Data Structures

Tree Traversal

CS 225 September 17, 2025

Brad Solomon & Harsha Tirumala

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Extra Credit Reminder

MP submission on PL has two separate submissions

The extra credit portion will only test part 1

Completion of the extra credit portion by the following Monday is worth 4 points

MP_stickers feedback form out now!

Anonymous feedback form worth 2 points **collectively**

↳ 70% or more ↗

Learning Objectives

Review and expand on foundational tree terminology

Discuss the tree ADT

Explore tree implementation details

Binary Tree

→ Acyclic, Connected + ≤ 2 children

Lets define additional terminology for different **types** of binary trees!

1. Full Tree

2. Perfect Tree

3. Complete Tree

Binary Tree: full

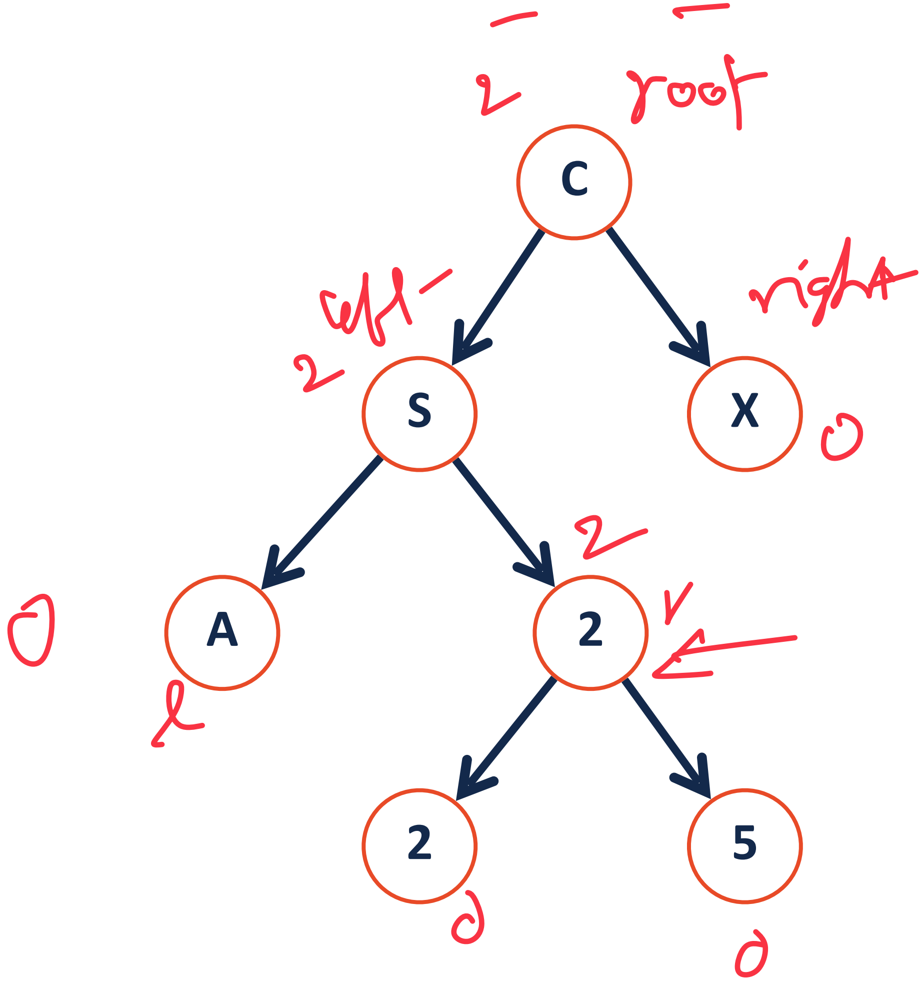
A **full tree** is a binary tree where every node has either 0 or 2 children

A tree **F** is **full** if and only if:

1.

2.

3.



Binary Tree: full

A **full tree** is a binary tree where every node has either 0 or 2 children

A tree **F** is **full** if and only if:

1. $F = \emptyset$

✓

0 or 2

C

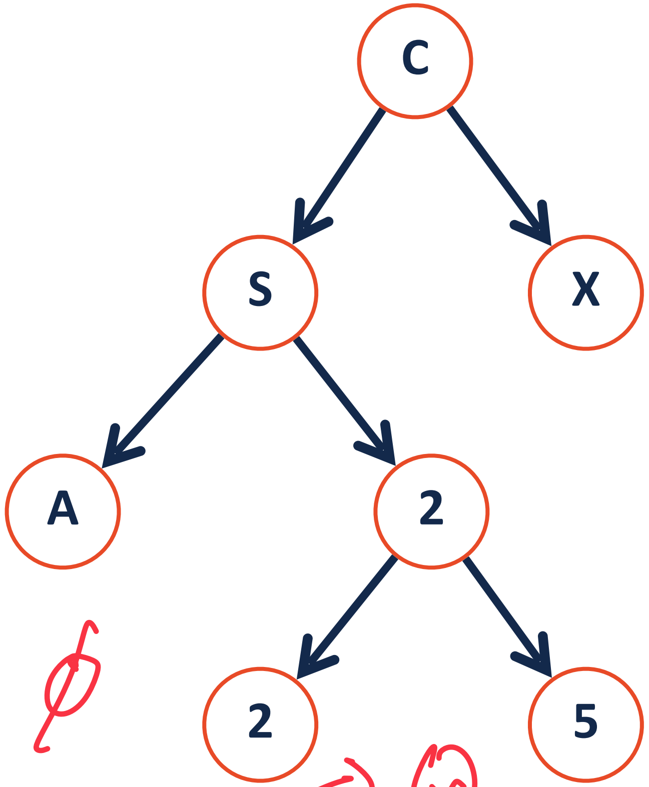
2. $F = (data, \emptyset, \emptyset)$

3. $F = (\overset{root}{data}, F_l \neq \emptyset, F_r \neq \emptyset)$

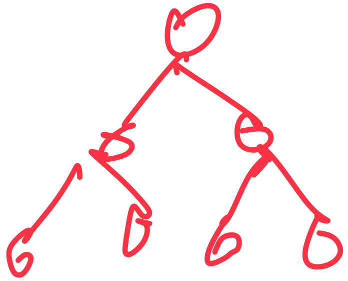
fun

$F = (data, F_L, F_R)$

$\emptyset$



Full binary tree : Size → Odd



- Question - Which of the following are possible sizes (# of nodes) for a full binary tree?

a) 2 b) 5 c) 7 d) 8

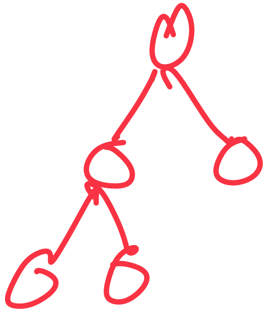
A) 2 and 8 4%

[✓ B) 5 and 7 78%]

C) 7 only 13%

D) All of these 6%

E) None of these



Join Code : 225

Binary Tree: perfect

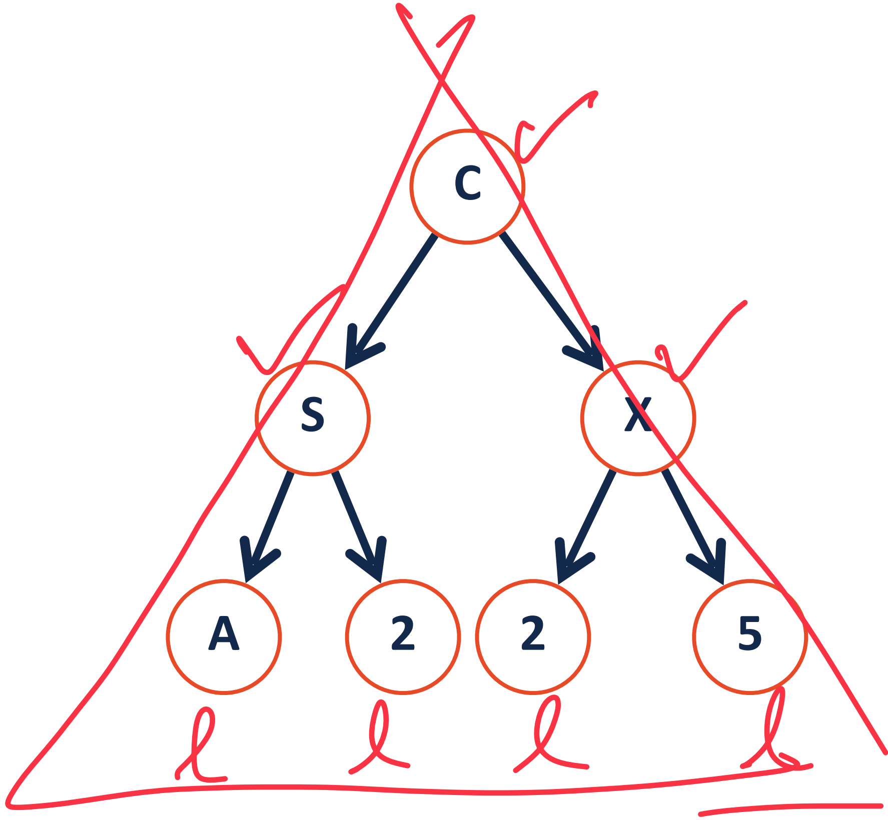
A **perfect tree** is a binary tree where…

Every internal node has 2 children and all leaves are at the same level.

A tree **P** is **perfect** if and only if:

1.

2.



Binary Tree: perfect

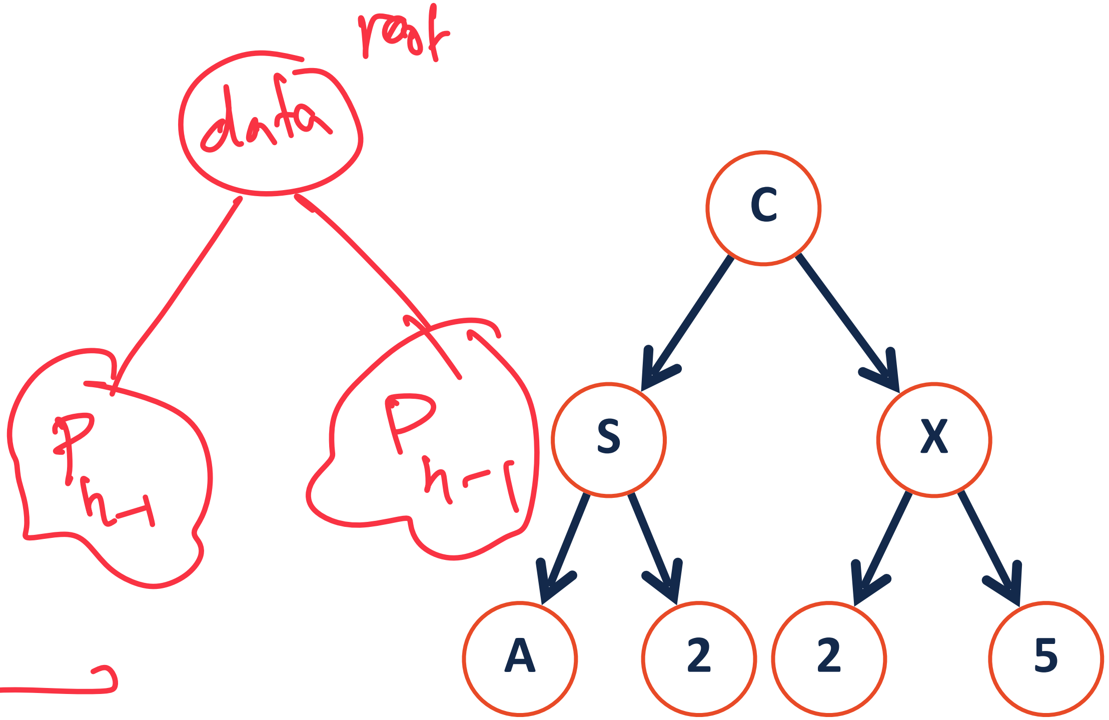
A **perfect tree** is a binary tree where…

Every internal node has 2 children and all leaves are at the same level.

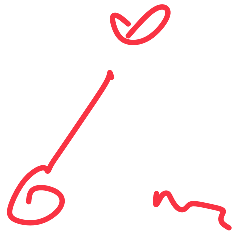
A tree **P** is **perfect** if and only if:

1. $\underline{P_h} = (data, P_{h-1}, P_{h-1})$

2. $\underline{P_0 = (data, \emptyset, \emptyset)} \equiv \underbrace{P_{-1} = \emptyset}_{\text{date}}$



Perfect binary tree : Size → $2^K - 1$



- Question - Which of the following are possible sizes (# of nodes) for a perfect binary tree?

a) 2 b) 5 c) 7 d) 8

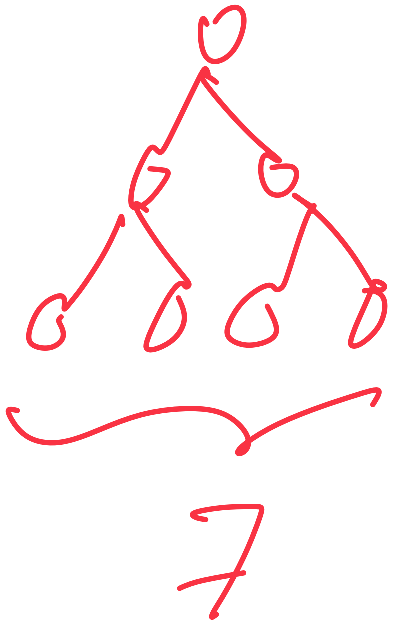
A) 2 and 8 3%

B) 5 and 7 7%

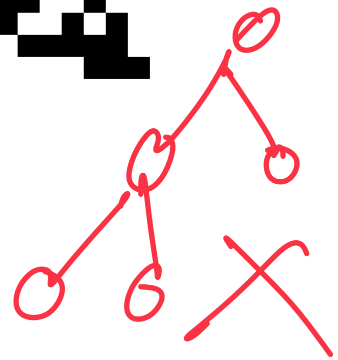
[✓ C) 7 only 86%]

D) All of these

E) None of these 3%



Join code : 225



Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

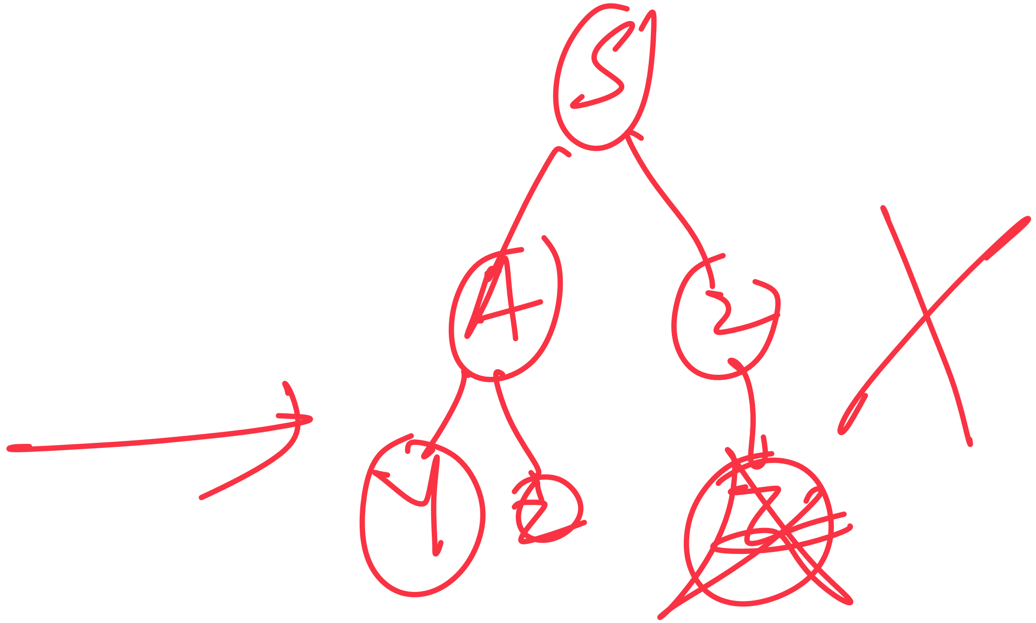
The last level contains at least one node (and is pushed to left)

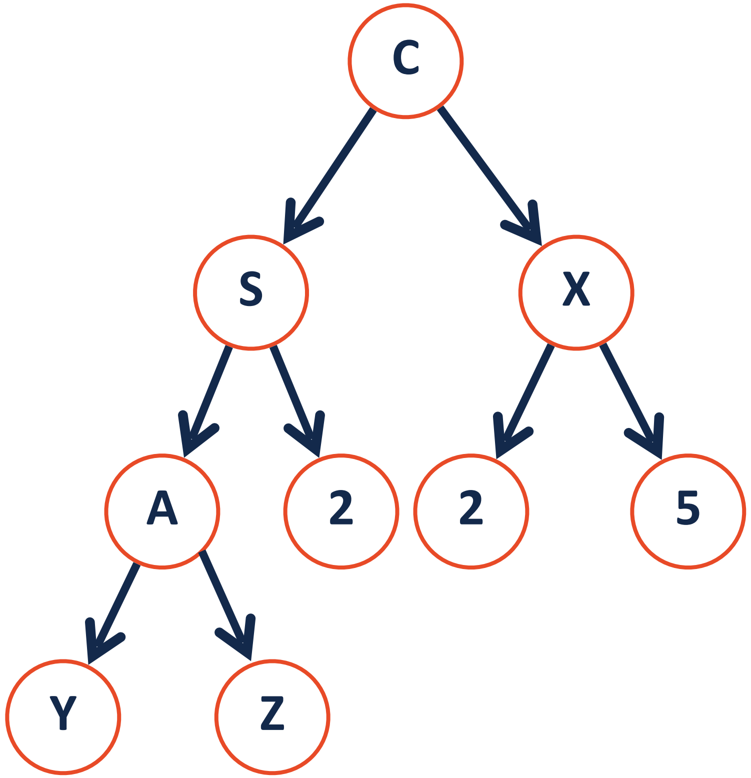
A tree **C** is **complete** if and only if:

1.

2.

3.





Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

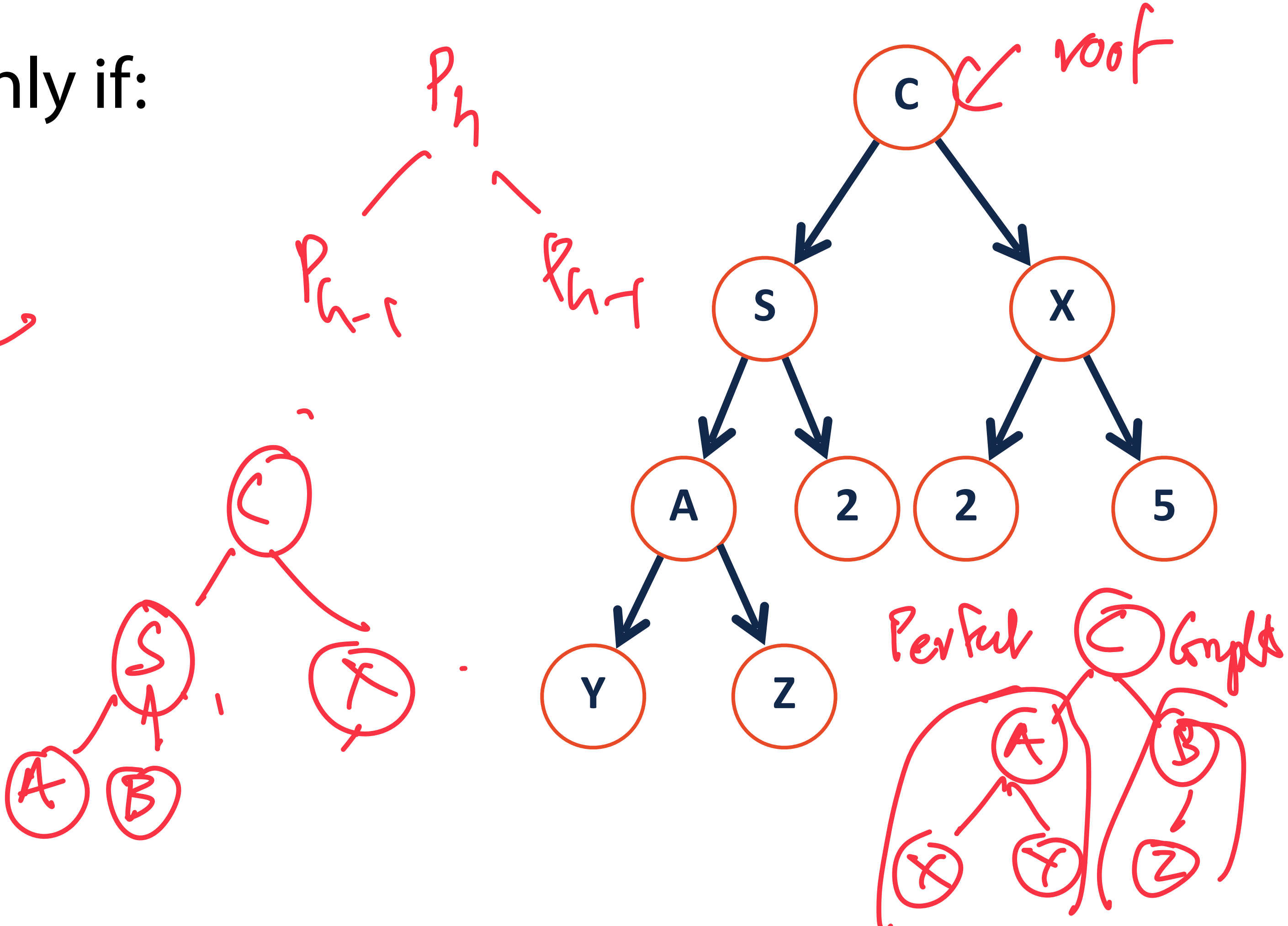
The last level contains at least one node (and is pushed to left)

A tree **C** is **complete** if and only if:

1. $C_h = (data, C_{h-1}, P_{h-2})$

2. $C_h = (data, P_{h-1}, C_{h-1})$

3. $C_{-1} = \emptyset$



Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

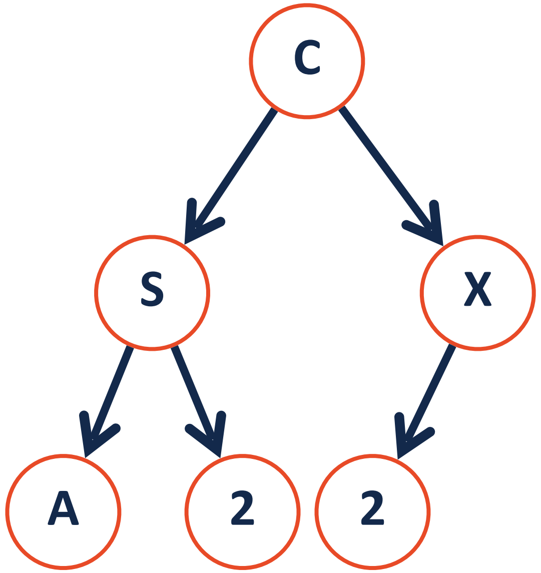
The last level contains at least one node (and is pushed to left)

A tree **C** is **complete** if and only if:

1. $C_h = (data, C_{h-1}, P_{h-2})$

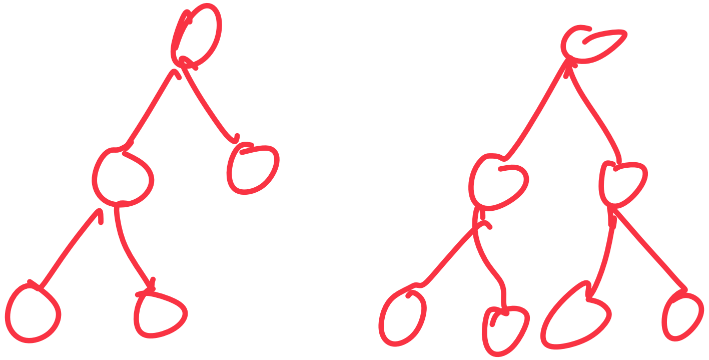
2. $C_h = (data, P_{h-1}, C_{h-1})$

3. $C_{-1} = \emptyset$



Left - right

Complete binary tree : Size



- Question - Which of the following are possible sizes (# of nodes) for a complete binary tree?

a) 2 b) 5 c) 7 d) 8

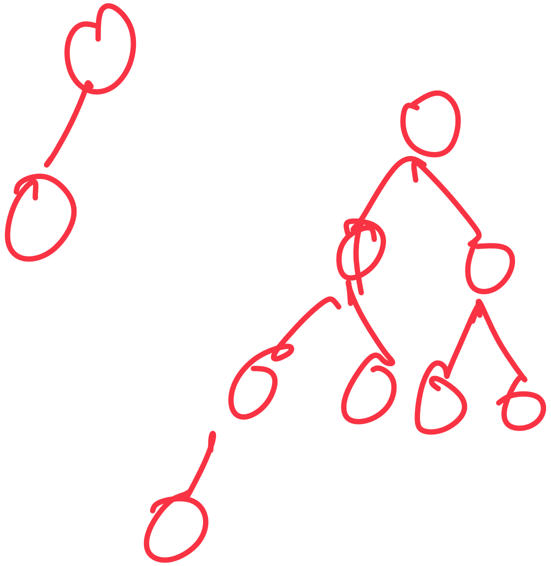
A) 2 and 8 30%

B) 5 and 7 15%

C) 7 only 4% ✓

[✓ D) All of these] 52%

E) None of these



Join code : 225

Binary Tree

Why do we care?

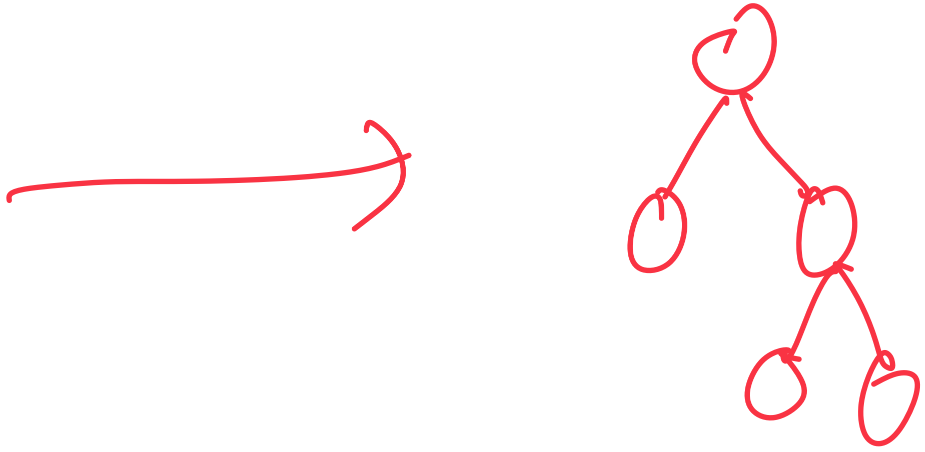
1. Terminology instantly defines a particular tree structure

2. Understanding how to think ‘recursively’ is very important.

Binary Tree: Thinking with Types

Is every **full** tree **complete**?

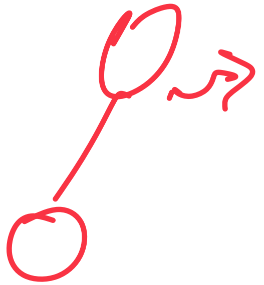
✗



Is every **complete** tree **full**?

✗

✓



=1child

2

Tree ADT

Insert – insert new nodes

Remove – remove nodes

Traverse – options

Find – Search (Structure)

Constructor –

List.h

```
#pragma once

template <typename T>
class List {
  public:
    /* ... */
  private:
    class ListNode {
      T & data;

      ListNode * next;



      ListNode(T & data) :
       data(data), next(NULL) { }
    };



    ListNode *head_;
    /* ... */
};
```

Tree.h

```
#pragma once

template <typename T>
class BinaryTree {
  public:
    /* ... */
  private:











    };

    TreeNode *root_;
    /* ... */
};
```

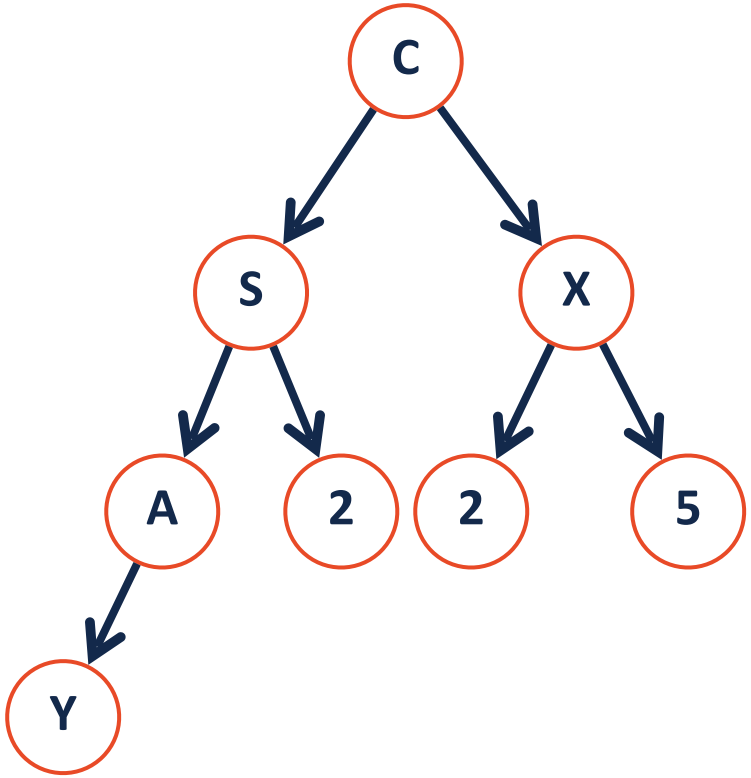
List.h

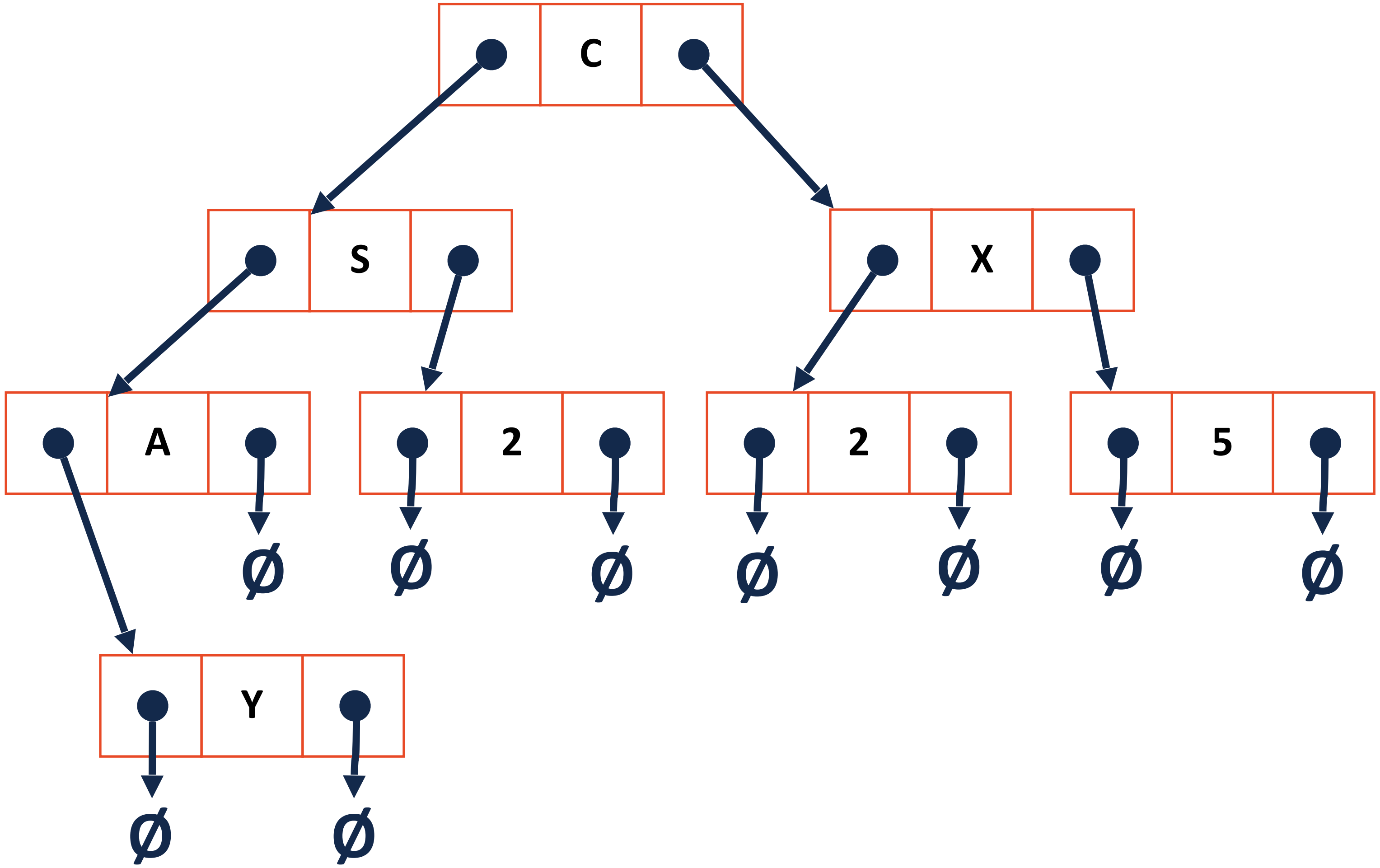
```
#pragma once

template <typename T>
class List {
  public:
    /* ... */
  private:
    class ListNode {
      T & data;

      ListNode * next;



      ListNode(T & data) :
       data(data), next(NULL) { }
    };



    ListNode *head_;
    /* ... */
};
```

Tree.h

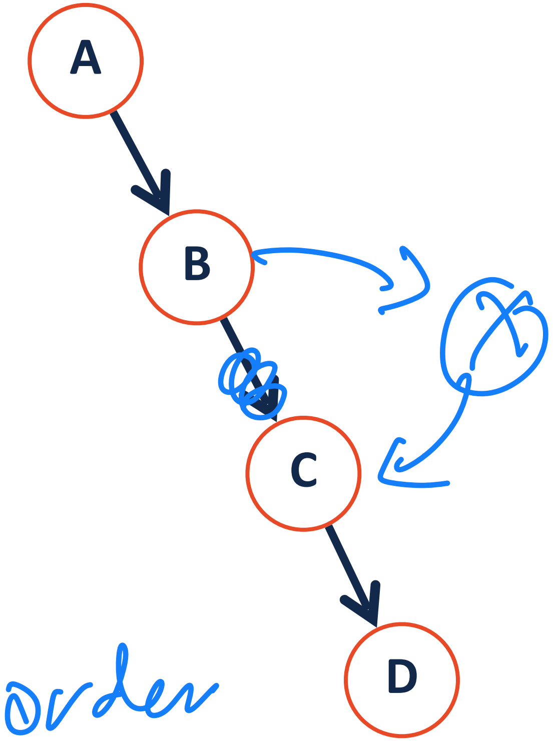
```
#pragma once

template <typename T>
class BinaryTree {
  public:
    /* ... */
  private:
    class TreeNode {
      T & data;

      TreeNode * left;

      TreeNode * right;

      TreeNode(T & data) :
       data(data), left(NULL),
right(NULL) { }

    };

    TreeNode *root_;
    /* ... */
};
```

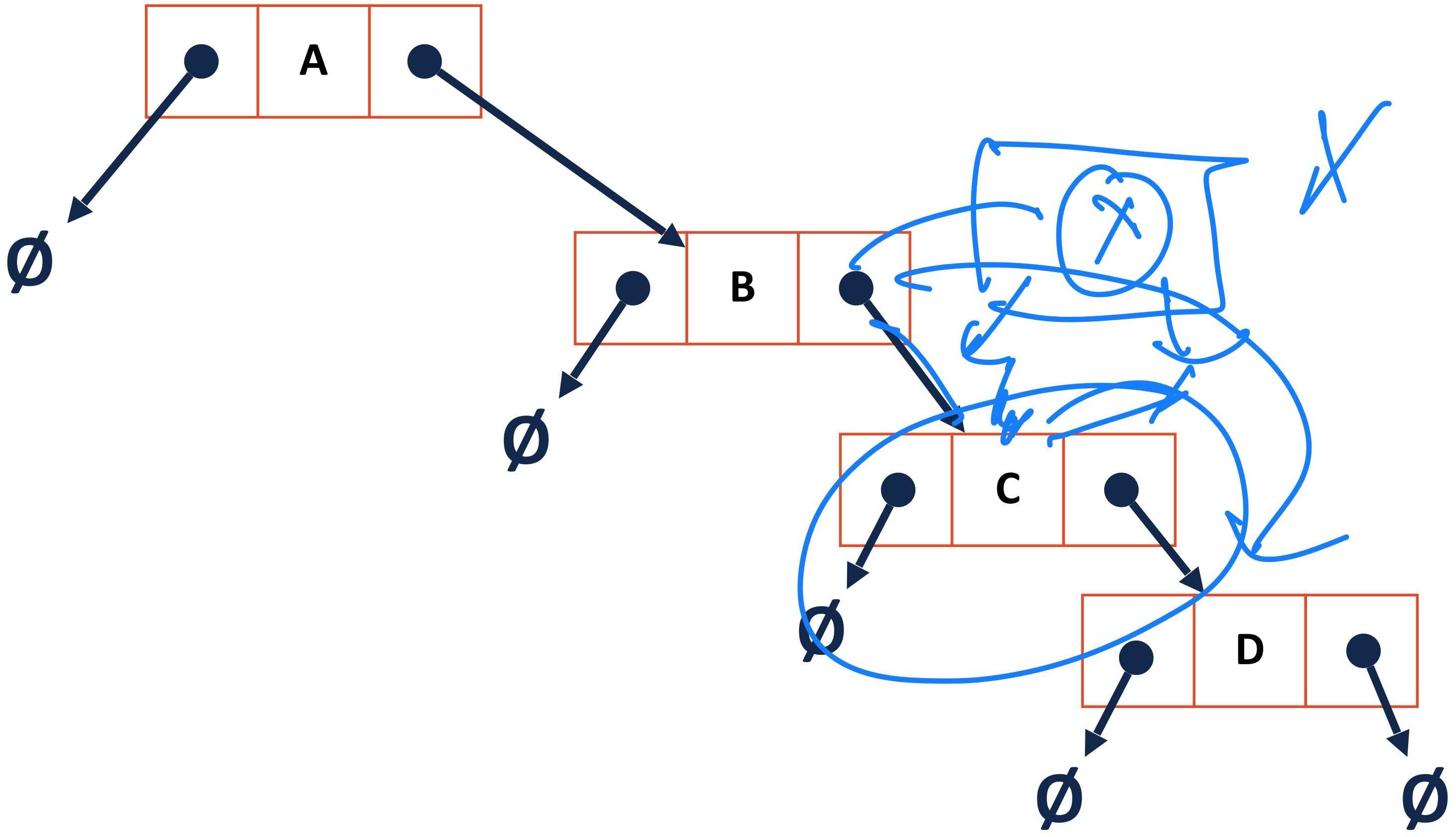
Visualizing trees





Tree Insert / Remove acts like Linked List



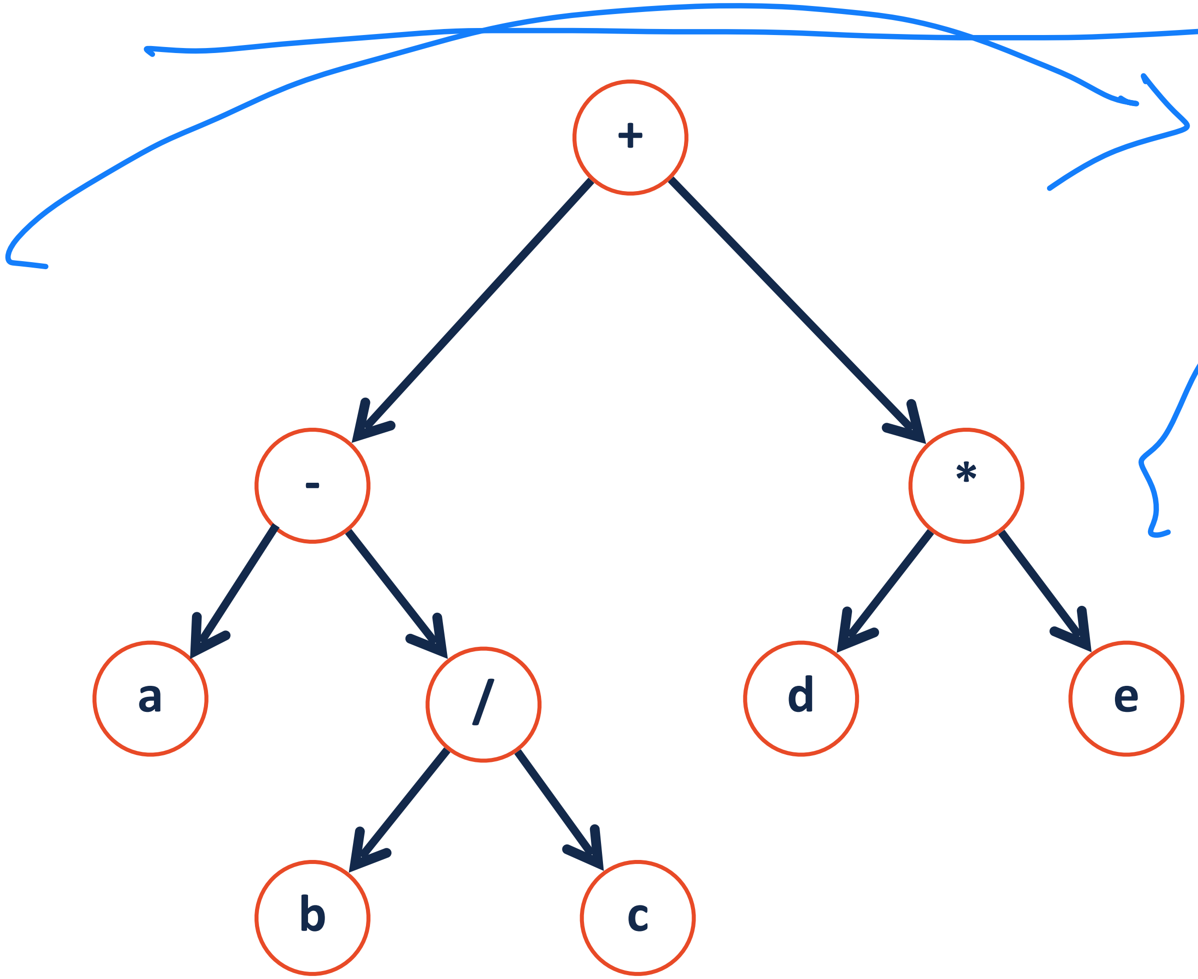


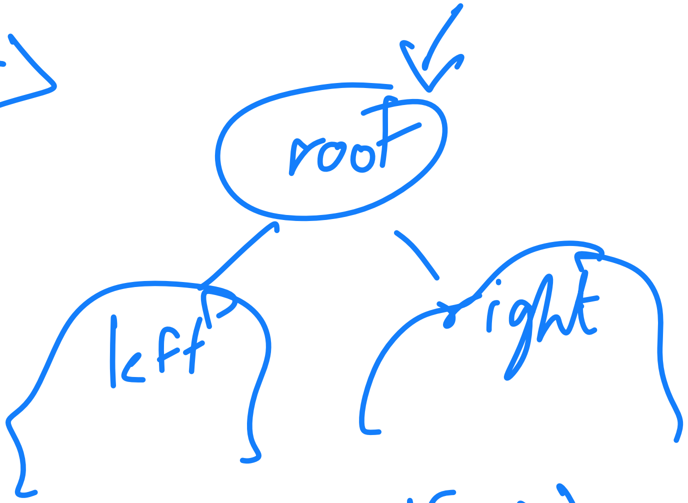
order
of
opn

Tree Traversal

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

Print



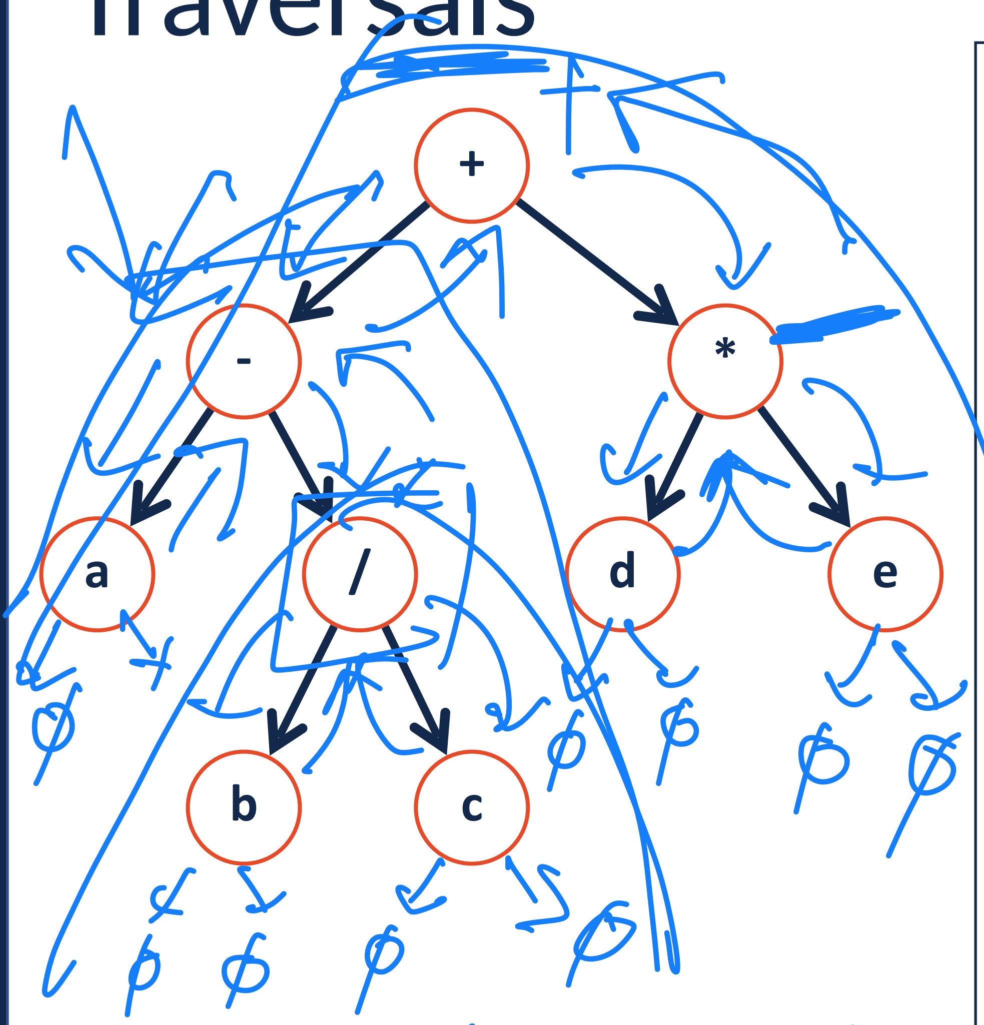


print(root)

left

right

Traversals

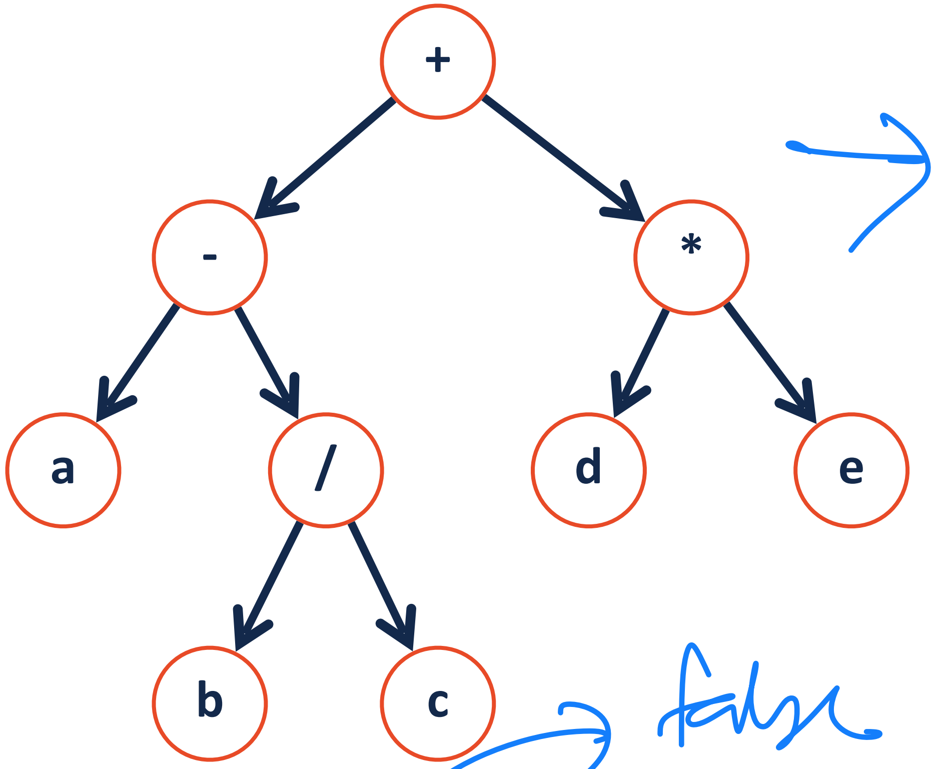


+ - a / b c * d e

```
template<class T>
void BinaryTree<T>::______Order(TreeNode * root)
{


















}
```

pre-order traversal

Traversals



false

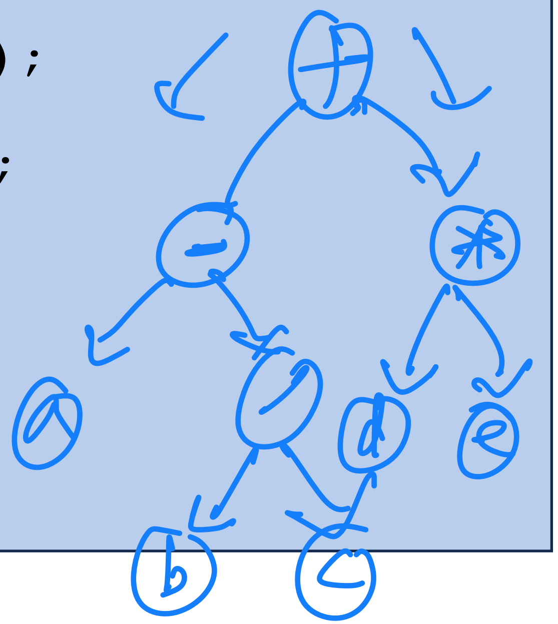
if (ptr)

ptr = nullptr

```
template<class T>
void BinaryTree<T>::preOrder(TreeNode * root)
{

  if (root) {

      print(root)________;

      preOrder(root->left);

      ____________________;

      preOrder(root->right);

      ____________________;

  }




}
```

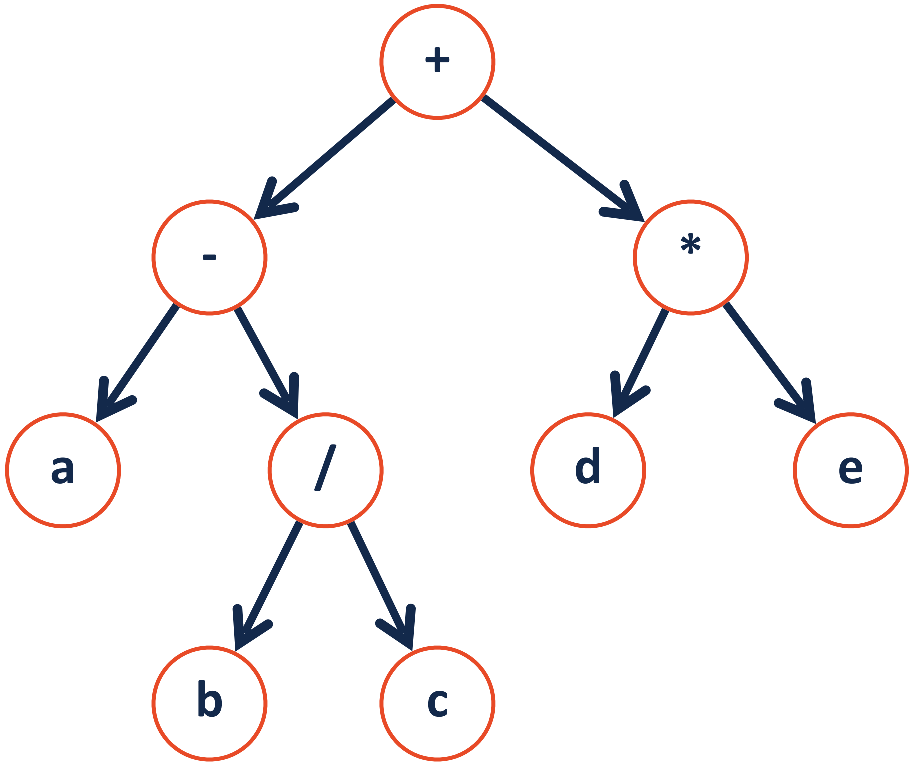
T = ϕ (false)

pre order traversal
root



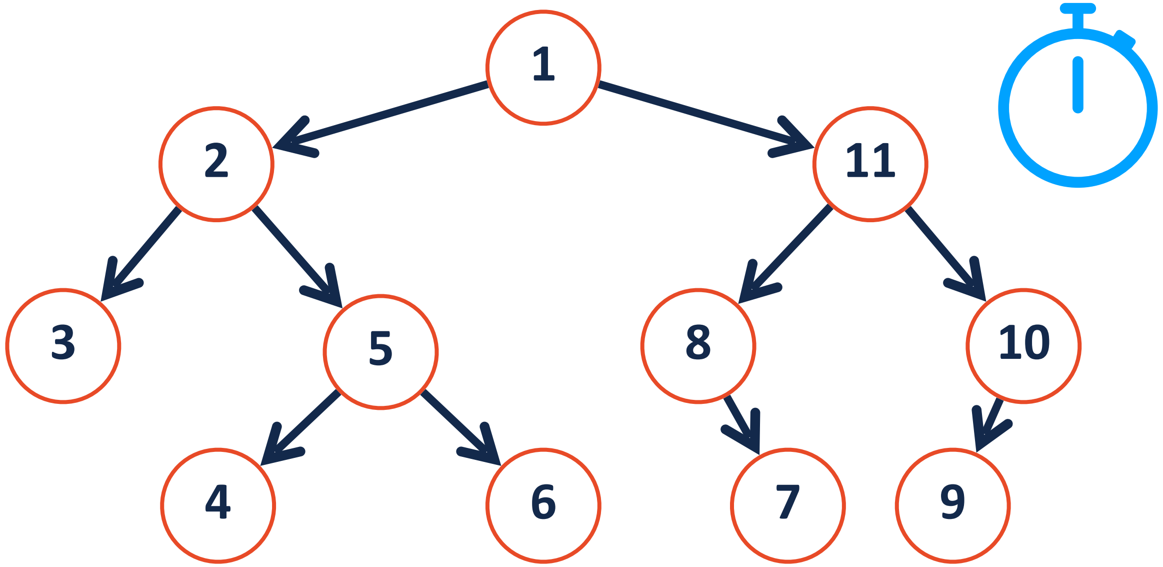
Copy a tree

Traversals



```
template<class T>
void BinaryTree<T>::_____Order(TreeNode * root)
{

  if (root) {

      ______________________;

      _____Order(root->left);

      ______________________;

      _____Order(root->right);

      ______________________;

  }



}
```

Tree Traversals

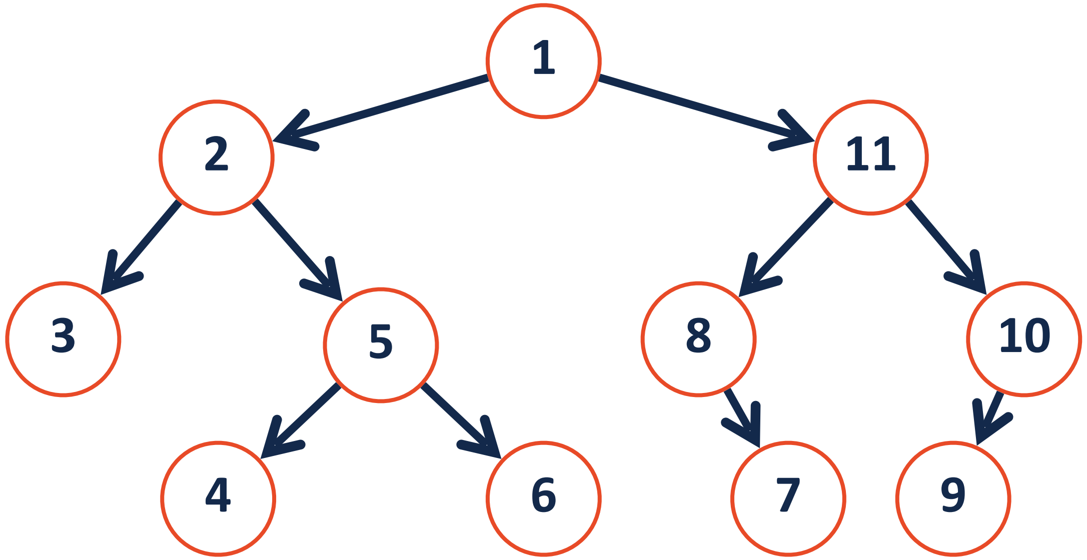


Pre-order: ✓ root left right

In-order: left root right

Post-order: left right root

Tree Traversals (Solution)



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!

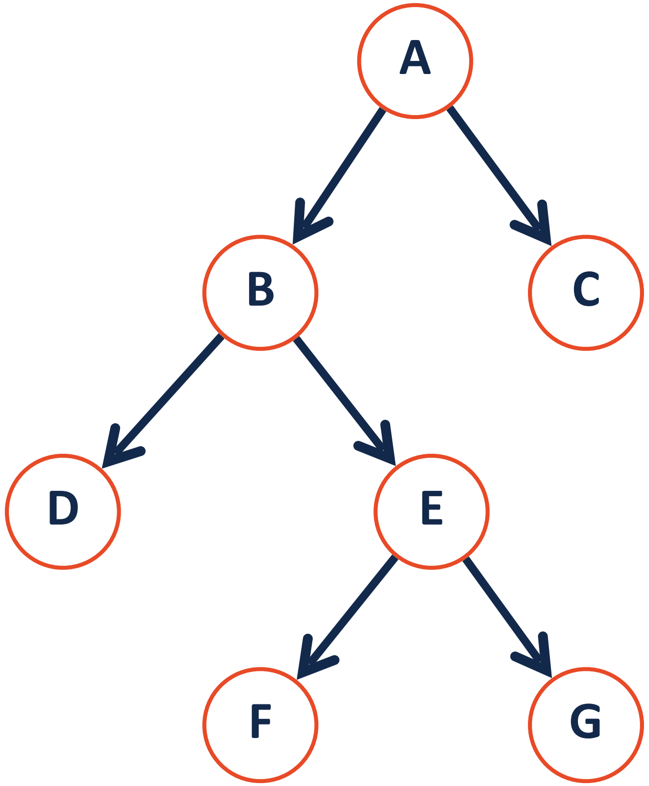
Post-order: **3** 4 6 5 2 7 8 9 10 11 **1**

Tip: Post always starts with leftmost node. Root is always LAST node!

Traversal vs Search

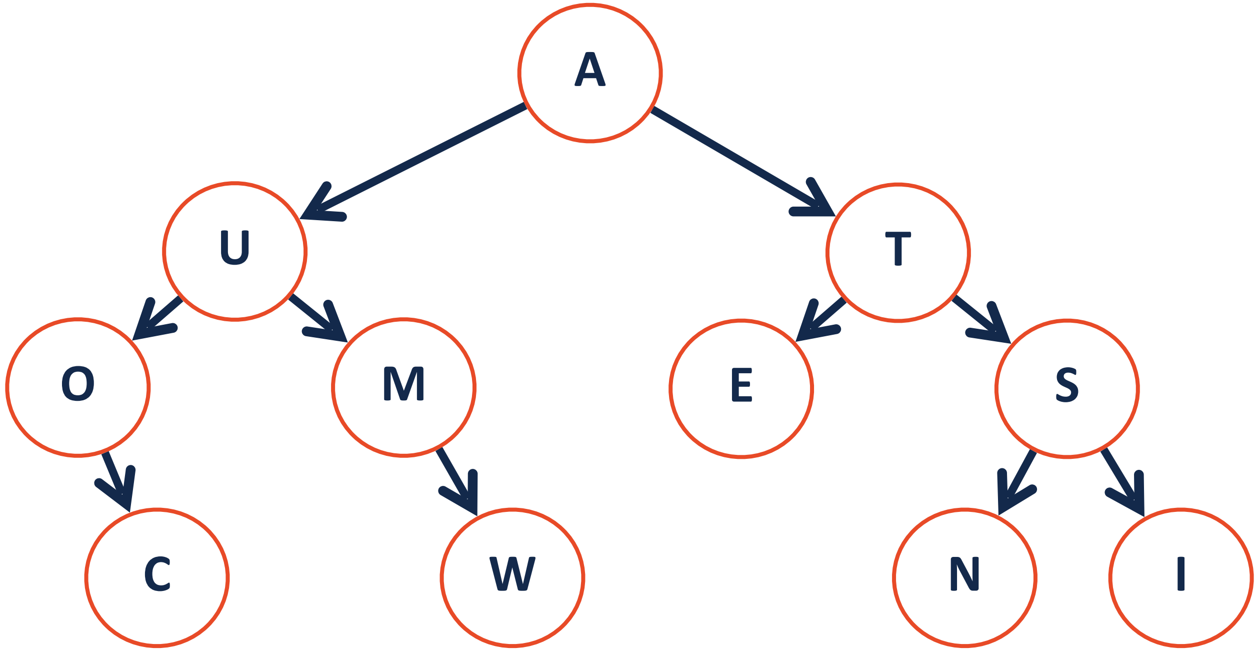
Traversal

Search



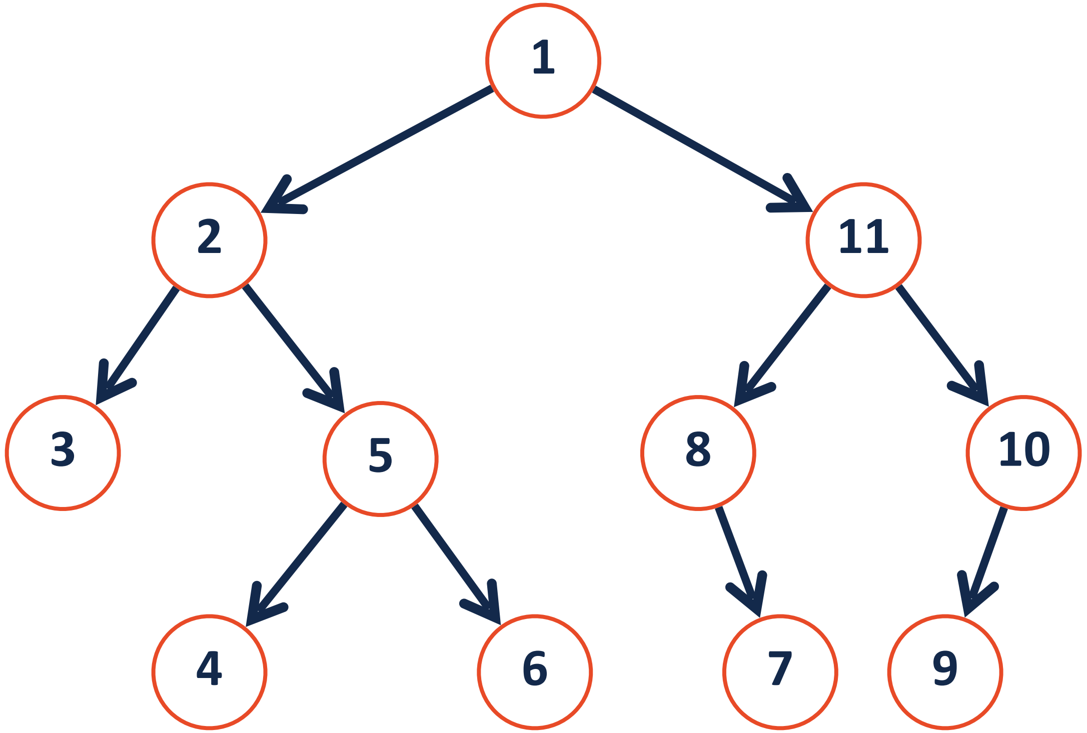
Tree Search

There are two main approaches to searching a binary tree:



Depth First Search

Explore as far along one path as possible before backtracking

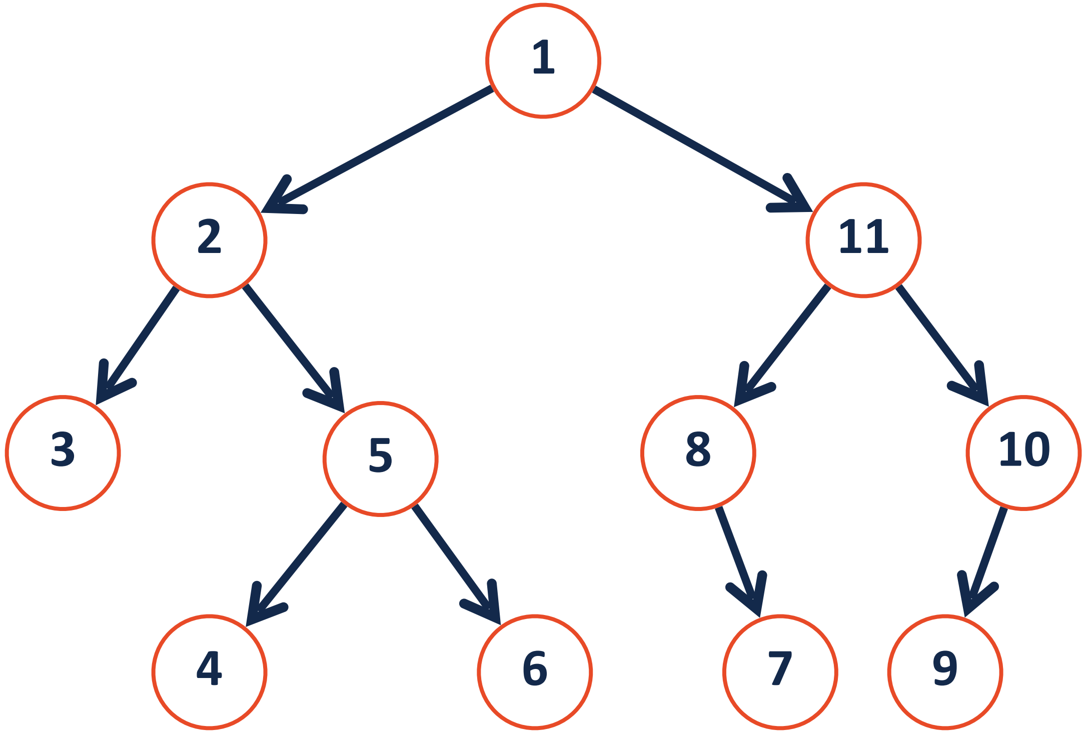


Depth First Search

Explore as far along one path as possible before backtracking

Make a stack (initialized with root)

```
While stack not empty:
    tmp = stack.pop()
    print(tmp)
    stack.push(tmp->right)
    stack.push(tmp->left)
```

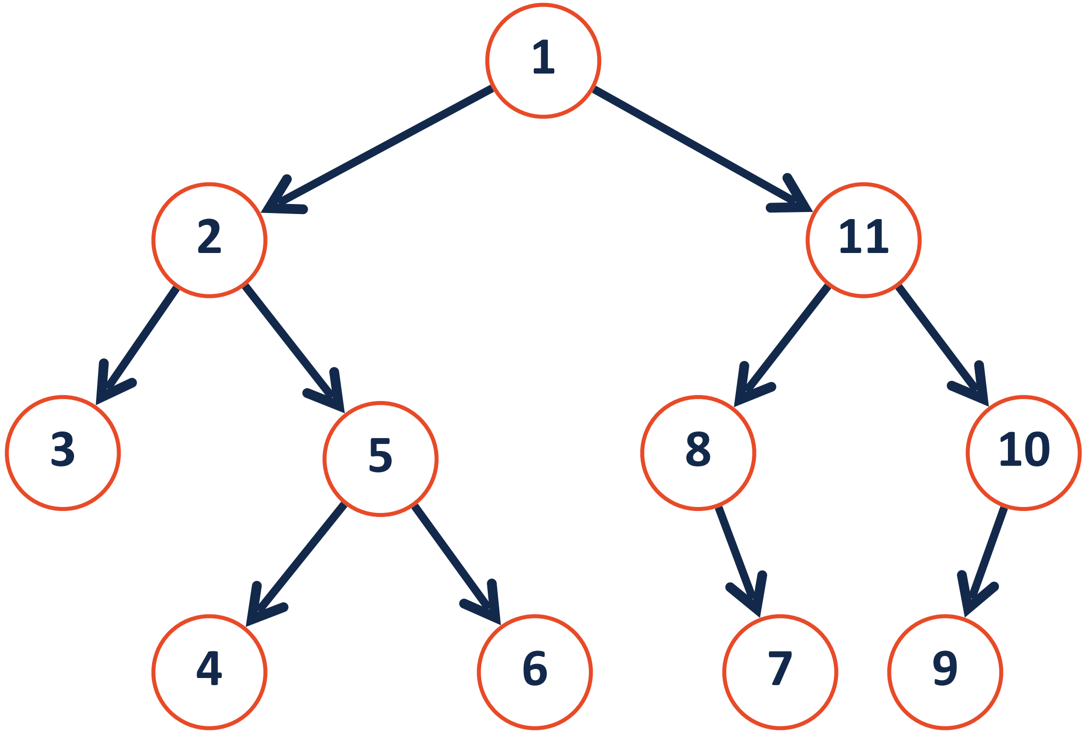


Stack:

Print:

Breadth First Search

Fully explore depth i before exploring depth i+1

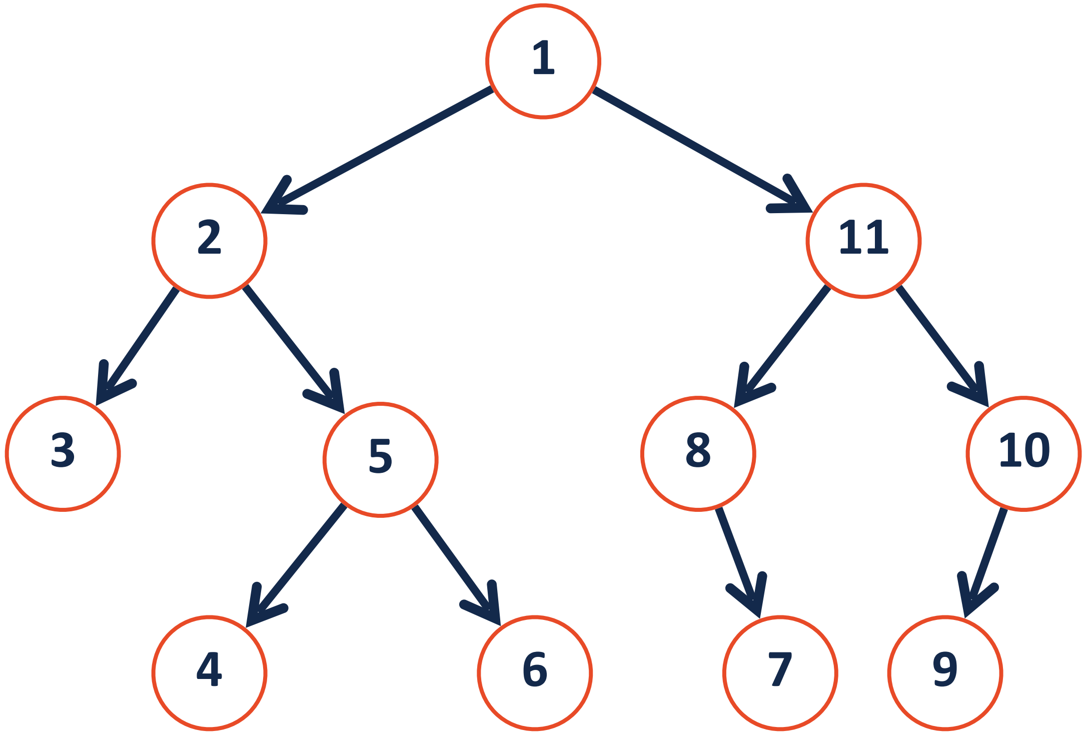


Breadth First Search

Fully explore depth i before exploring depth i+1

Make a queue (initialized with root)

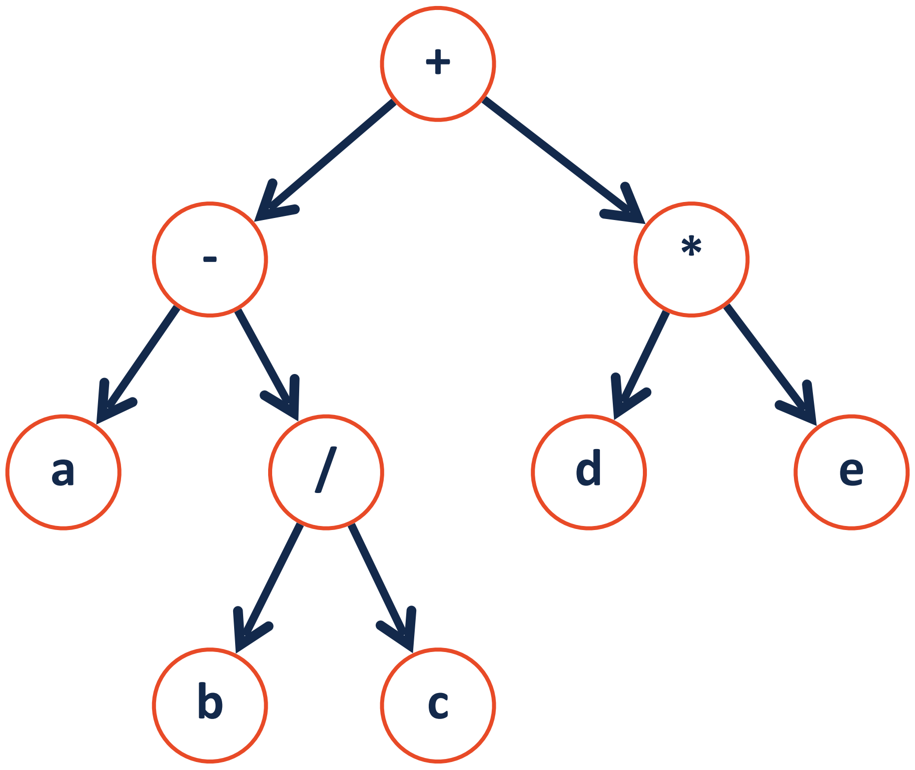
```
While queue not empty:
    tmp = queue.dequeue()
    print(tmp)
    queue.enqueue(tmp->left)
    queue.enqueue(tmp->right)
```



Queue:

Print:

Level-Order Traversal



```
template<class T>
void BinaryTree<T>::lOrder(TreeNode * root)
{

    Queue<TreeNode*> q;
    q.enqueue(root);

    while( q.empty() == False){

        TreeNode* temp = q.head();
        process(temp);

        q.dequeue();

        q.enqueue(temp->left);
        q.enqueue(temp->right);

    }
}
```

Tree Search

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

What do we trade in order to do so?