Data Structures
Tree Definitions

CS 225
Brad Solomon

September 14, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

MP_Lists out now!

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

Exam 1 (9/16 — 9/18)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam is out on PL now

Topics covered can be found on website

Register now

https://courses.engr.illinois.edu/cs225/fa2025/exams/

Learning Objectives

Review iterators with a practical example

Review trees and binary trees

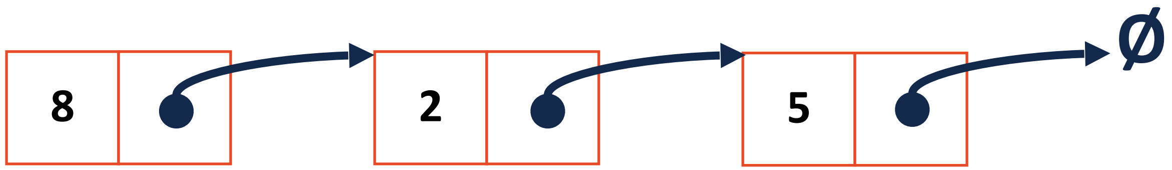
Practice tree theory with recursive definitions and proofs

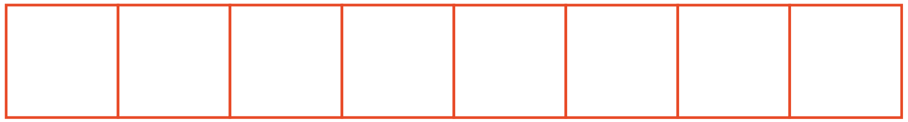
Discuss the tree ADT

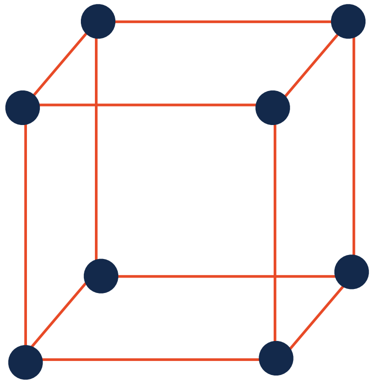
Explore tree implementation details

Iterators

We want to be able to loop through all elements for any underlying implementation in a systematic way







Cur. Location	Cur. Data	Next
`ListNode * curr`	Curr->data	Curr->next
`unsigned index`	data[index]	index++
`Some form of (x, y, z)`	???	???

Iterators

For a class to implement an iterator, it needs two functions:

```
Iterator begin()
```

Returns an Iterator object pointing at the 'first item'

```
Iterator end()
```

Returns an Iterator object pointing one entry past end of dataset

Iterators

The actual iterator is defined as a class **inside** the outer class:

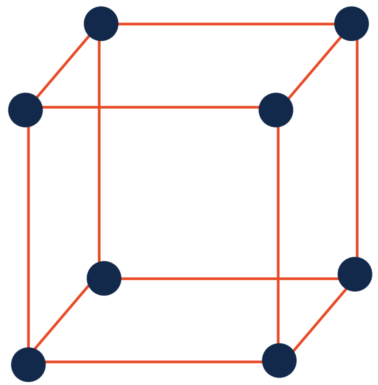
1. It must be of base class **`std::iterator`**

2. It must implement at least the following operations:

```
Iterator& operator ++()

const T & operator *()

bool operator !=(const Iterator &)
```

Iterators

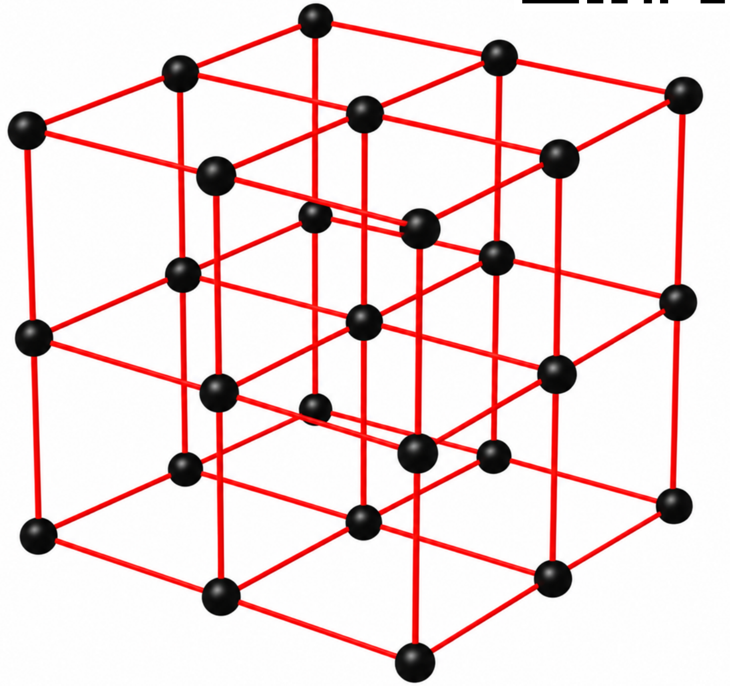
```
class Cube { // ... (Skipping for space)
    class CubeIt { // ... (Skipping for space)

    CubeIt& operator++() {
        ++ix_;
        if (ix_ == cube_->x_dim) {
            ix_ = 0;
            ++iy_;
            if (iy_ == cube_->y_dim) {
                iy_ = 0;
                ++iz_;
            }
        }
        return *this;
```



```
Cube::Iterator it = myCube.begin();

while (it != myCube.end()) {
    std::cout << *it << " ";
    it++;
}
```

```
class Cube { // ... (Skipping for space)
    class CubeIt { // ... (Skipping for space)

    CubeIt& operator++() {
        ++ix_;
        if (ix_ == cube_->x_dim) {
            ix_ = 0;
            ++iy_;
            if (iy_ == cube_->y_dim) {
                iy_ = 0;
                ++iz_;
            }
        }
        return *this;
```

Join Code: 225



If Cube.begin() returns a CubeIt(0, 0, 0),
what does Cube.end() return?

A) CubeIt(x_dim, y_dim, z_dim)

B) CubeIt(0, 0, z_dim)

C) CubeIt(-1, -1, -1)

D) More than one correct

Iterators

Implementing the mandatory interface enables easy looping!

```
template <class T>
class List {

    class ListIterator : public std::iterator<std::bidirectional_iterator_tag, T> {
      public:

        ListIterator& operator++();

        ListIterator& operator--()

        bool operator!=(const ListIterator& rhs);

        const T& operator*();
    };

    ListIterator begin() const;

    ListIterator end() const;
};
```

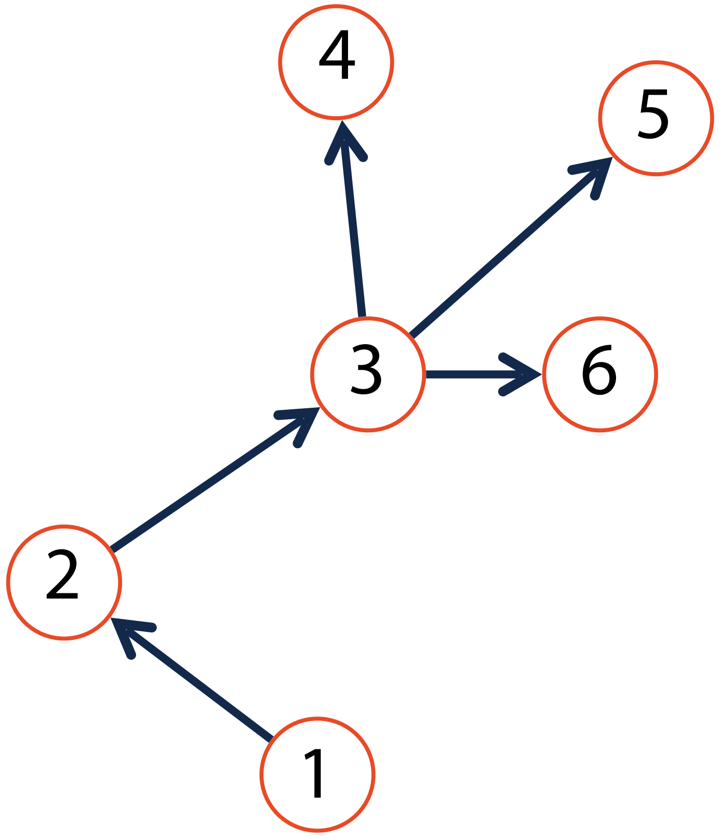
Trees

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

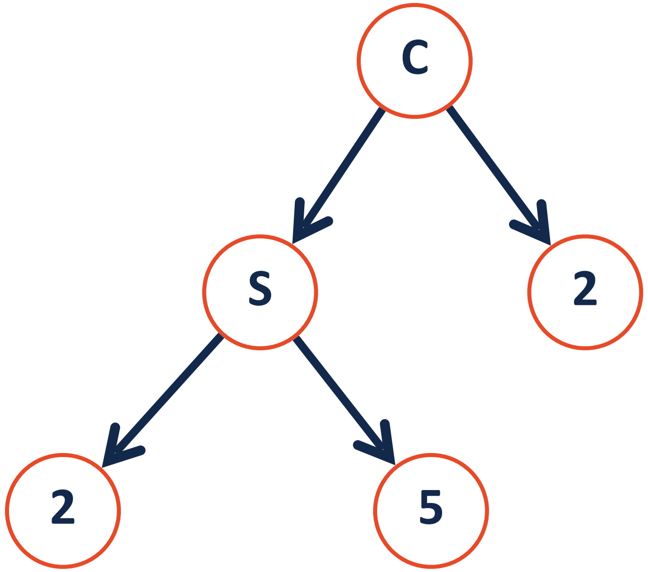


Binary Tree

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

1. $T = \emptyset$

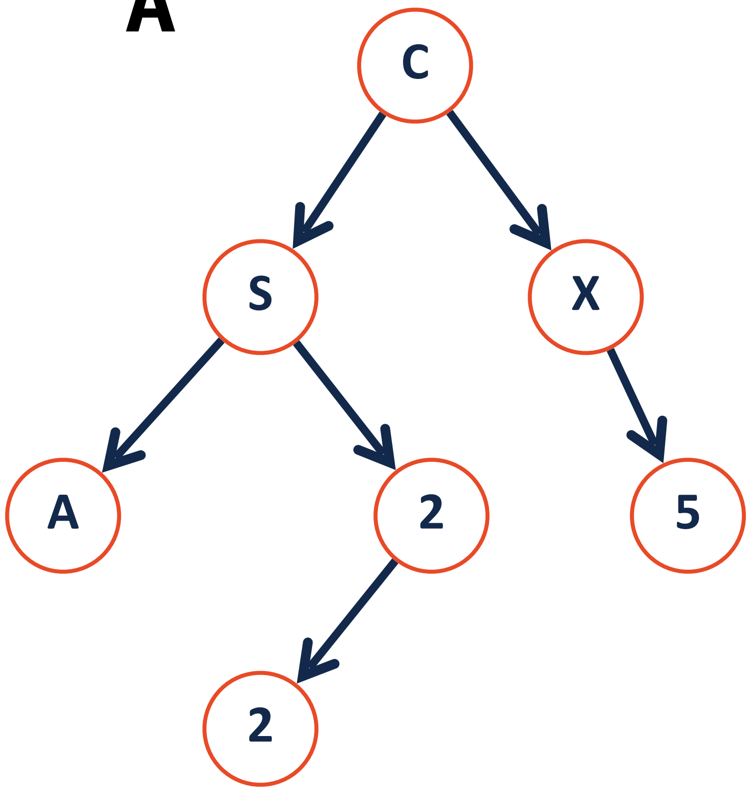
2. $T = (data, T_L, T_R)$



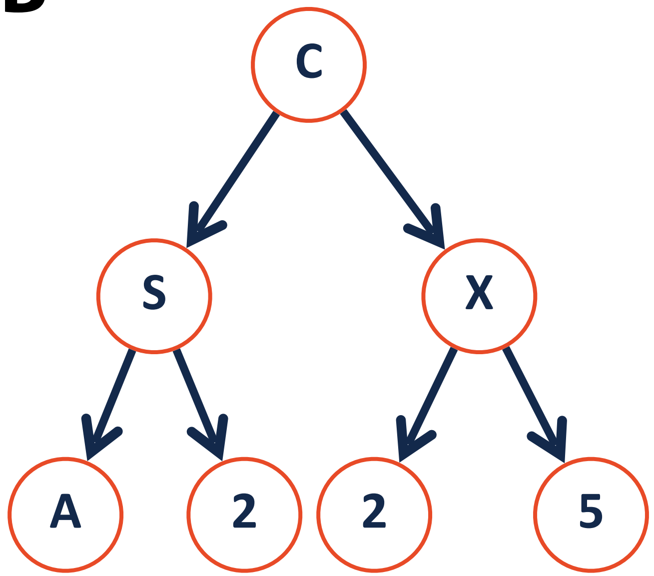
Which of the following are binary trees?

Join Code: 225

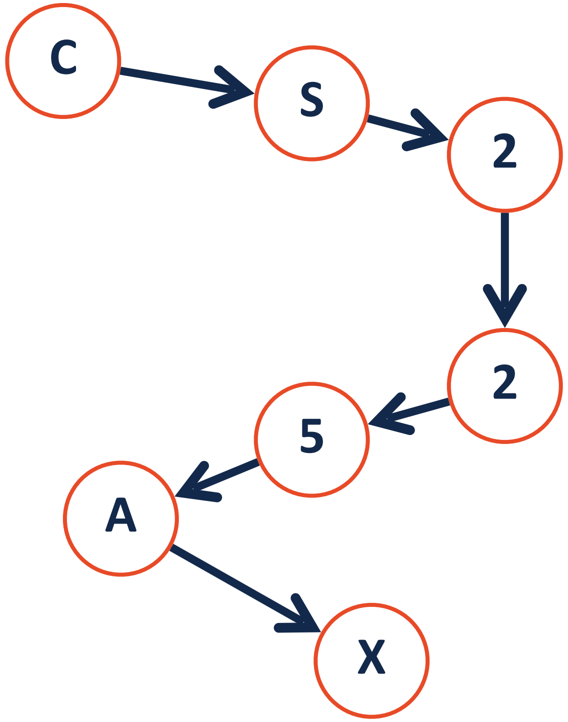
A



B

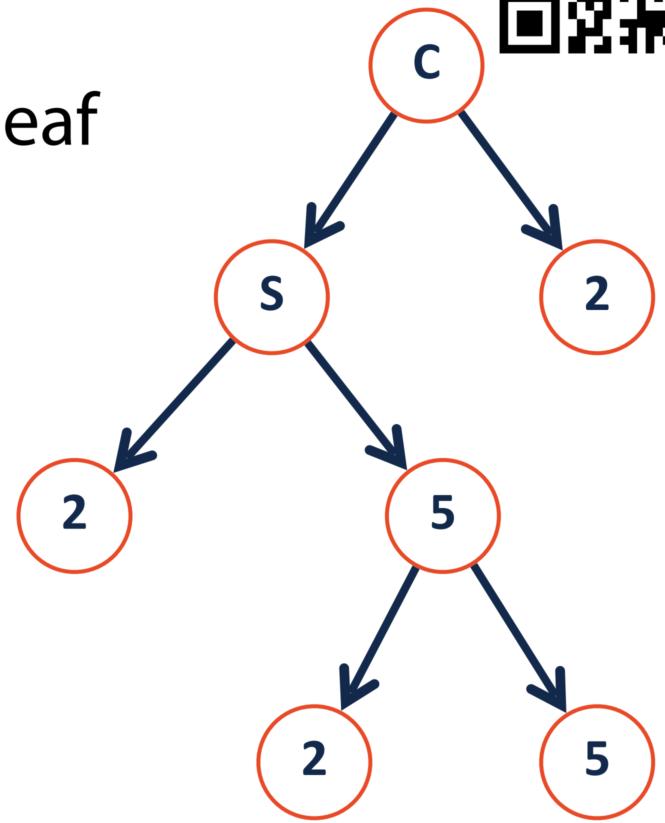


C



Binary Tree Height

Height: The length of the longest path from root to leaf



What is the height of a tree with ***zero*** nodes?

Binary Tree Height

```
height(T) = 1 + max(height(TL), height(TR))
```

Base Case: The height of the empty tree is -1

Recursive Step: Get height of left and right subtrees

Combining: Tree height is 1 plus the max of left or right height

Binary Tree

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

1.

2.

3.

Binary Tree: full

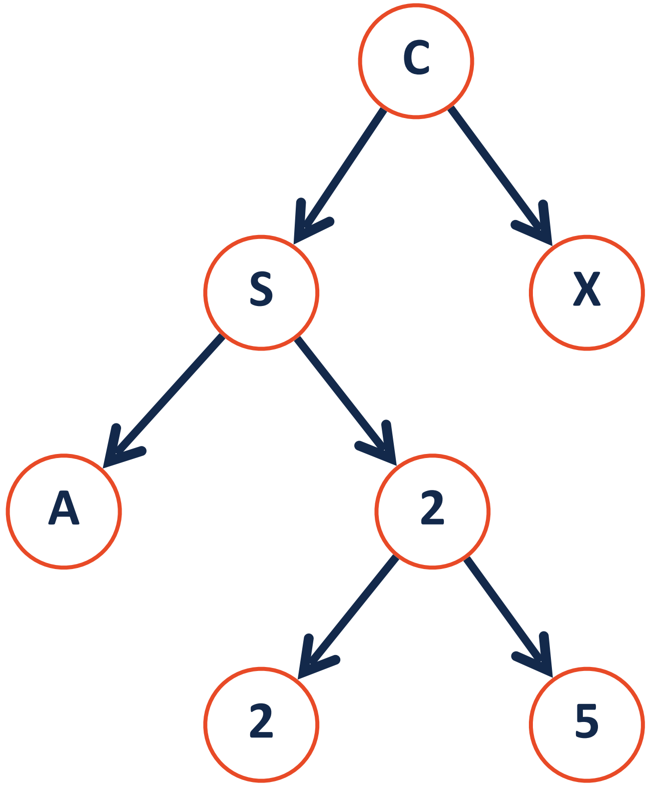
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1.

2.

3.



Binary Tree: full

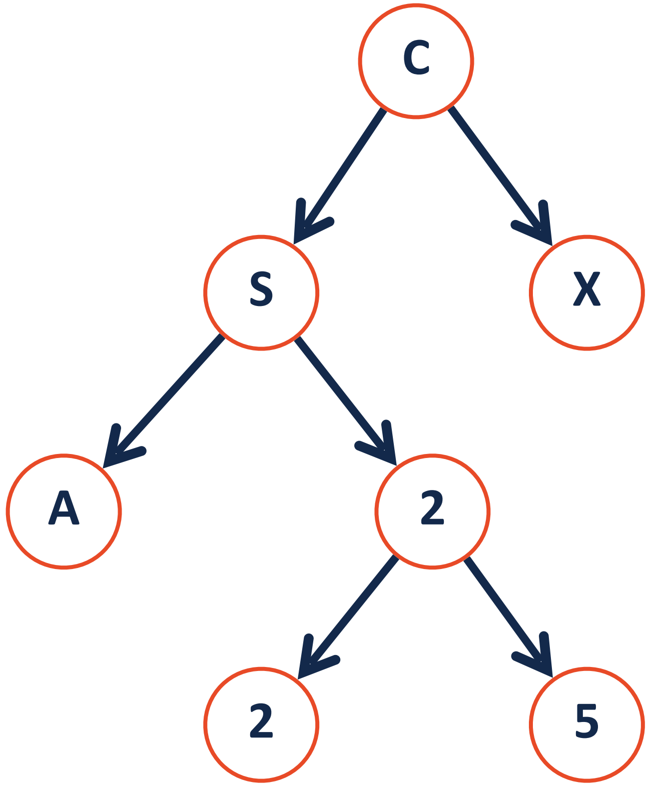
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1. $F = \emptyset$

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

3. $F = (data, F_l \neq \emptyset, F_r \neq \emptyset)$



Binary Tree: full

Join Code: 225

Which of the following are possible sizes (# of nodes) for…

… a full binary tree?

A) 2 Nodes

C) 7 Nodes

B) 5 Nodes

D) 8 Nodes

Binary Tree: perfect

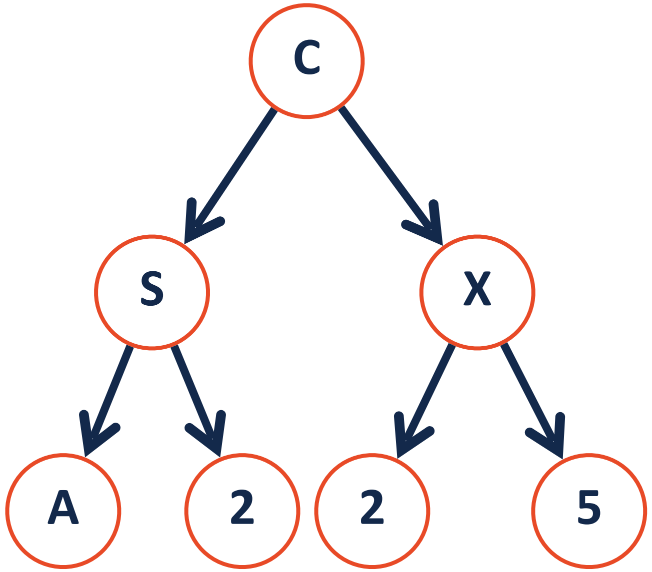
A **perfect tree** is a binary tree where…

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

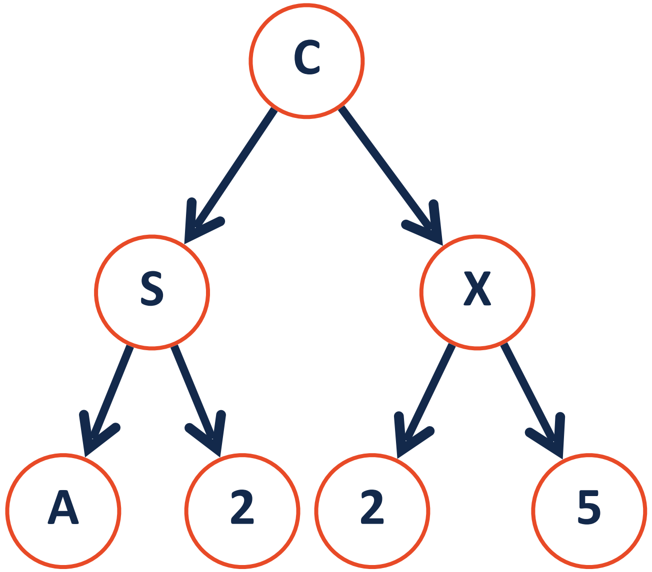
A **perfect tree** is a binary tree where…

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

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

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

2. $P_0 = (data, \emptyset, \emptyset) \equiv \; P_{-1} = \emptyset$



Binary Tree: perfect

Join Code: 225

Which of the following are possible sizes (# of nodes) for…

… a perfect binary tree?

A) 2 Nodes

B) 5 Nodes

C) 7 Nodes

D) 8 Nodes

Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

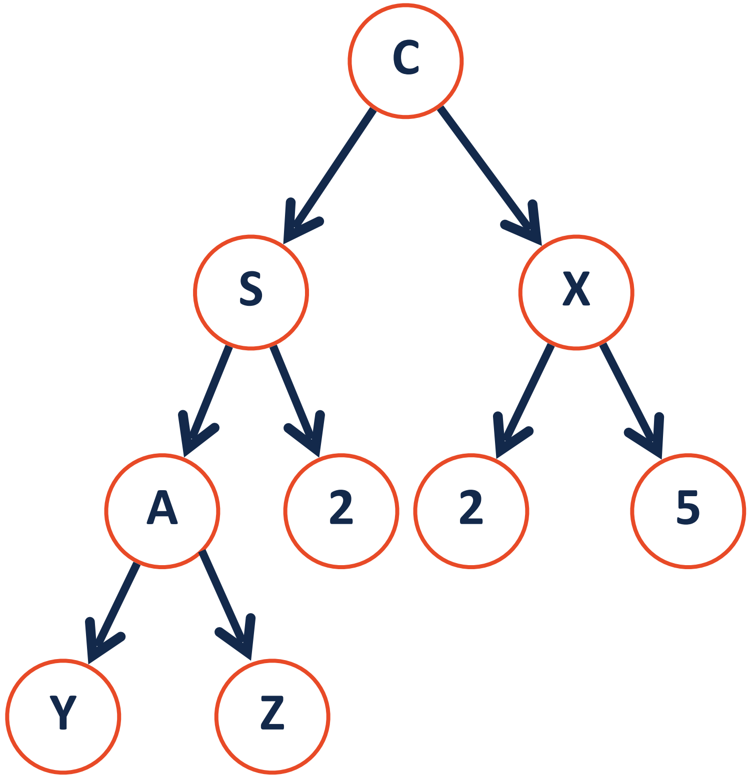
The last level contains at least one node (and is pushed to left)

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.

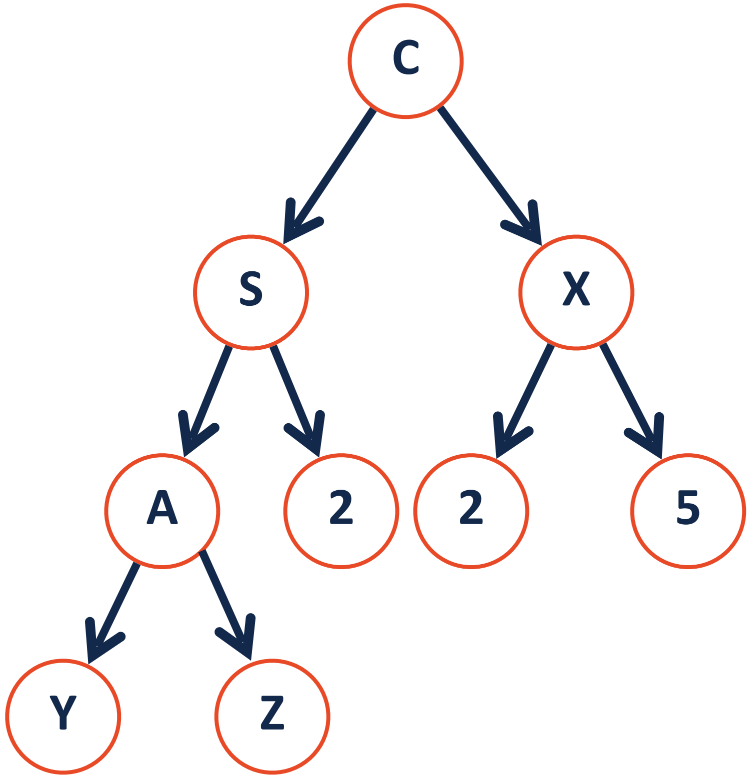
The last level contains at least one node (and is pushed to left)

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: perfect

Join Code: 225

Which of the following are possible sizes (# of nodes) for…

… a perfect binary tree?

A) 2 Nodes

C) 7 Nodes

B) 5 Nodes

D) 8 Nodes

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**?

Binary Tree: Practicing Proofs

Theorem: If there are **n** objects in our representation of a binary tree, then there are _________ 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:

Binary Tree: Practicing Proofs

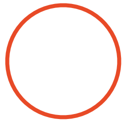
Theorem: If there are **n** objects in our representation of a binary tree, then there are **n+1** NULL pointers.

Base Case:

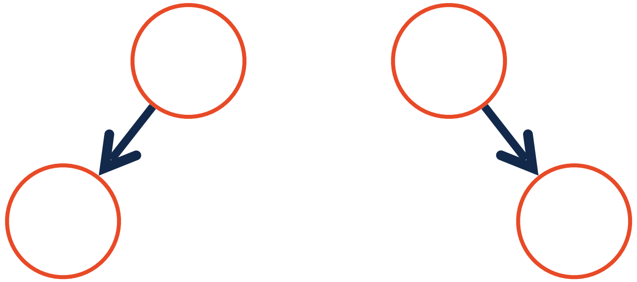
Let F(n) be the max number of NULL pointers in a tree of n nodes

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:

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 => 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 => total (n-1) pointers pointing to them

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

Tree ADT

Insert

Remove

Traverse

Find

Constructor

BinaryTree.h

```
#pragma once

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

  private:
















};
```

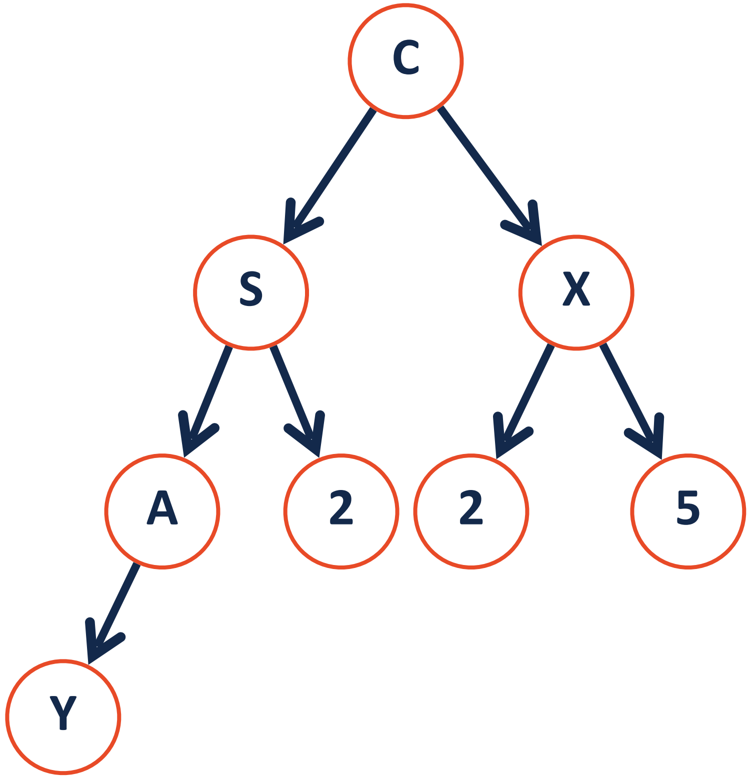
List.h

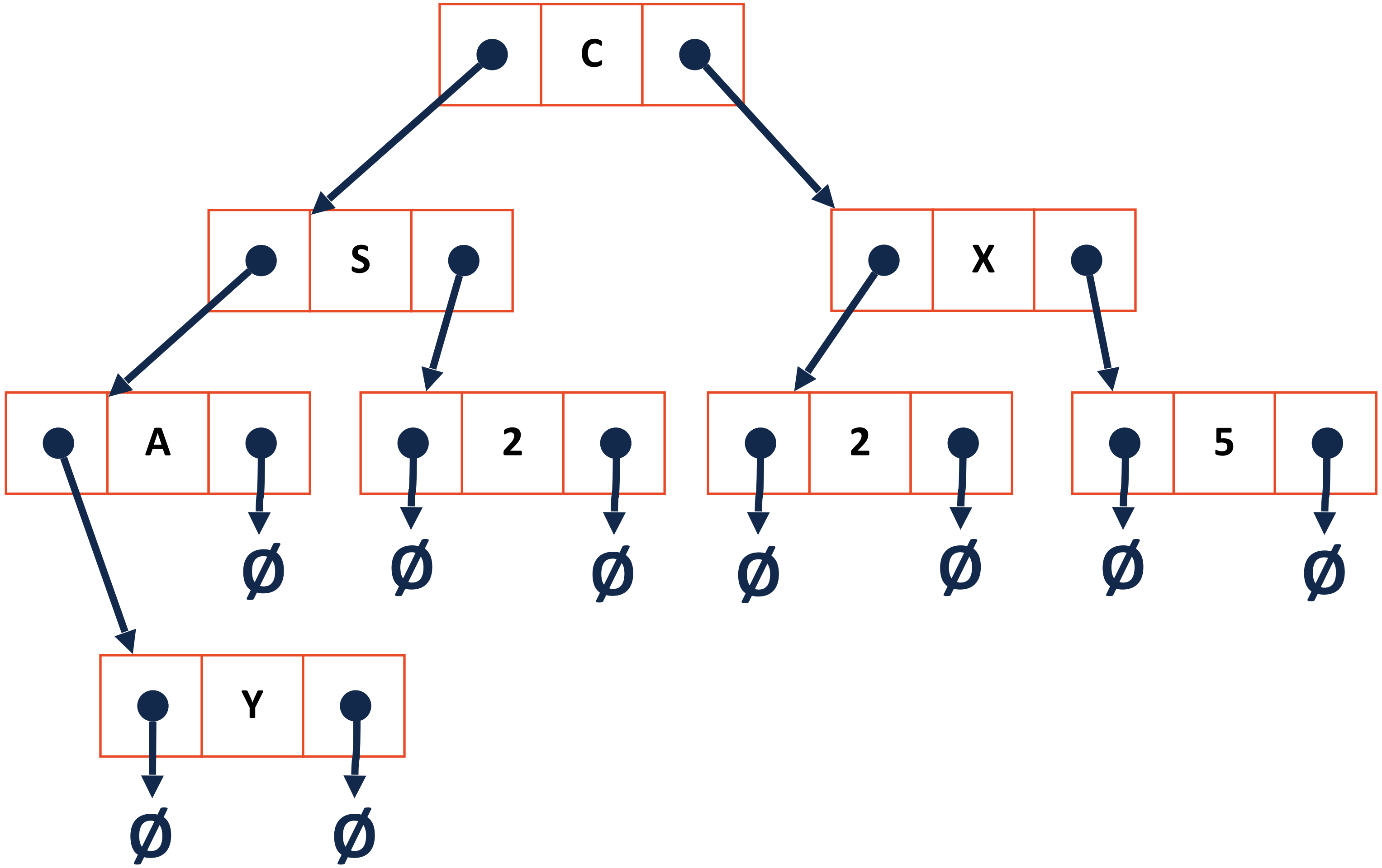
```cpp
#pragma once

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

      ListNode * next;



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



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

Tree.h

```cpp
#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_;
    /* ... */
};
```

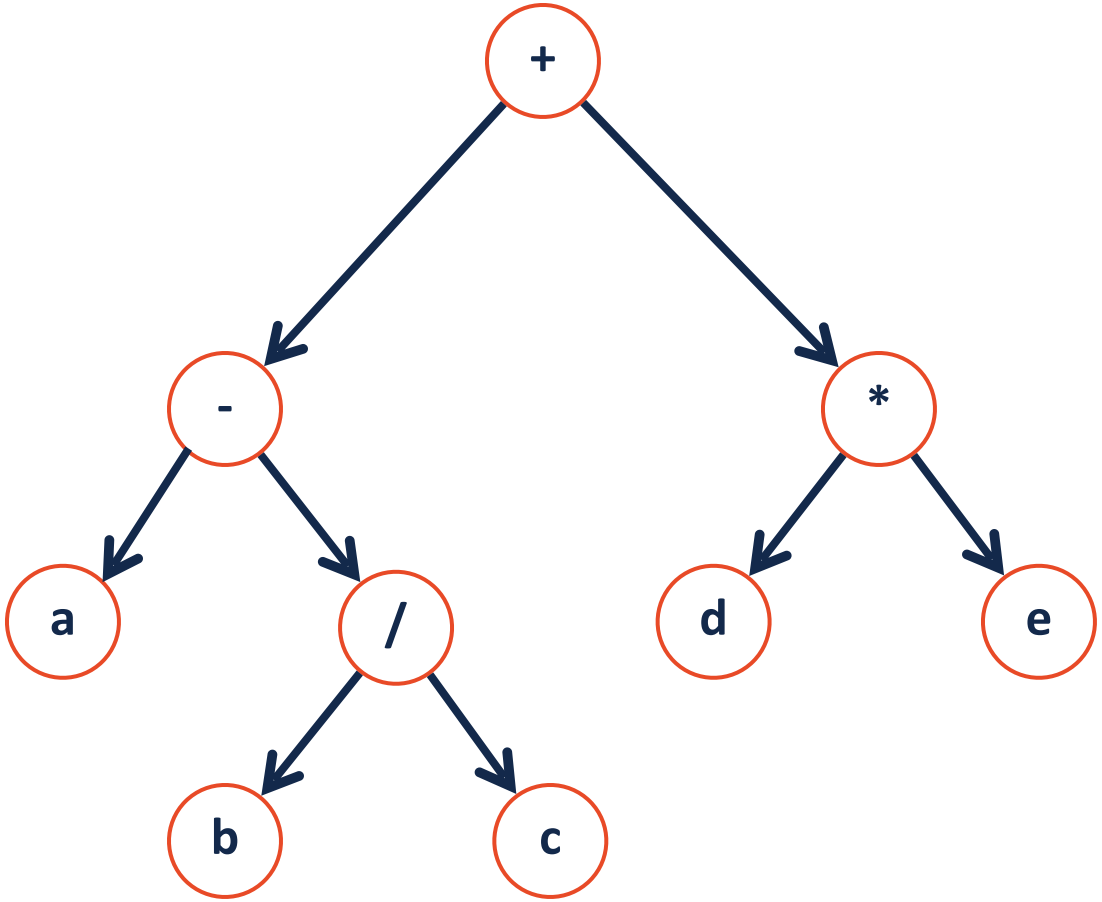
Visualizing trees



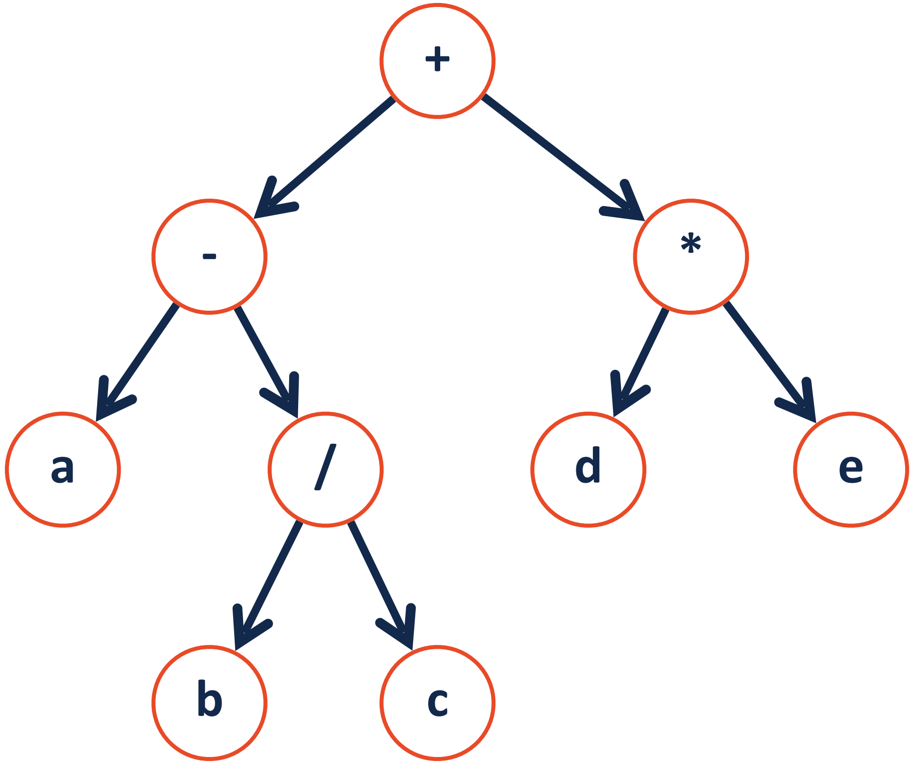


Tree Traversal

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)
{
}
```