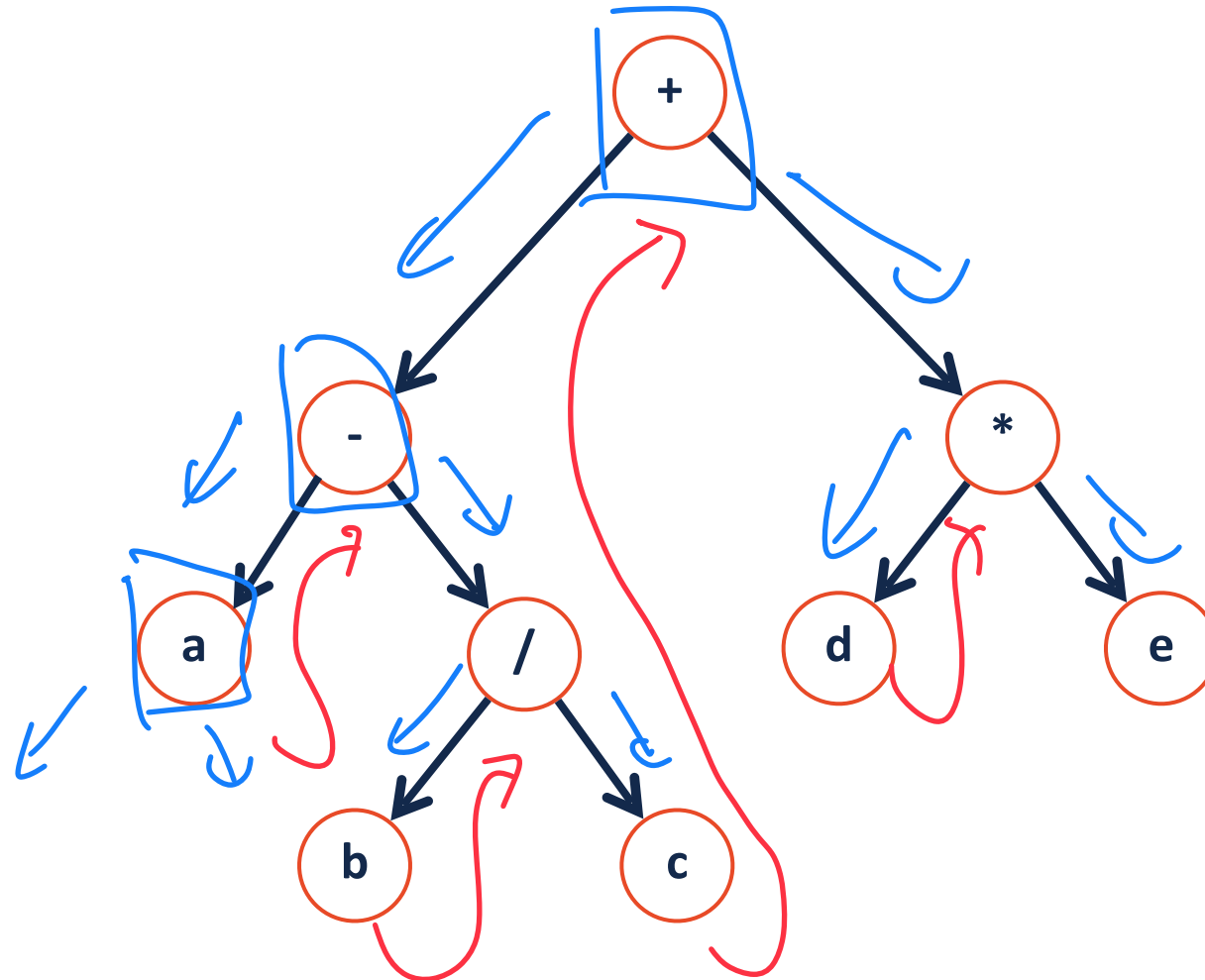


When inserting, insert @ node and specify left right

# Tree Traversal

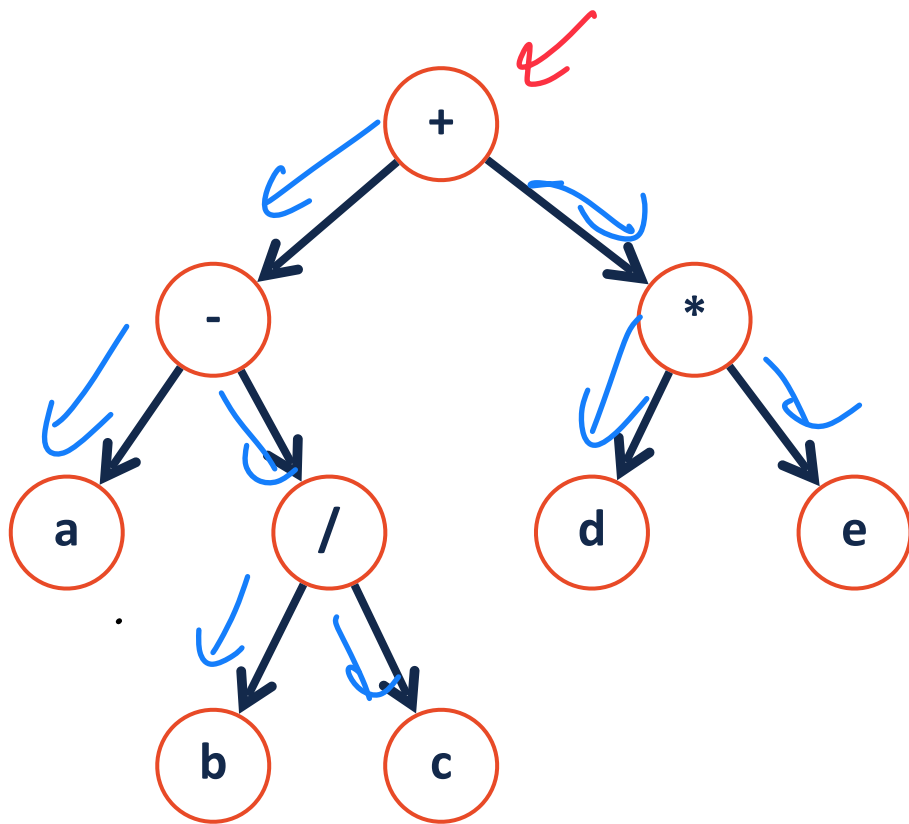
A **traversal** of a tree  $T$  is an ordered way of visiting every node once.

Look at cur  
Recurse left  
Recurse right



# Traversals

Recursive Problem



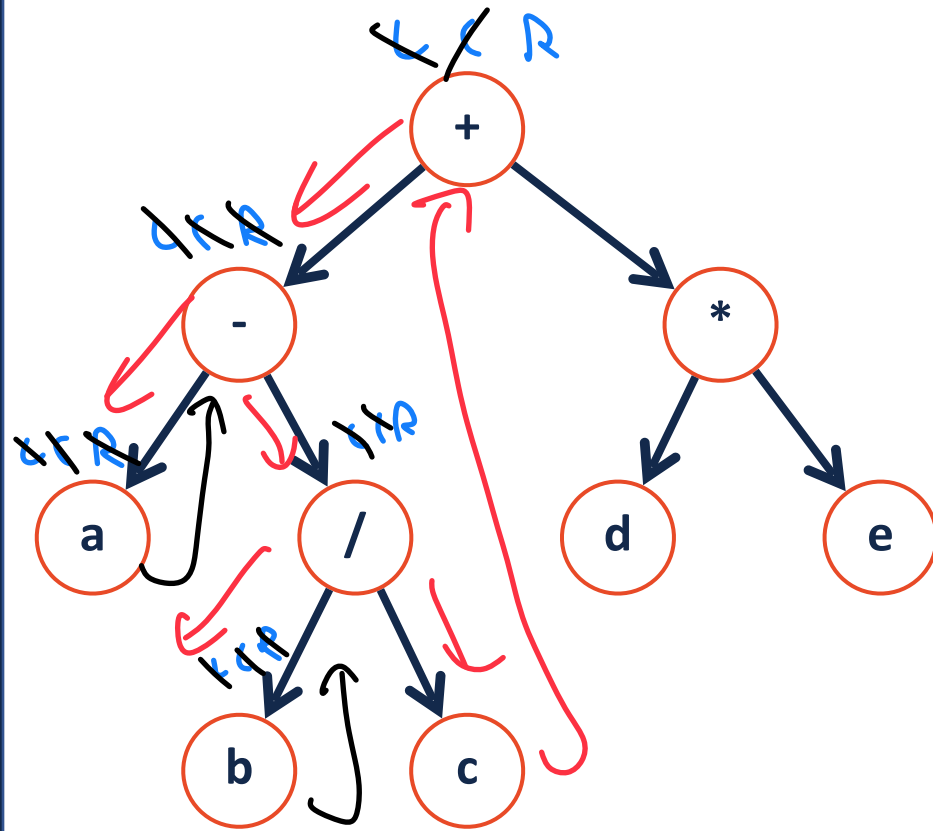
+ - a / b c \* d e

```

1  template<class T>
2  void BinaryTree<T>:: Pre Order(TreeNode * root)
3  {
4      // my base case
5      // everything inside this
6      // if is guaranteed to
7      // have a non-null Node
8
9      print (root);
10
11     Pre order (root -> left);
12
13     Pre order (root -> right);
14
15     // equiv to
16     // if (root != nullptr)
17
18     // Alt base case
19     // if root is leaf {
20     // if root->left exists, call recursion
21 }

```

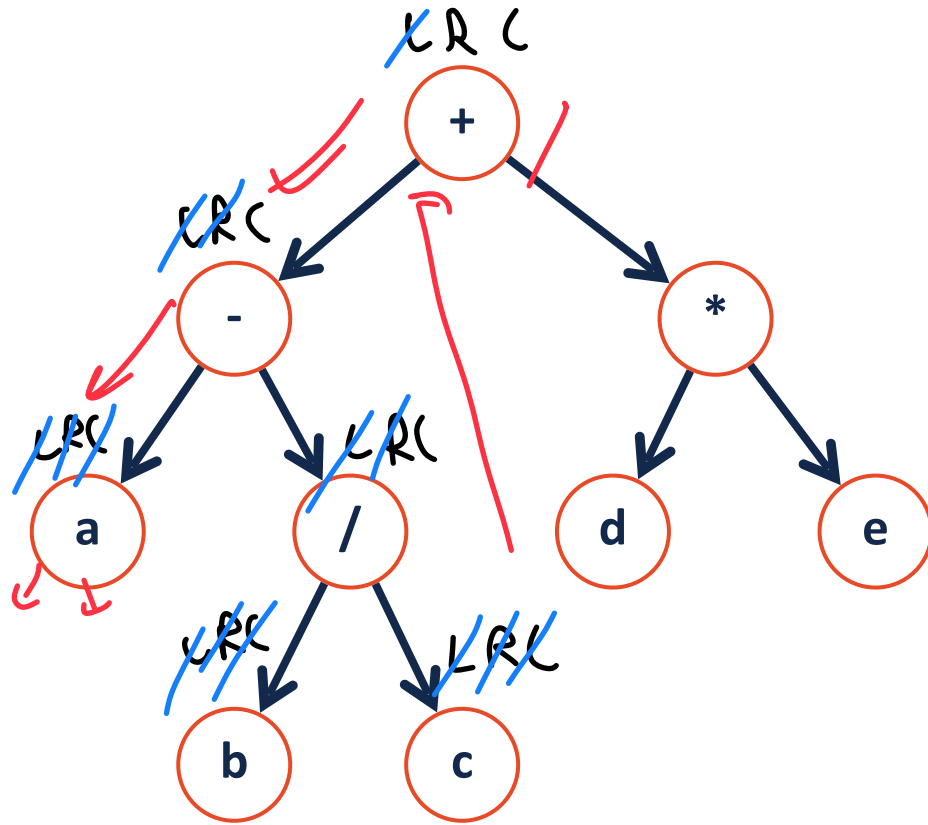
# Traversals



$a - b / c + d * e$

```
1 template<class T>
2 void BinaryTree<T>::in_Order(TreeNode * root)
3 {
4     if (root) {
5         _____;
6         in_Order(root->left);
7         print (root);
8         in_Order(root->right);
9         _____;
10    }
11
12
13
14
15
16
17 }
18
19
20
21 }
```

# Traversals



a b c / - d e \* +

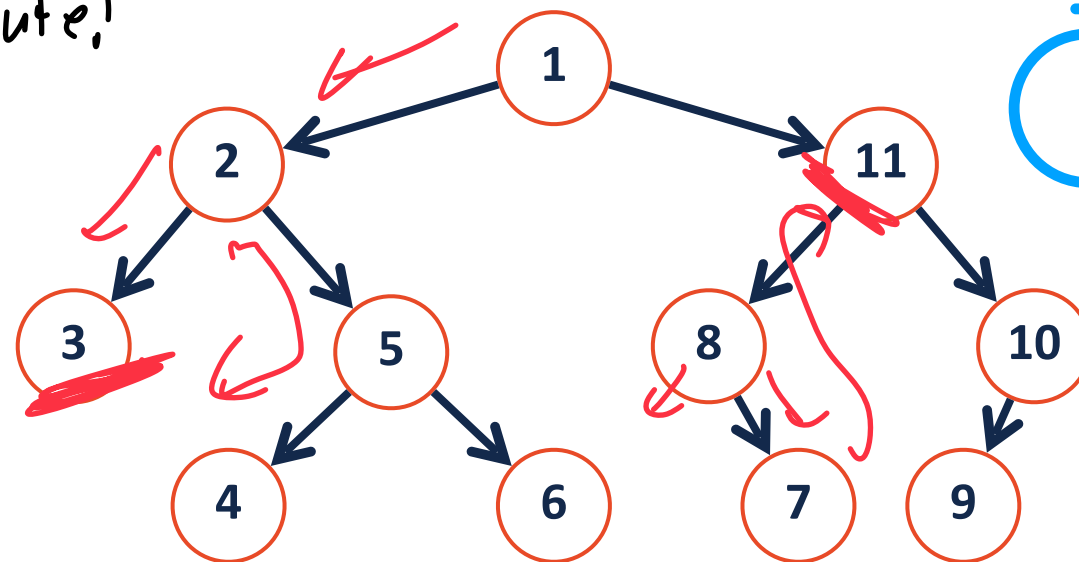
```
1 template<class T>
2 void BinaryTree<T>::PostOrder(TreeNode * root)
3 {
4     if (root) {
5         _____;
6         _____Order(root->left);
7         _____;
8         _____Order(root->right);
9         _____print(root);
10    }
11
12
13
14
15
16
17 }
18
19
20
21 }
```

# Tree Traversals

20 more minute!



Practice on own!



**Pre-order:**

1 2 3 5 4 6 11 8 7 10 9

**In-order:**

3 2 4 5 6 2 8 7 11 9 10

**Post-order:**

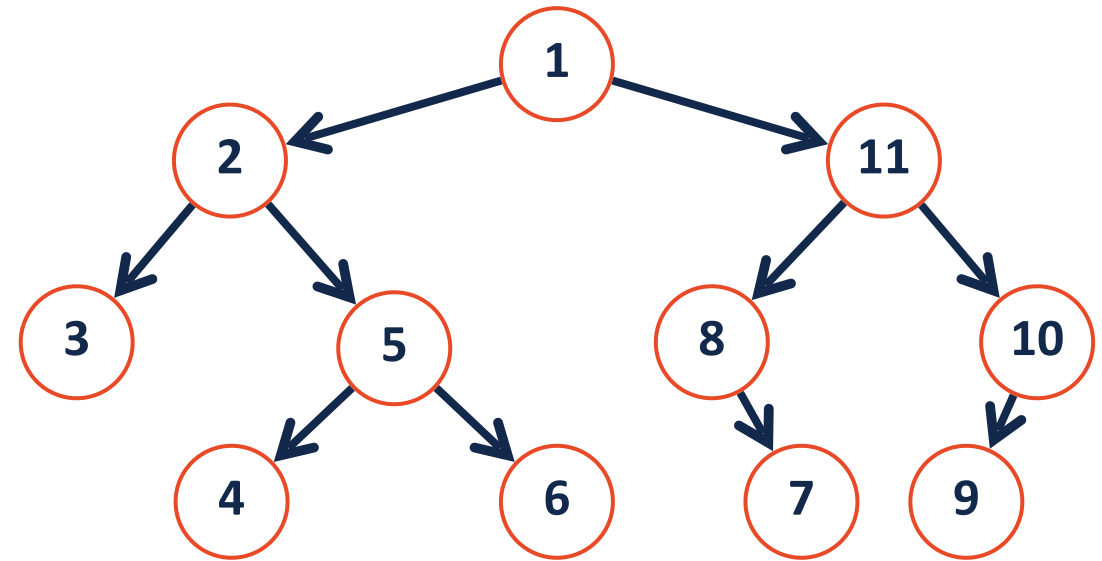
3 4 6 5 2 7 8 9 10 11 2

root  
left  
right

left  
root  
right

left  
right  
root

# Tree Traversals (Solution)



Updated 9/21: Postorder tip corrected

**Pre-order:** 1 2 3 5 4 6 11 8 7 10 9

Tip: Preorder always starts with root!

**In-order:** 3 2 4 5 6 1 8 7 11 9 10

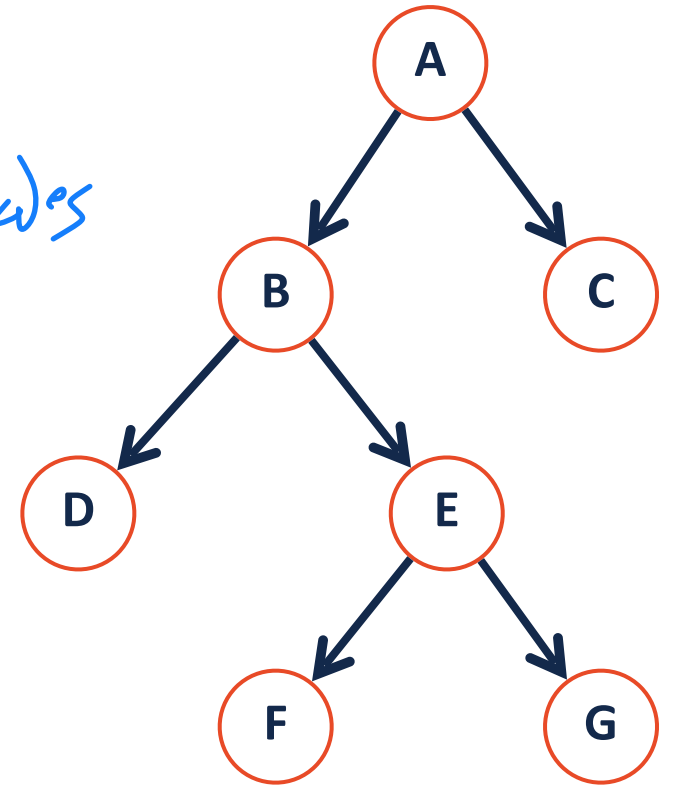
Tip: Inorder always starts with leftmost node. Root is after all left nodes!

**Post-order:** 3 4 6 5 2 7 8 9 10 11 1

Tip: Root is always LAST node! First node depends on tree structure

# Traversal vs Search

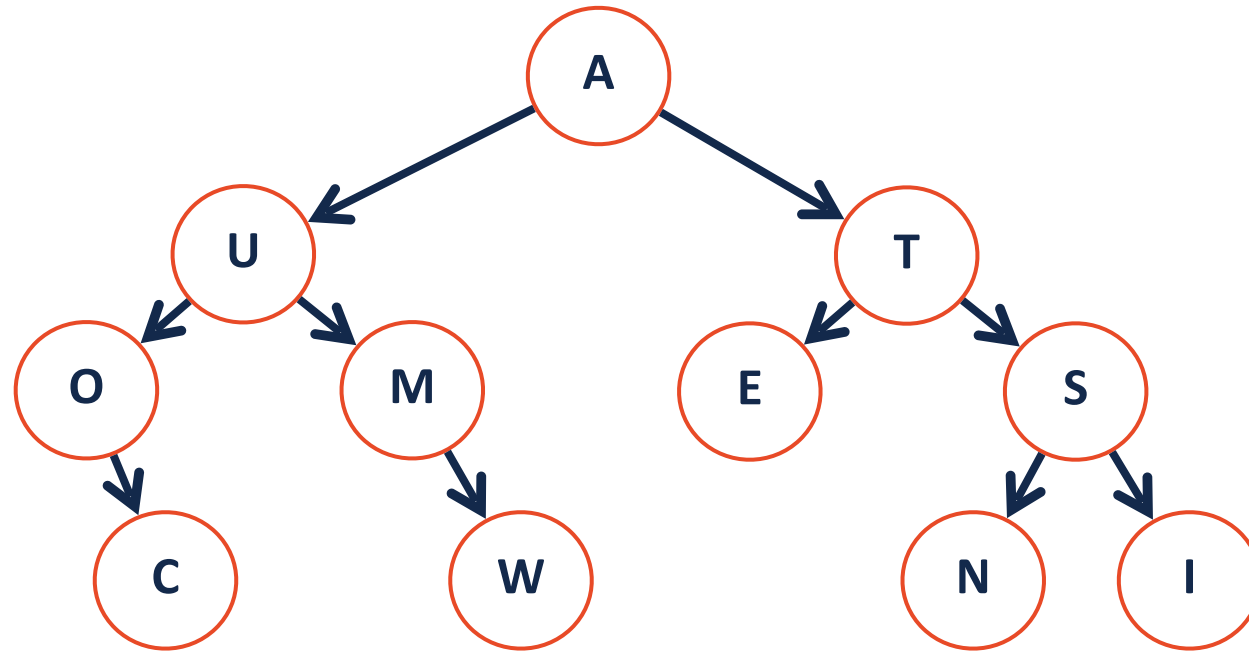
Traversal — Always looks at all nodes



Search — Might look at all nodes

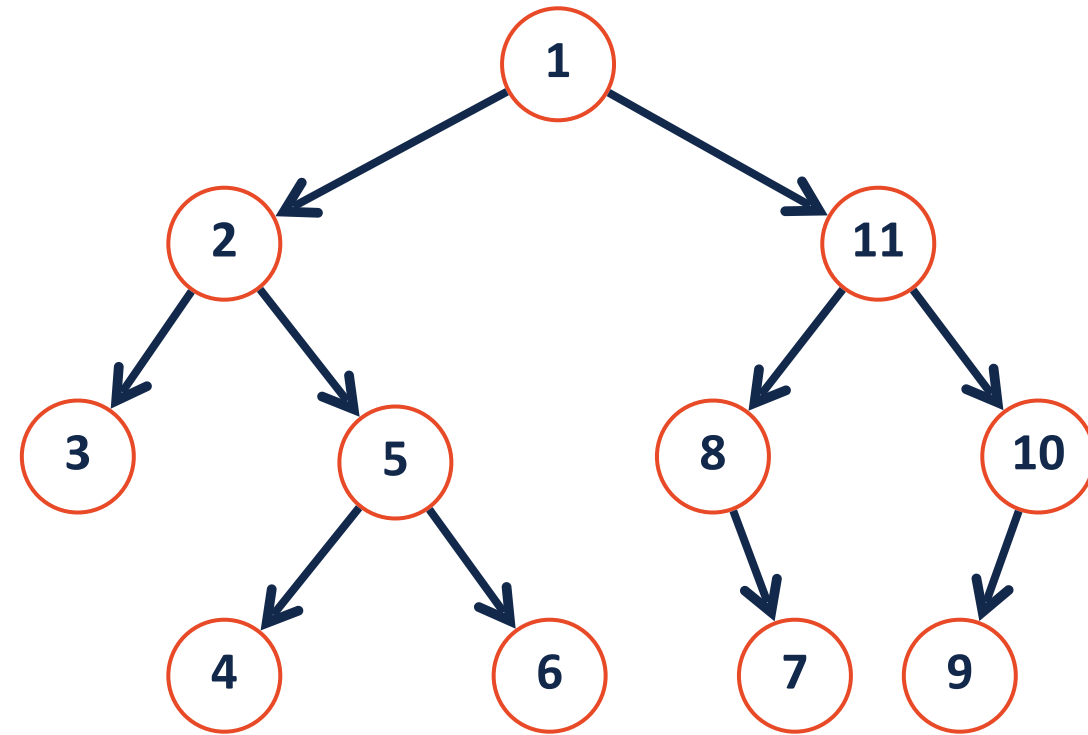
# Tree Search

There are two main approaches to searching a binary tree:



# Depth First Search

**Explore as far along one path as possible before backtracking**



# Depth First Search

**Explore as far along one path as possible before backtracking**

Make a stack initialized with root

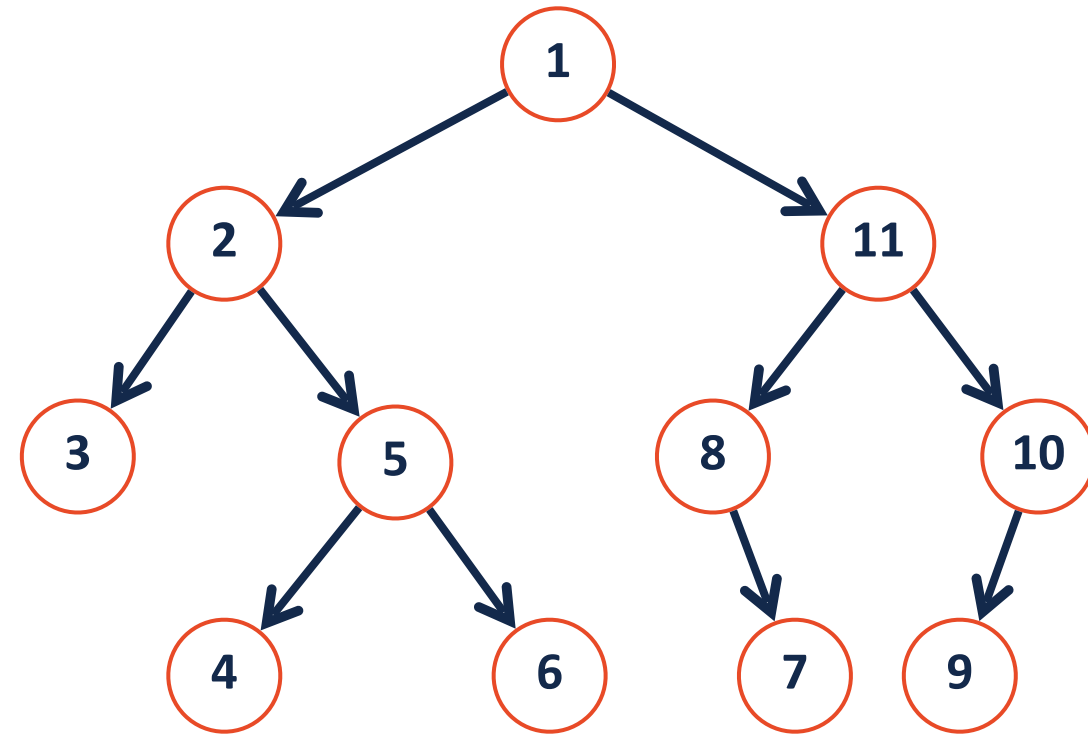
While stack isn't empty:

- Pop top element (as tmp)

- Check if tmp is query (visual: 'print')

- Push tmp->right to stack

- Push tmp->left to stack



Stack: 1, 11, 2, 5, 3, 6, 4, 10, 8, 7, 9

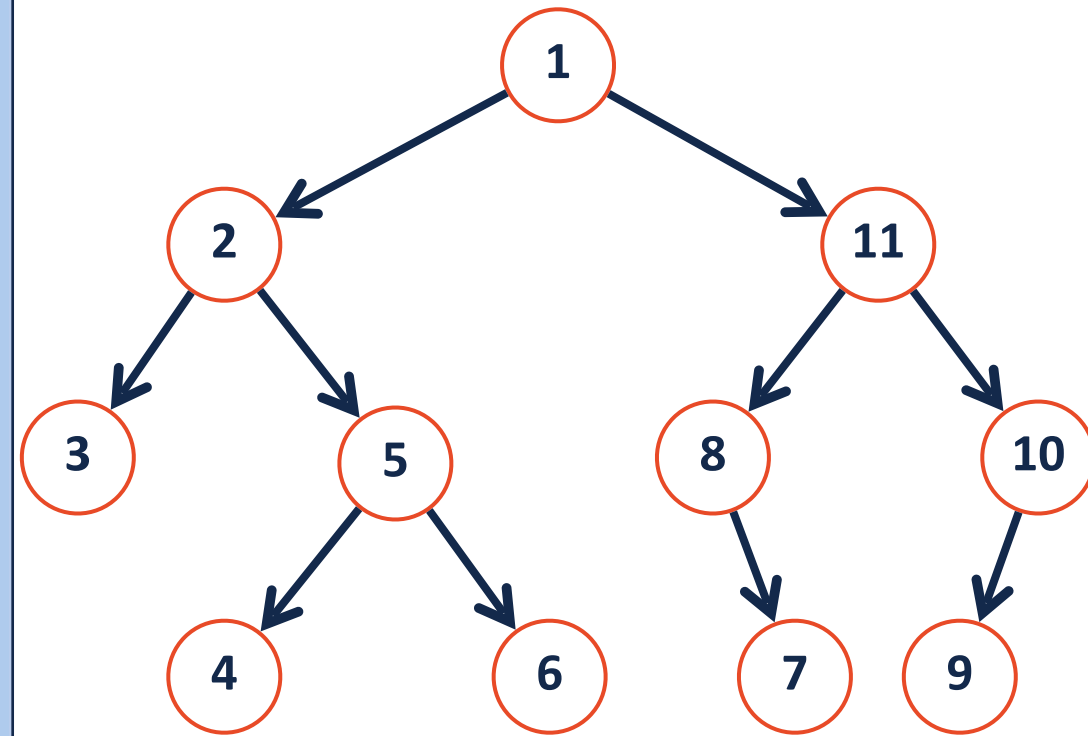
Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

# Depth First Search



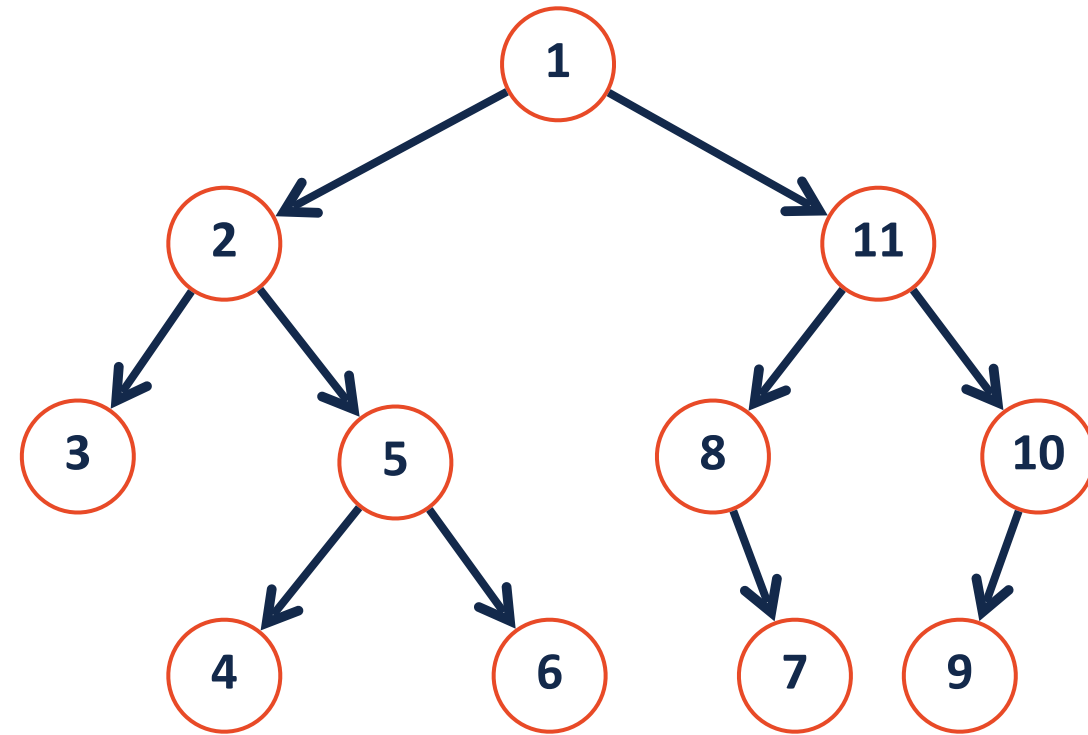
**Explore as far along one path as possible before backtracking**

```
1 template<class T>
2 bool BinaryTree<T>::DFS_search(TreeNode*
   root, T& query)
3 {
4     if (root) {
5         if (root->data == query) {
6             return true;
7         }
8         if (DFS_search(root->left, query)) {
9             return true;
10        }
11        if (DFS_search(root->right, query)) {
12            return true;
13        }
14    }
15    return false;
16 }
17
18
```



# Breadth First Search

**Fully explore level  $i$  before exploring level  $i+1$**



# Breadth First Search

**Fully explore level  $i$  before exploring level  $i+1$**

Make a queue initialized with root

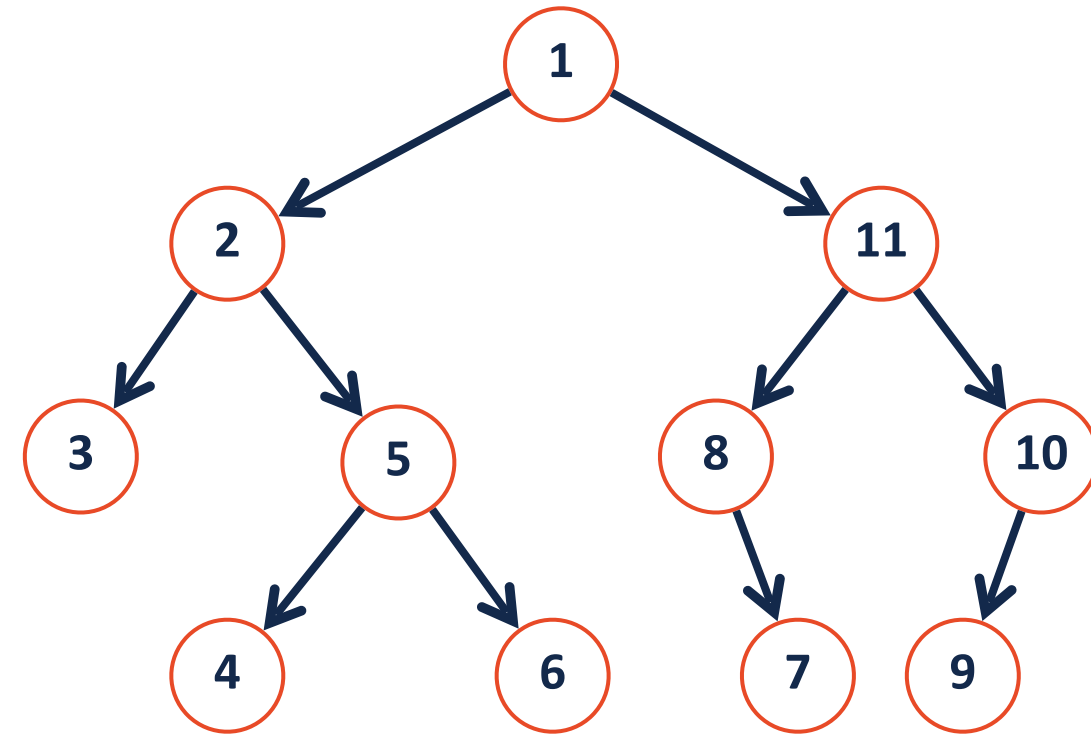
While queue isn't empty:

- Dequeue front element (as tmp)

- Check if tmp is query (visual: 'print')

- Enqueue tmp->left

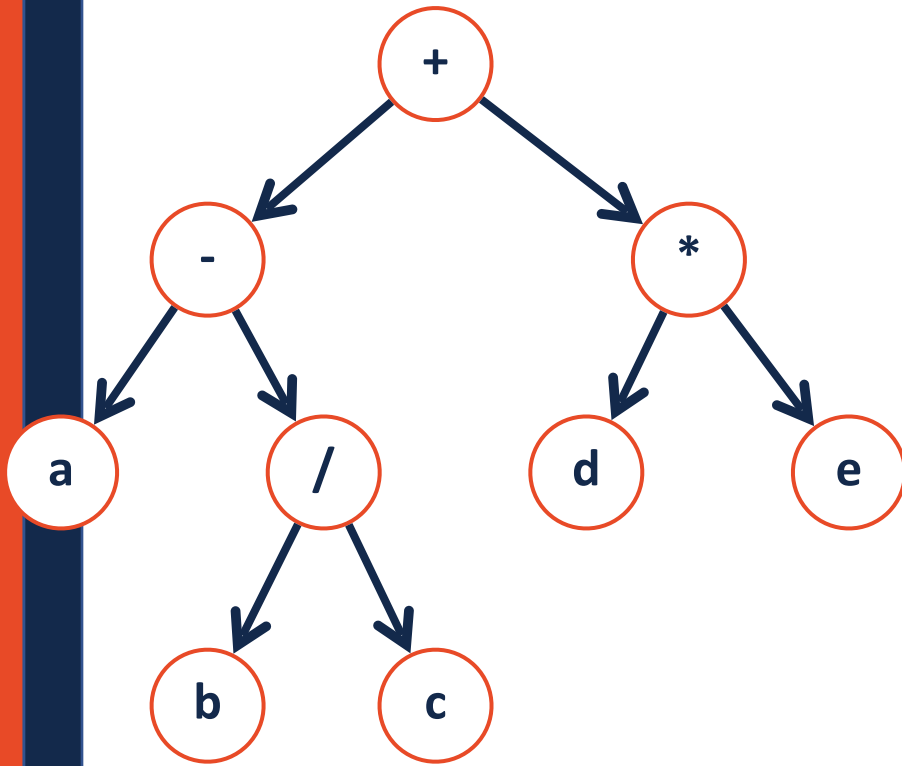
- Enqueue tmp->right



Queue: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

# Level-Order Traversal



```
1 template<class T>
2 bool BinaryTree<T>::BFS_search(TreeNode* root, T&
  query)
3 {
4     if (!root) return false;
5
6     std::queue<TreeNode*> q;
7     q.push(root); // enqueue == push
8
9     while (!q.empty()) {
10         TreeNode* temp = q.front();
11         q.dequeue();
12
13         if (temp->data == query) return true;
14
15         if (temp->left) q.enqueue(temp->left);
16         if (temp->right) q.enqueue(temp->right);
17     }
18
19     return false;
20 }
```

# Tree Search Tradeoffs

How can we improve our ability to search a binary tree?

What do we trade in order to do so?