Data Structures

Trees

CS 225

Brad Solomon & G Carl Evans

September 13, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Review trees and binary trees

Practice tree theory with recursive definitions and proofs

Discuss the tree ADT

Explore tree implementation details

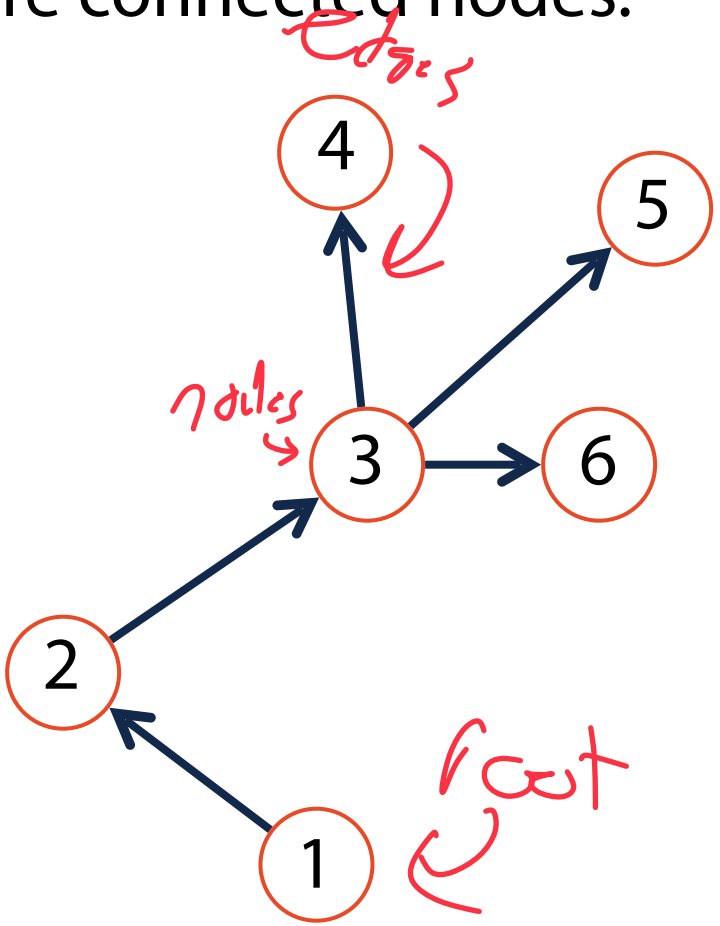
Trees

A non-linear data structure defined recursively as a collection of nodes where each node contains a value and zero or more connected nodes.

[In CS 225] a tree is also:

1) Acyclic — No path from node to itself

2) Rooted — A specific node is labeled root



Binary Tree

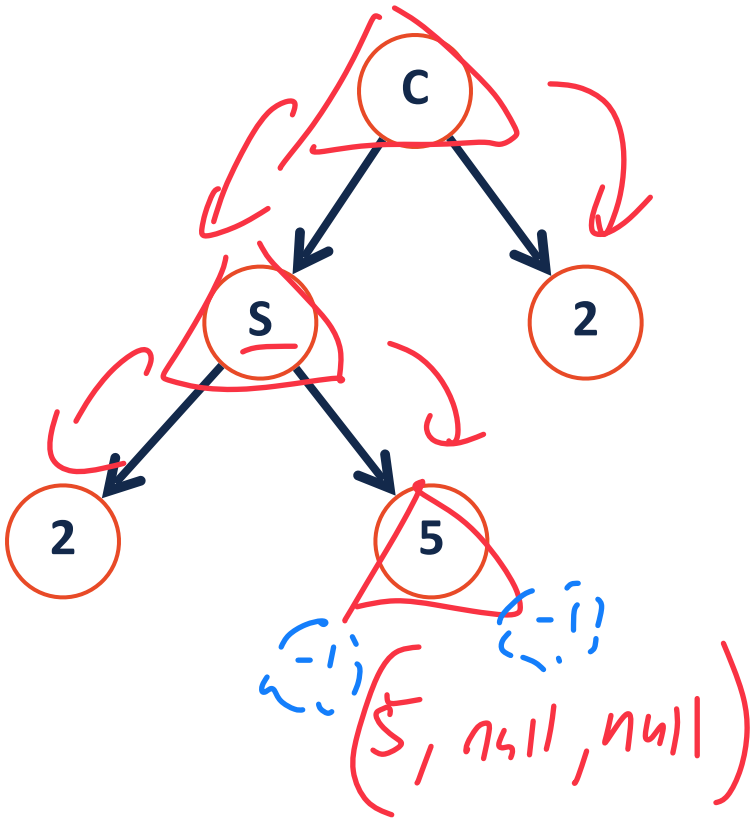
A **binary tree** is a tree T such that:

1. $T = \emptyset$

2. $T = (data, T_L, T_R)$

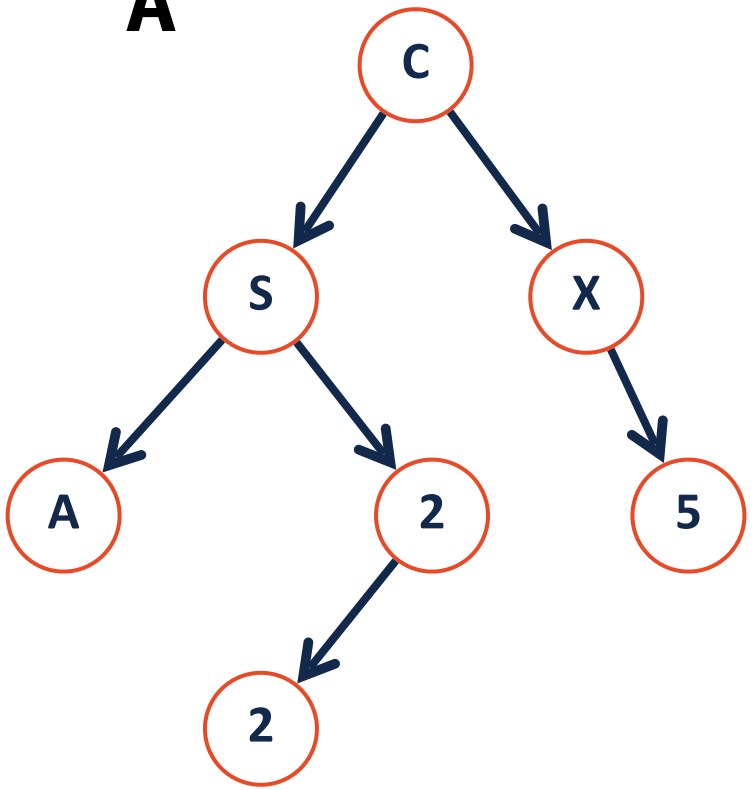
left child

right child



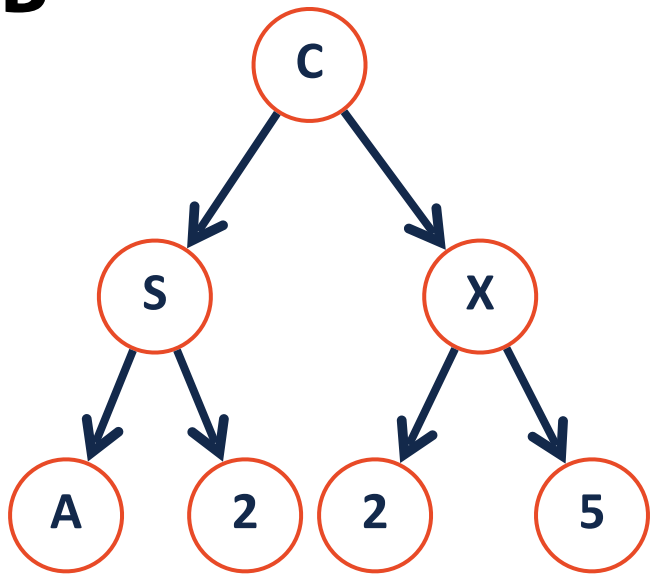
Which of the following are binary trees?

A



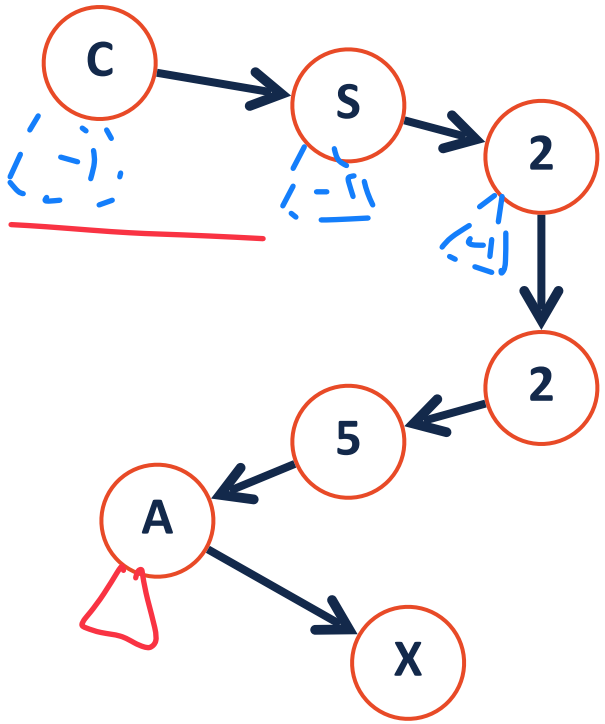
Most

B



Everyone

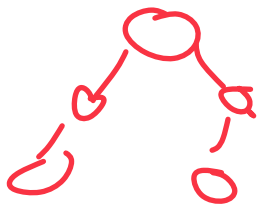
C



???

Next

Binary Tree

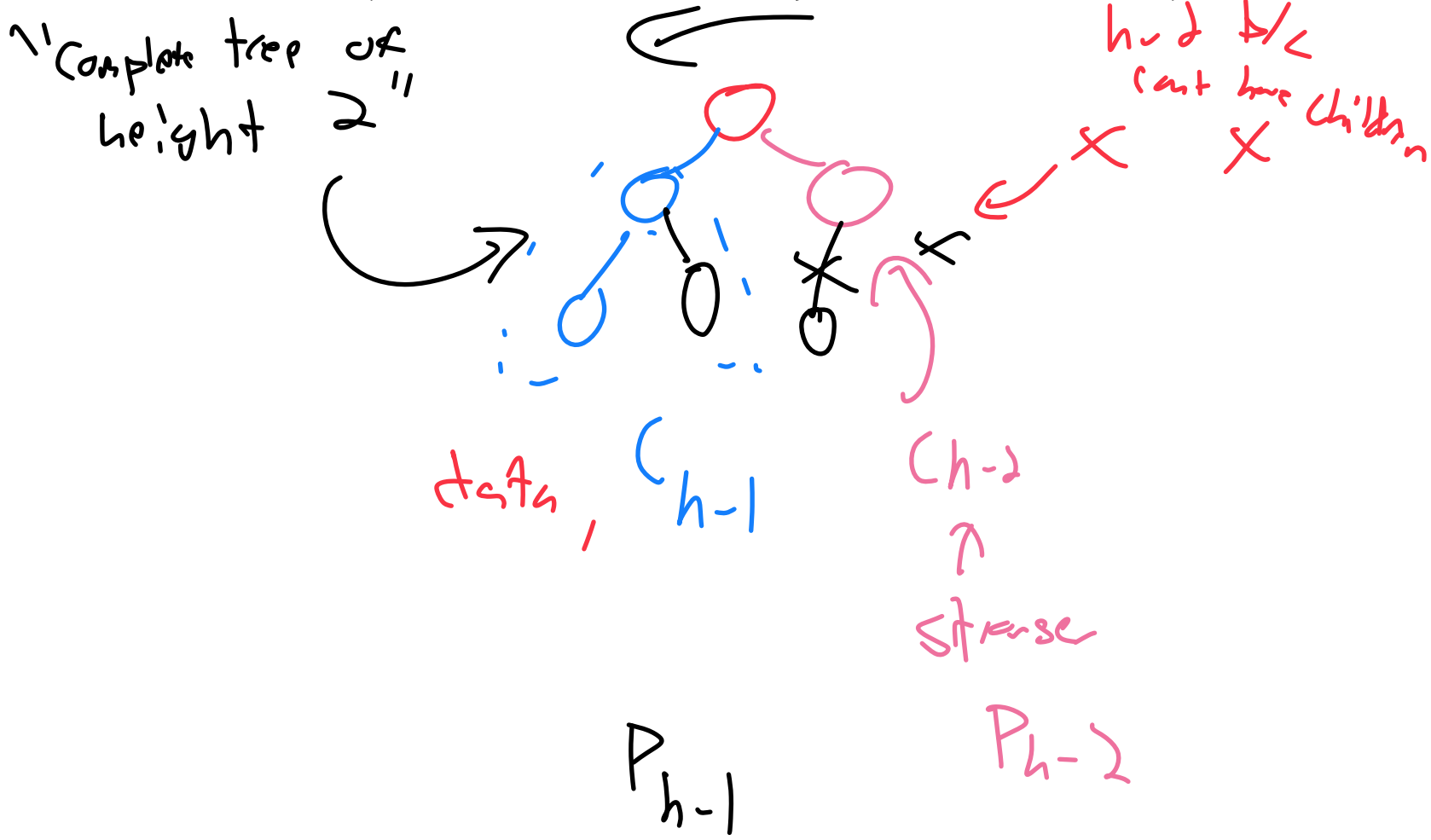


Lets define additional terminology for different **types** of binary trees!

1. Full

2. Perfect

3. Complete



Binary Tree: full

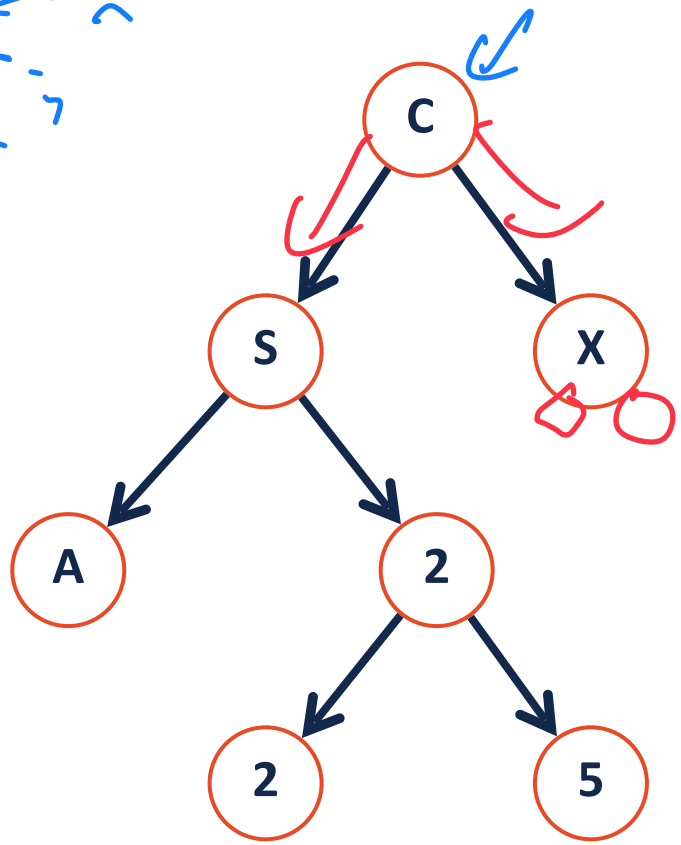
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1. F = null

2. F = (data, null, null) The single node

3. F = (data, F_L^*, F_R^*)* Both not null



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.

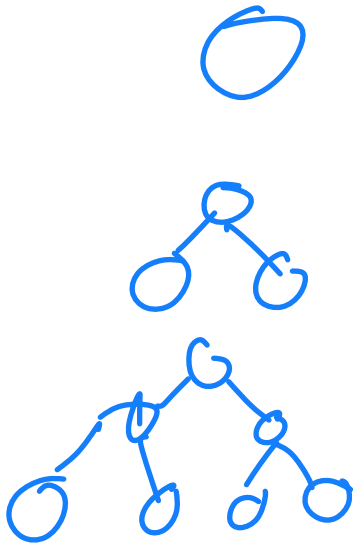
A tree **P** is **perfect** if and only if:

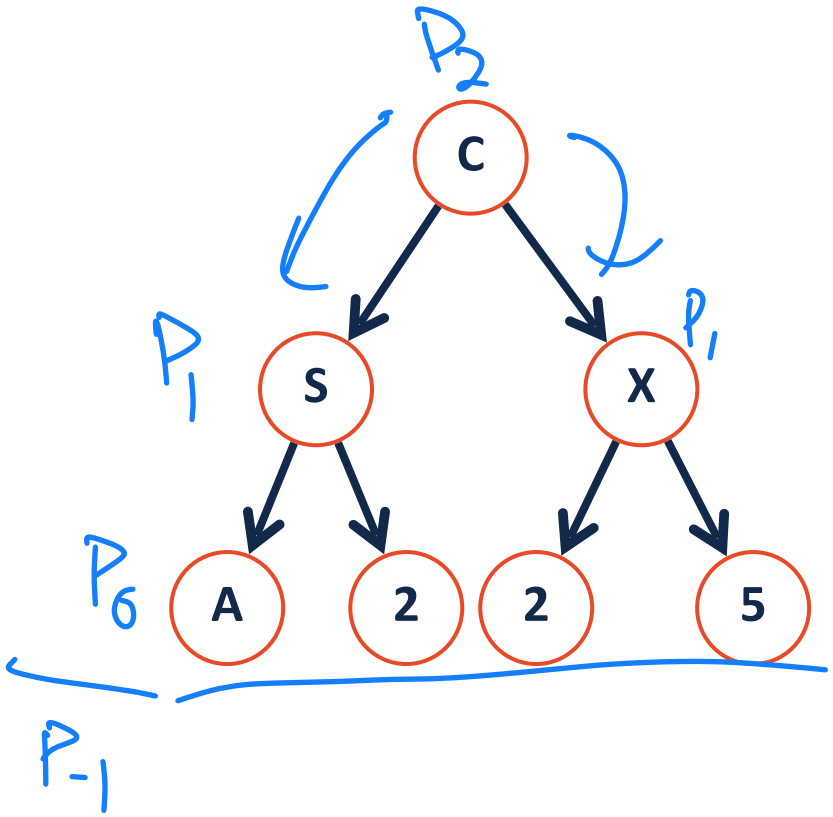
1. $P_h = (data, P_{h-1}, P_{h-1})$

   ↑ right

2. $P_{-1} = null$

$P_0 = (data, null, null)$





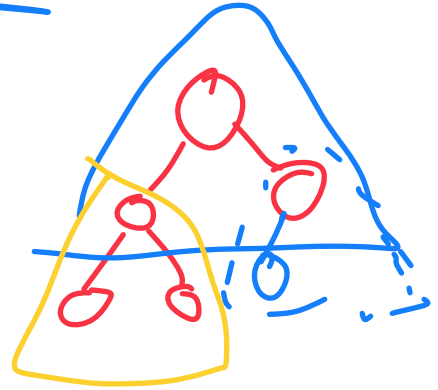
Binary Tree: complete

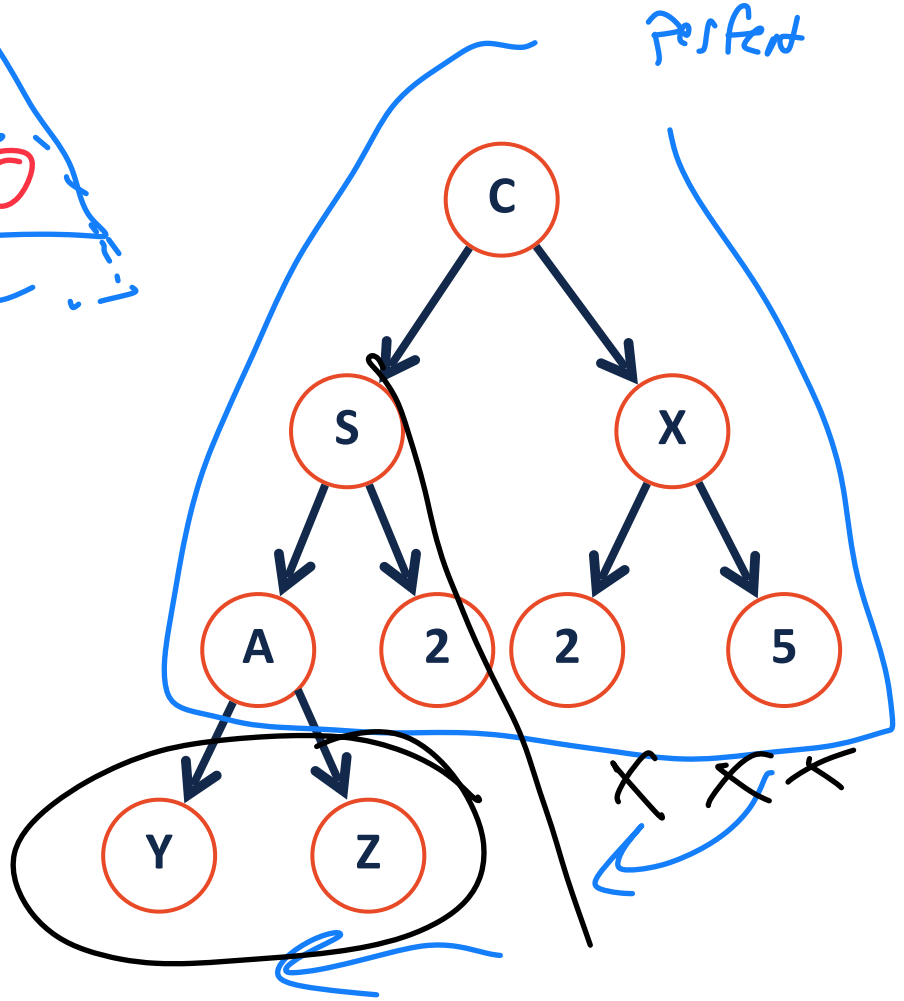
A **complete tree** is a B.T. where…

All levels are completely filled except the last (which 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. $C_{-1} = null$





Binary Tree

Why do we care?

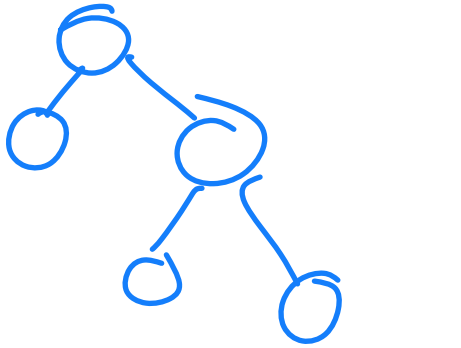
1. Terminology instantly defines a particular tree structure

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

Binary Tree: Thinking with Types

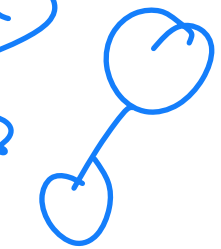
Is every **full** tree **complete**?

No



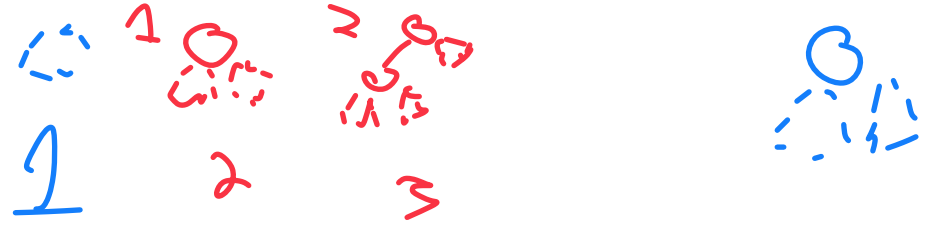
Is every **complete** tree **full**?

No



Yes

Binary Tree: Practicing Proofs



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

Proof by Induction

1) Base cases

2) Assume claim true until arbitrary k. Prove it works k+1

claim n+1 is true QED

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:

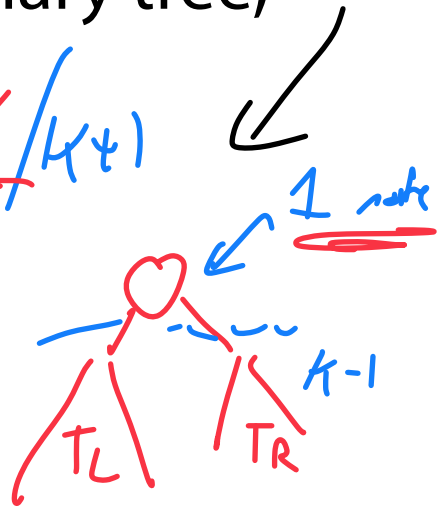
If tree is empty $n=0$, $\rightarrow$ 1 null

$n=1 \rightarrow$ 2 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.

Induction Step: Assume true up to K-1 / K nodes. Prove K / K+1



Let T be a tree of K nodes.

Let q be the number of nodes in T_L.

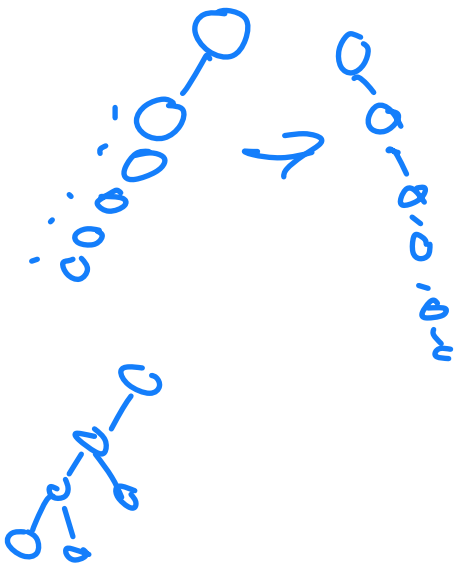
↳ $0 \leq q \leq K-1$ ← Both are smaller than K!

# of nodes in T_R: $K-1-q$ ←

Total # of nulls = nulls in T_L + nulls in T_R

$q+1 + (k-q-1) + 1$

$K+1$ ☺



Tree ADT

insert()

remove()

Traverse()

Find()

getData()

isLeaf()

getHeight()

length() ← nodes?

isEmpty()

~~operator+(const & Tree)~~

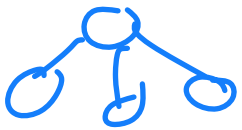
Create tree()

← Also this b/c data structure :<

specific. Trees

Sort()

getLeft() / Right()



BinaryTree.h

```
1  #pragma once
2
3  template <class T>
4  class BinaryTree {
5    public:
6      /* ... */
7
8    private:
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25 };
```

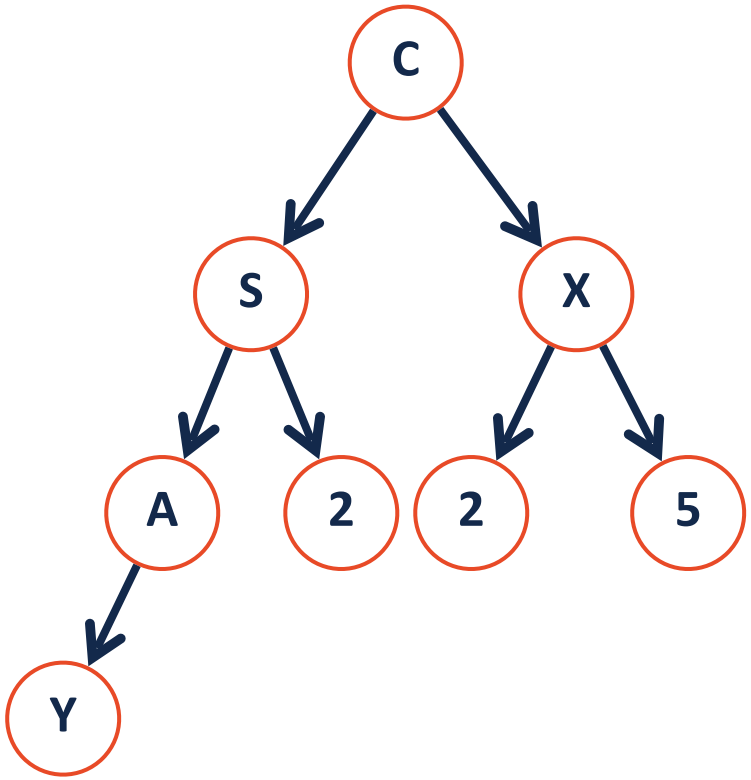
List.h

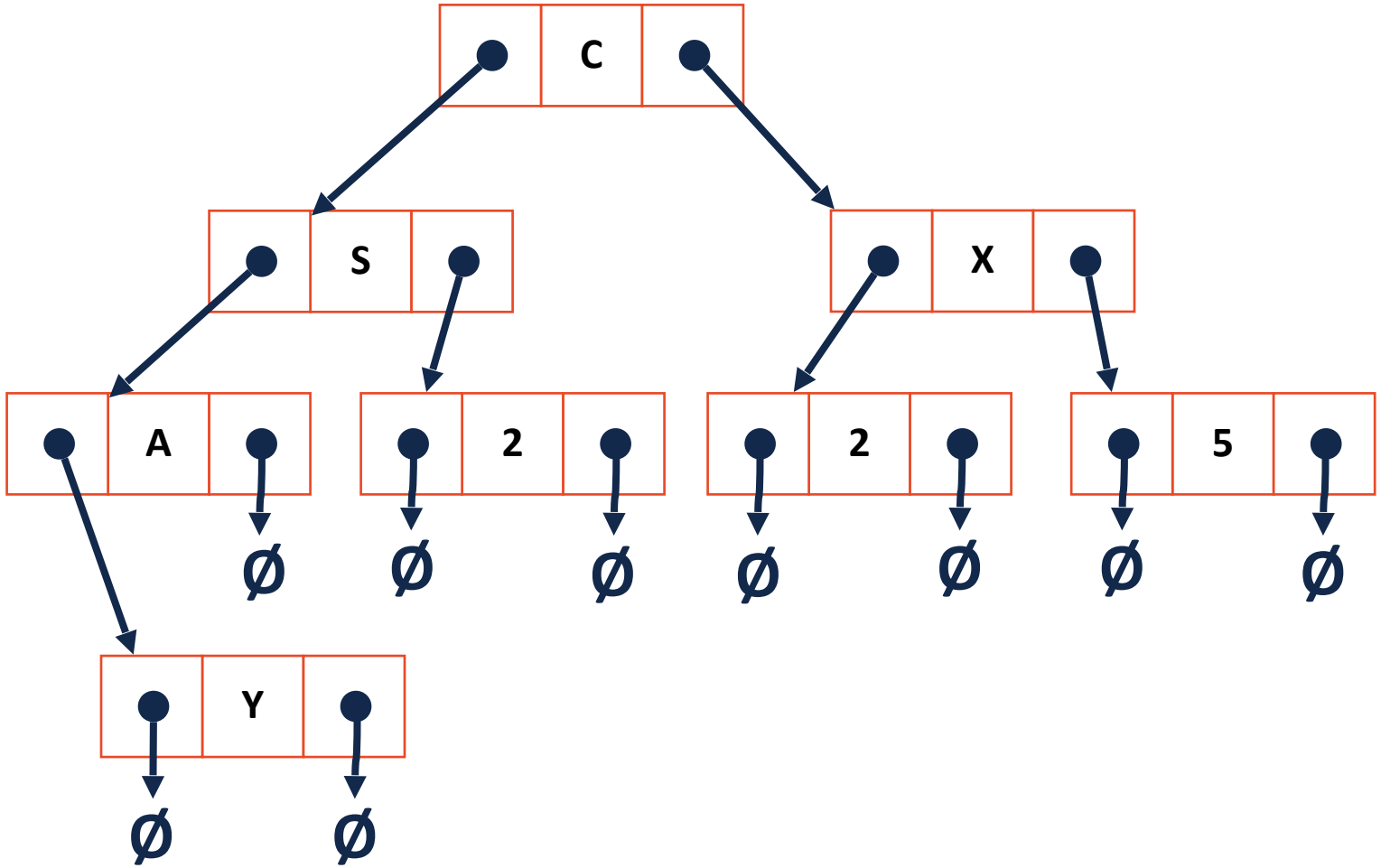
```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
6      /* ... */
7    private:
8      class ListNode {
9        T & data;
10
11       ListNode * next;
12
13
14
15       ListNode(T & data) :
16        data(data), next(NULL) { }
17     };
18
19
20
21     ListNode *head_;
22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

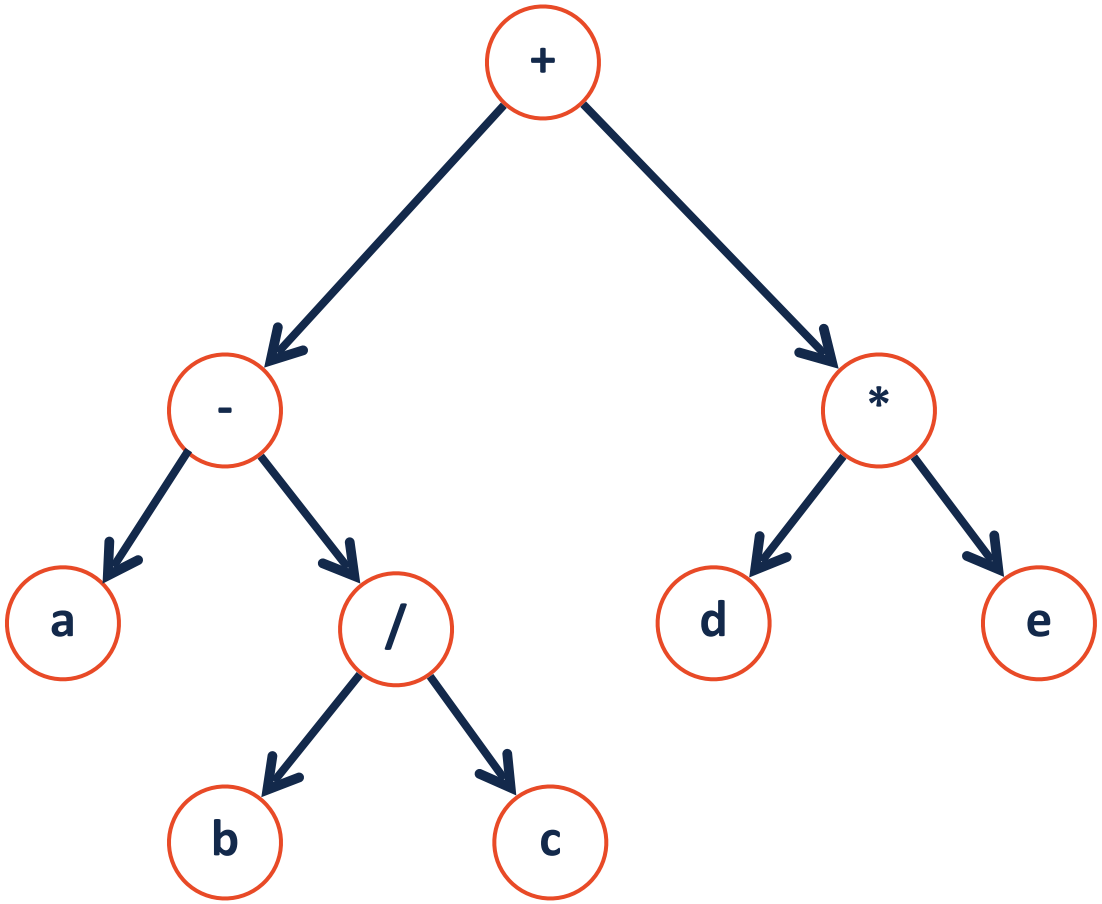
Visualizing trees



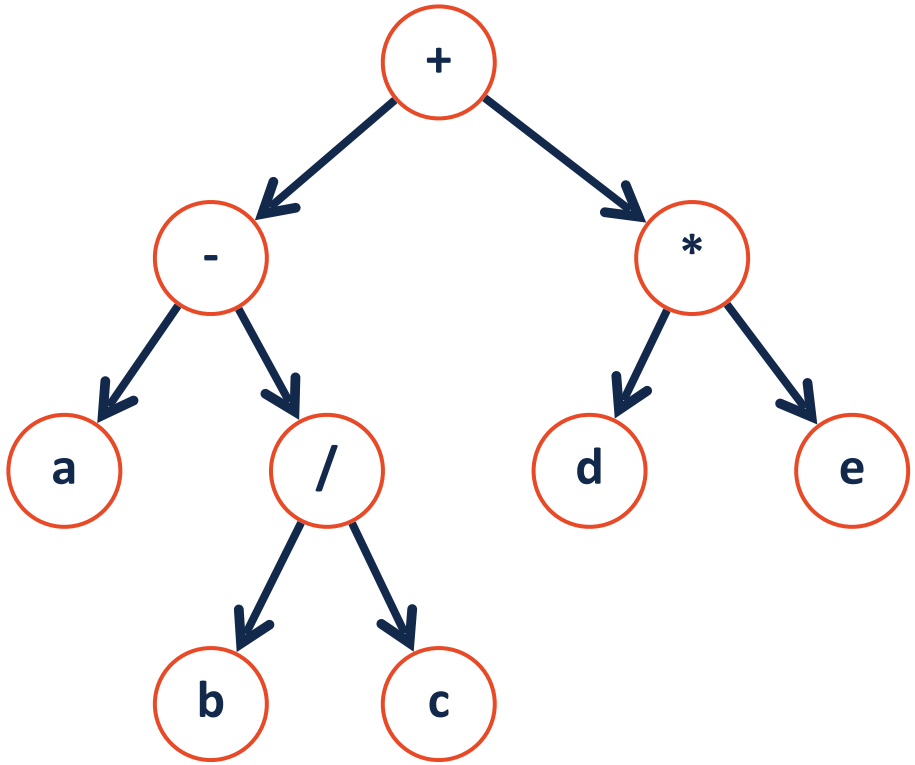


Tree Traversal

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

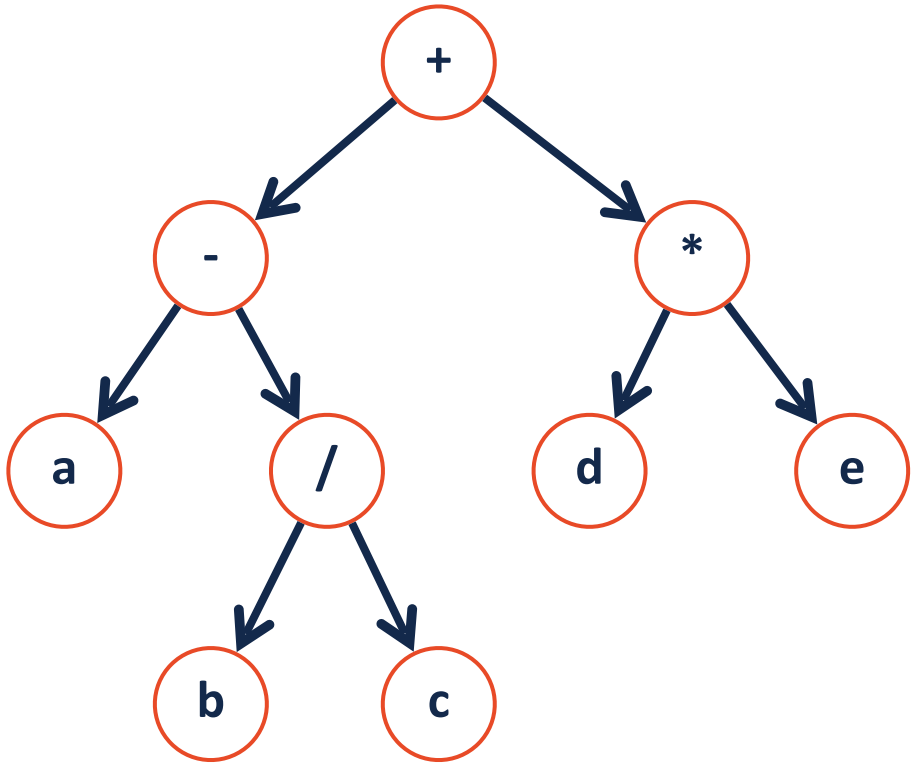


Traversals



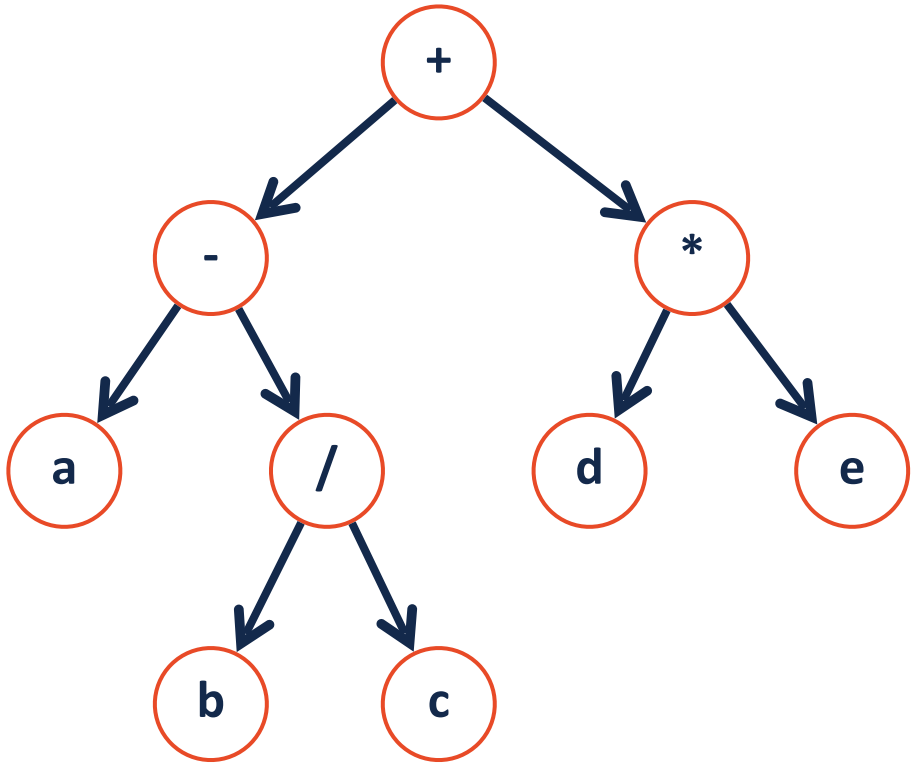

```
template<class T>
void BinaryTree<T>::_____Order(TreeNode * root)
{









}
```

Traversals



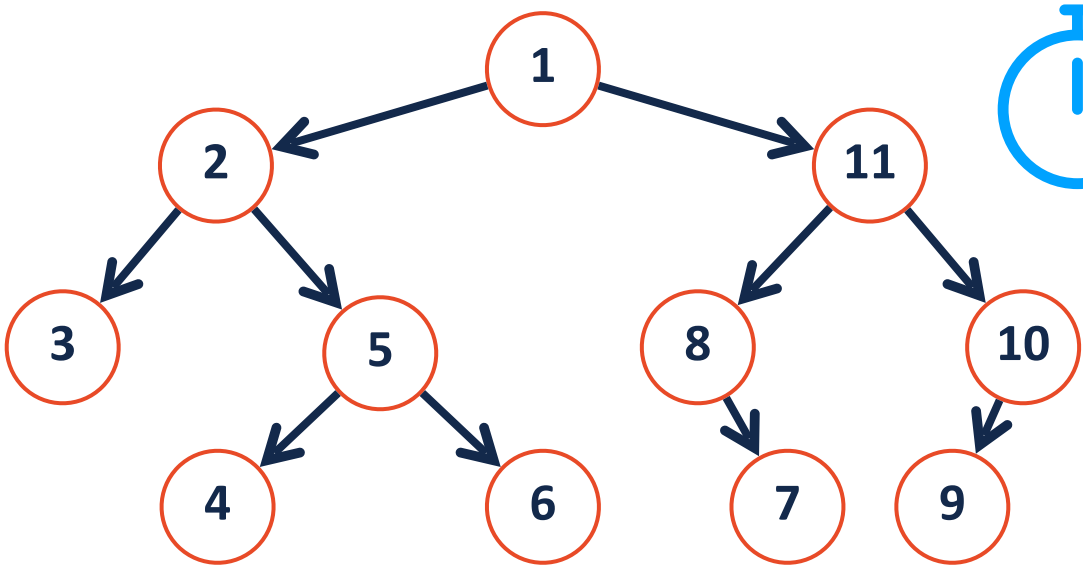

```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5    if (root) {
6
7        ____________________________;
8
9        _____Order(root->left);
10
11       ____________________________;
12
13       _____Order(root->right);
14
15       ____________________________;
16
17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

Traversals




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

Tree Traversals



Pre-order:

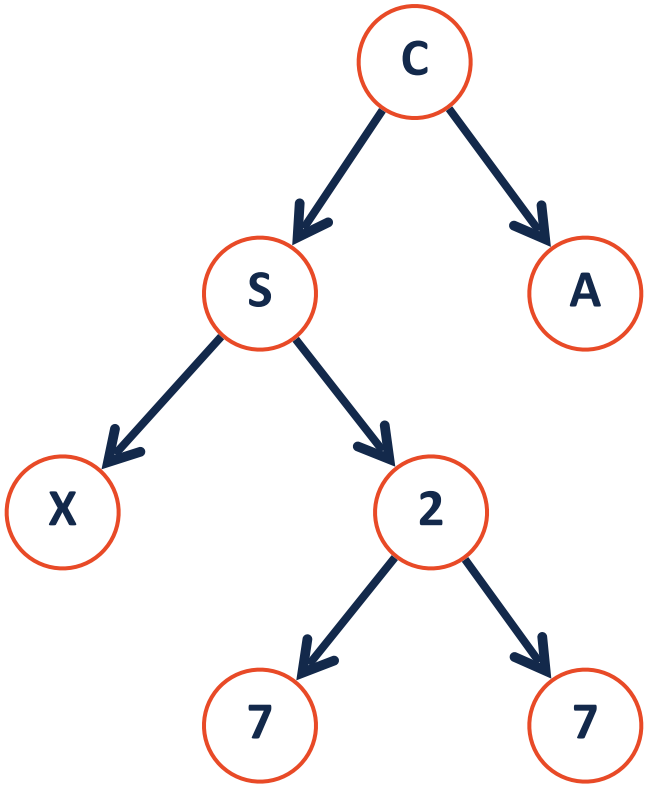
In-order:

Post-order:

Tree Traversals

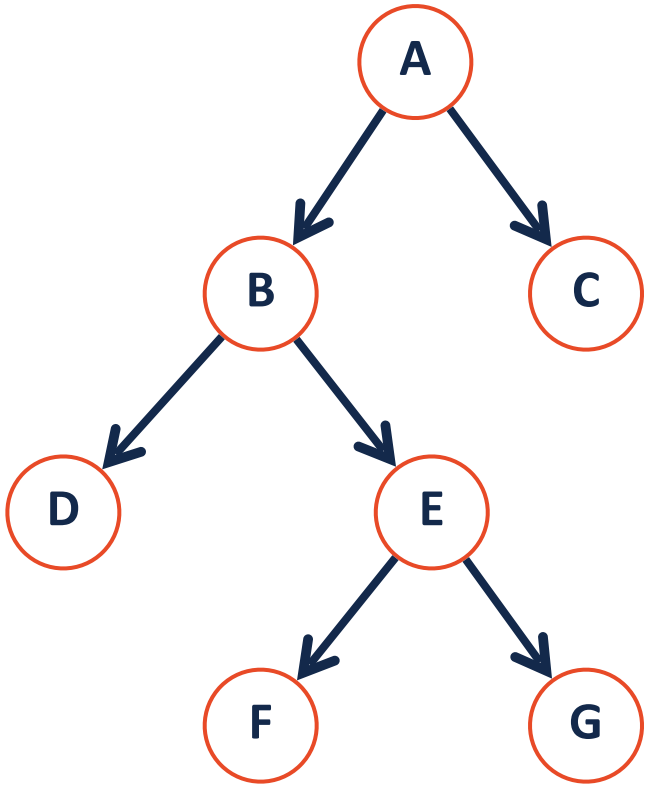
Pre-order: Ideal for copying trees

Post-order: Ideal for deleting trees



Traversal vs Search

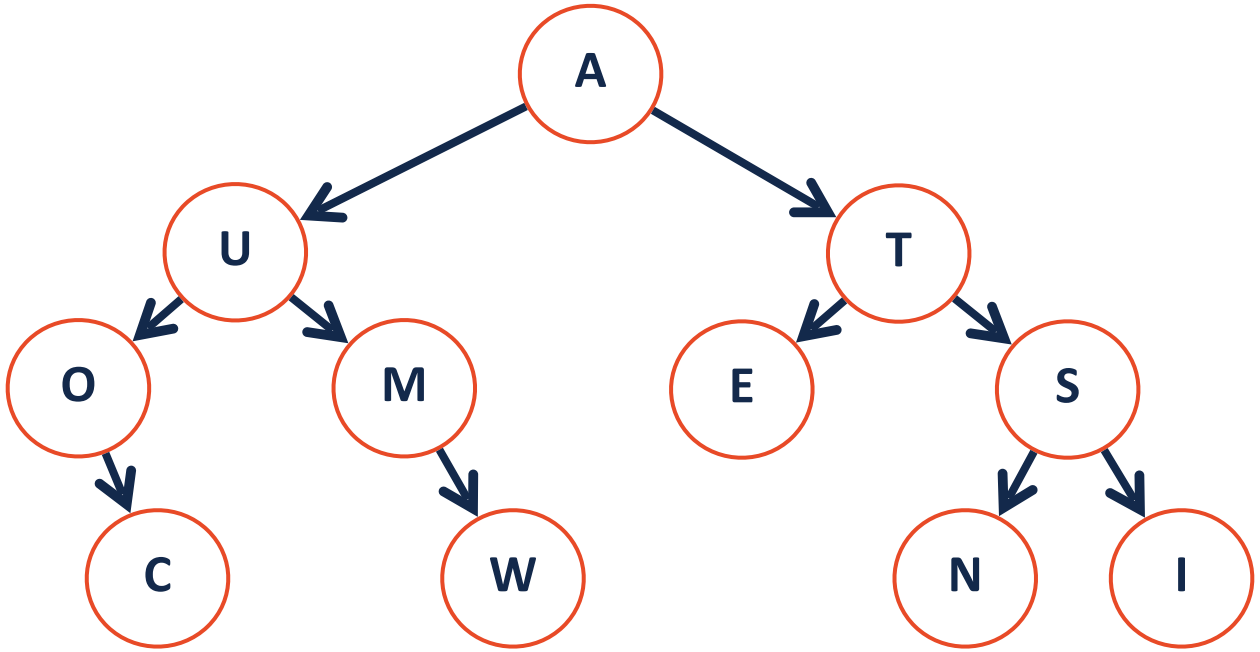
Traversal



Search

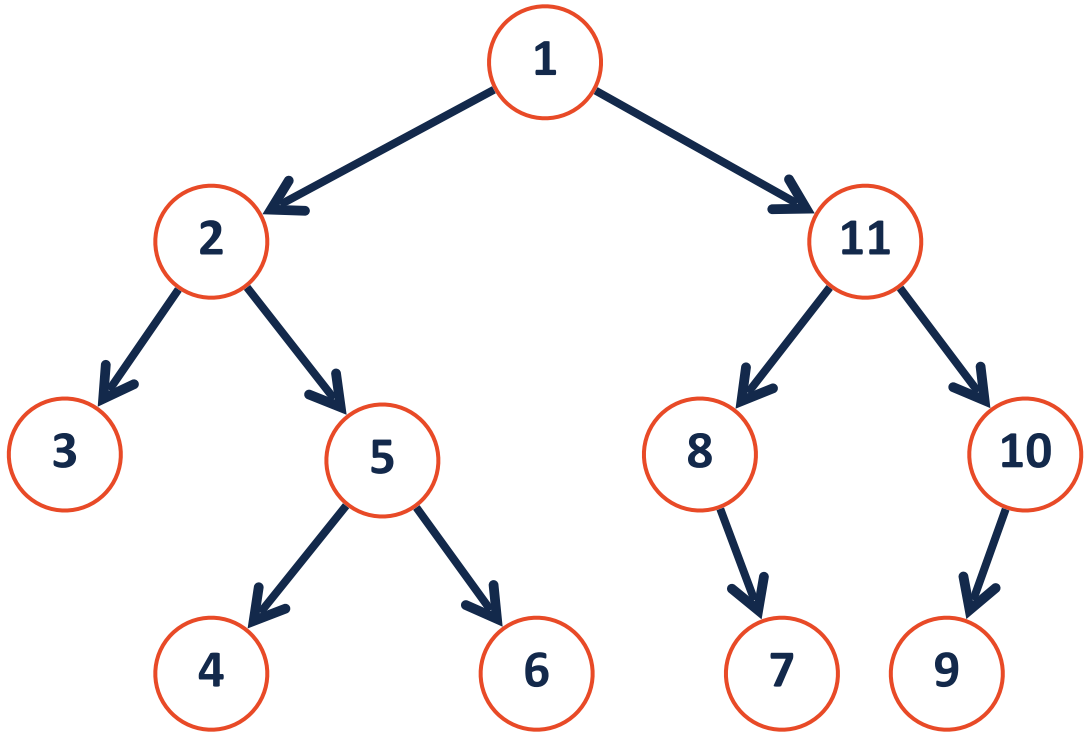
Tree Search

There are two main approaches to searching a binary tree:



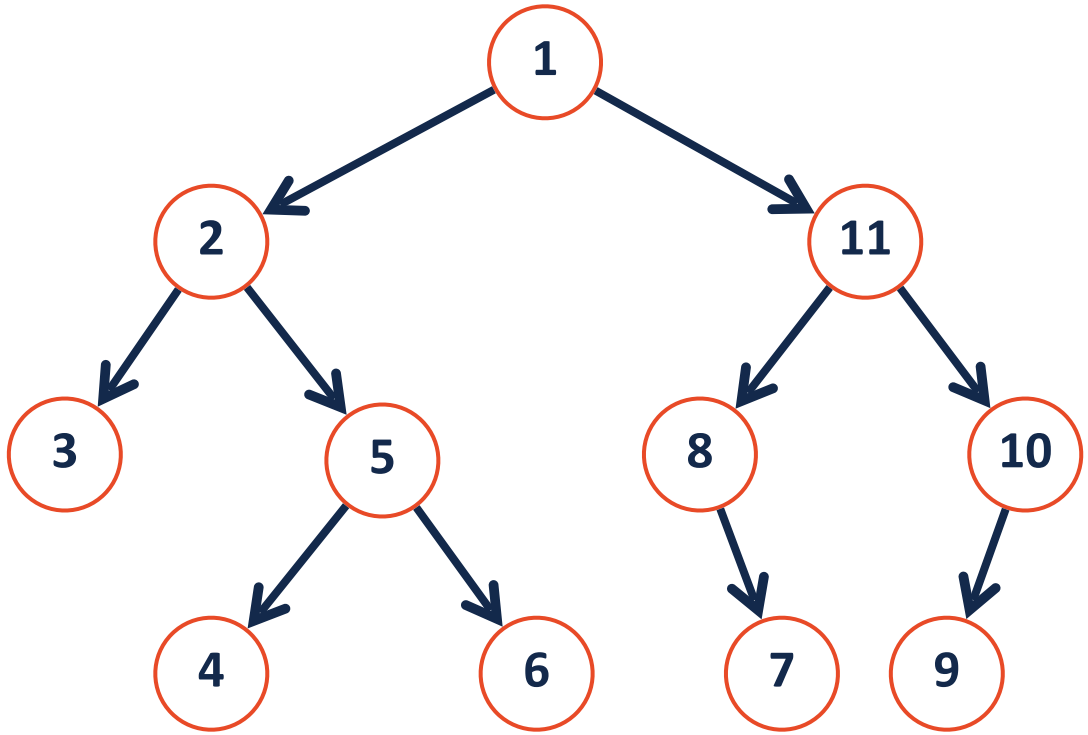
Depth First Search

Explore as far along one path as possible before backtracking

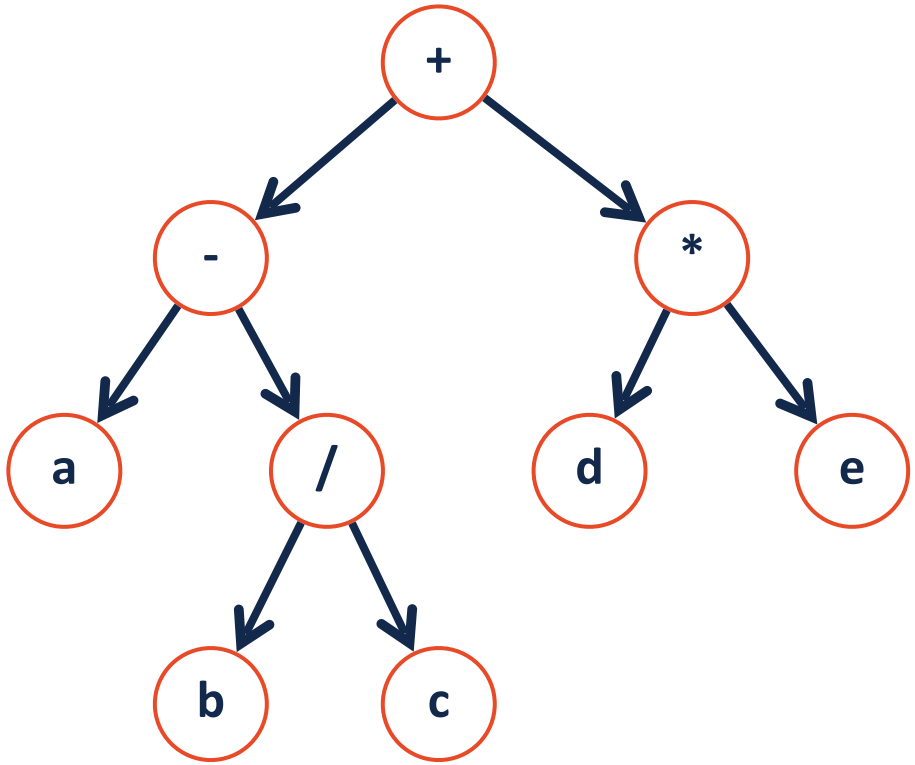


Breadth First Search

Fully explore depth i before exploring depth i+1



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?