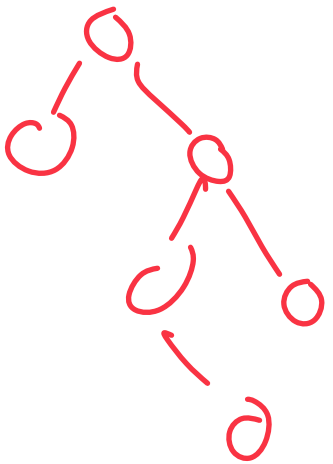
First Honors lecture today 5-6 PM
Siebel 0216
(static variables, functional language)

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

Request extensions for labs / MPs using online form!

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

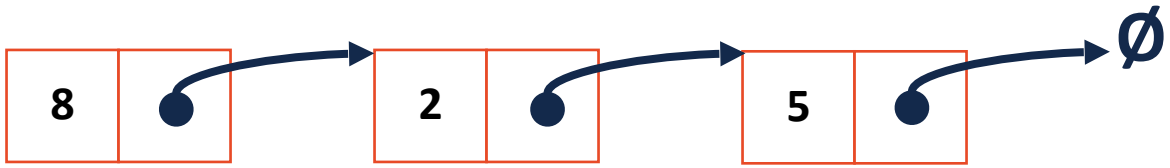
Practice tree theory with recursive definitions and proofs

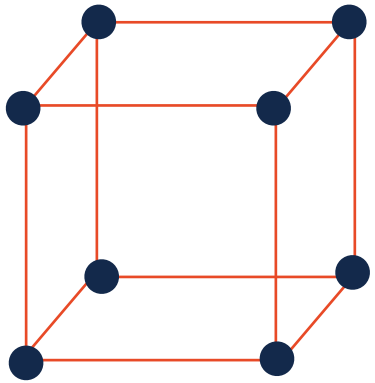
Discuss the tree ADT

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

```
1  class Cube { // ... (Skipping for space)
2      class CubeIt { // ... (Skipping for space)
3
4      CubeIt& operator++() {
5          ++ix_;
6          if (ix_ == cube_->x_dim) {
7              ix_ = 0;
8              ++iy_;
9              if (iy_ == cube_->y_dim) {
10                 iy_ = 0;
11                 ++iz_;
12             }
13         }
14         return *this;
```

(2)

user

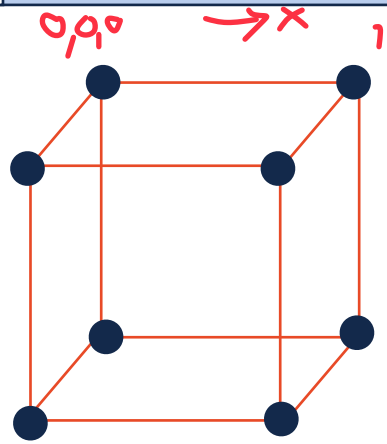
(0,0,0)

+1

check

0 0 0
1 0 0
0 1 0
1 1 0
0 0 1

gets run automatically

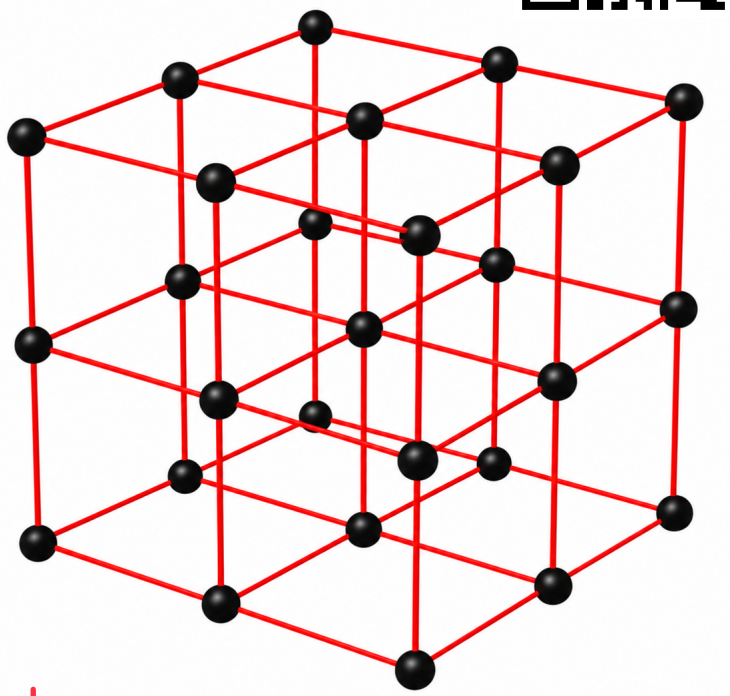



```
1  Cube::Iterator it = myCube.begin();
2
3  while (it != myCube.end()) {
4      std::cout << *it << " ";
5      it++;
6  }
7
```

```
1  class Cube { // ... (Skipping for space)
2      class CubeIt { // ... (Skipping for space)
3
4      CubeIt& operator++() {
5          ++ix_;
6          if (ix_ == cube_->x_dim) {
7              ix_ = 0;
8              ++iy_;
9              if (iy_ == cube_->y_dim) {
10                 iy_ = 0;
11                 ++iz_;
12             }
13         }
14         return *this;
```

Join Code: 225

(0,0,0)



x-dim-1 , y-dim-1 , z-dim-1

If Cube.begin() returns a CubeIt(0, 0, 0),

what does Cube.end() return?

A) CubeIt(x_dim, y_dim, z_dim) — 60%

B) CubeIt(0, 0, z_dim) — 20%

Because of my implementation

C) CubeIt(-1, -1, -1) — 10%

D) More than one correct — 10%

Iterators

Implementing the mandatory interface enables easy looping!

```
1  template <class T>
2  class List {
3
4      class ListIterator : public
   std::iterator<std::bidirectional_iterator_tag, T> {
5          public:
6
7              ListIterator& operator++();
8
9              ListIterator& operator--()
10
11             bool operator!=(const ListIterator& rhs);
12
13             const T& operator*();
14     };
15
16     ListIterator begin() const;
17
18     ListIterator end() const;
19 };
```

what it means to be of type 'iterator'

min required as *

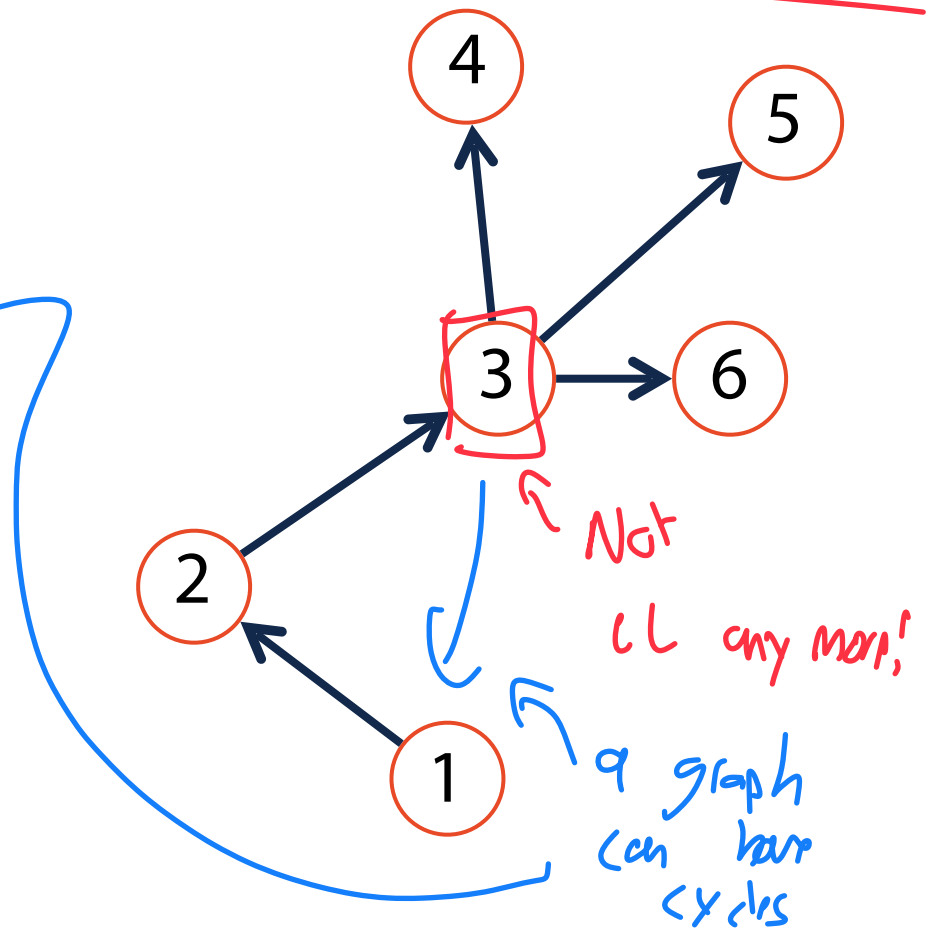
Trees

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

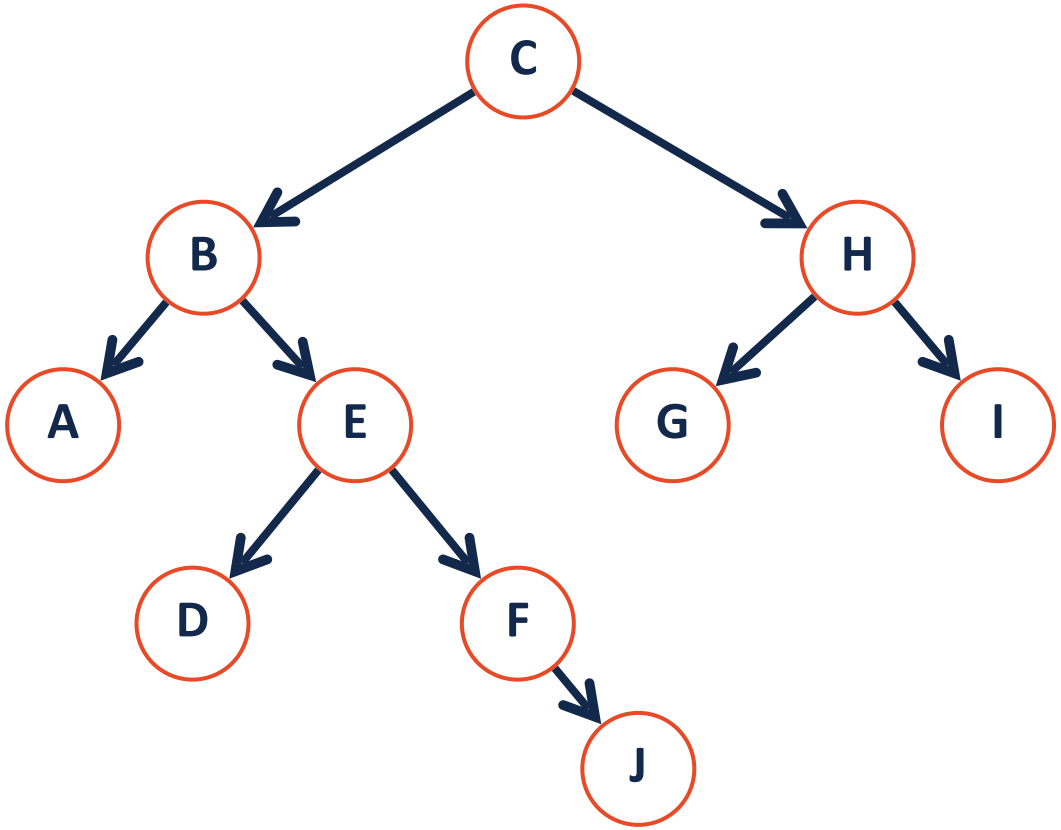
[In CS 225] a tree is also:

1) Acyclic — No path from node to itself

2) Rooted — A specific node is labeled root



Tree Terminology

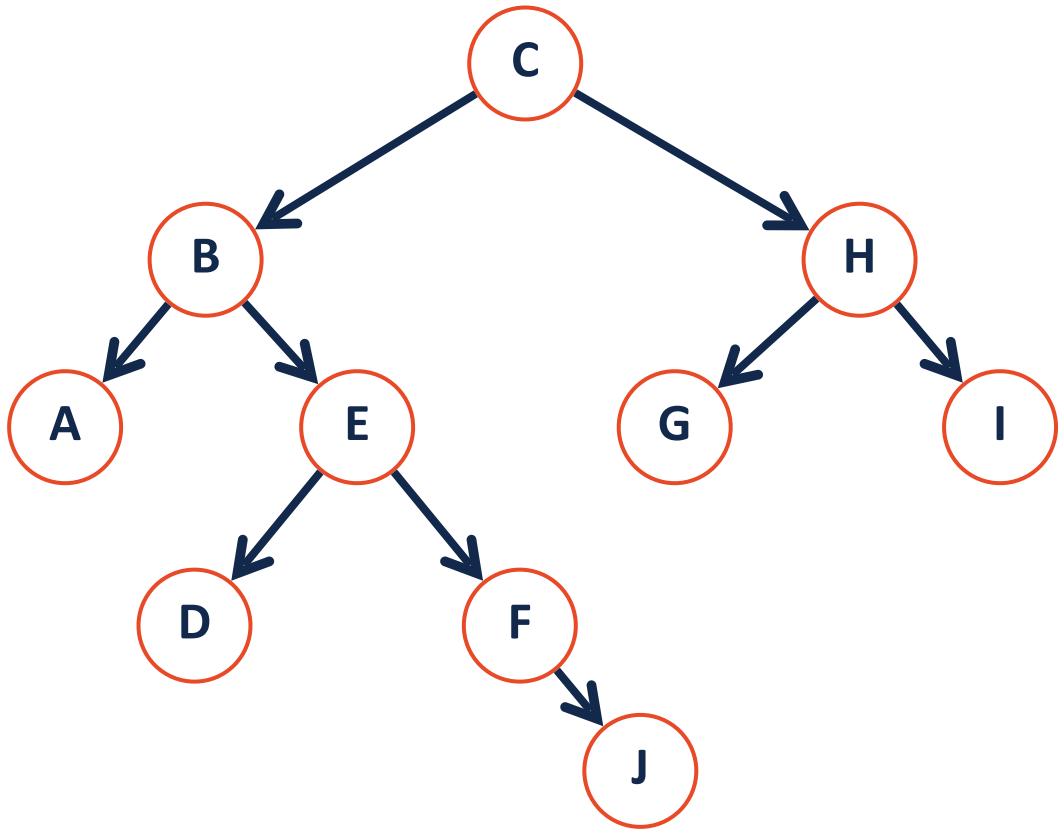


Node: The vertex of a tree

Edge: The connecting path between nodes

Path: A list of the edges (or nodes) traversed to go from node `start` to node `end`

Tree Terminology



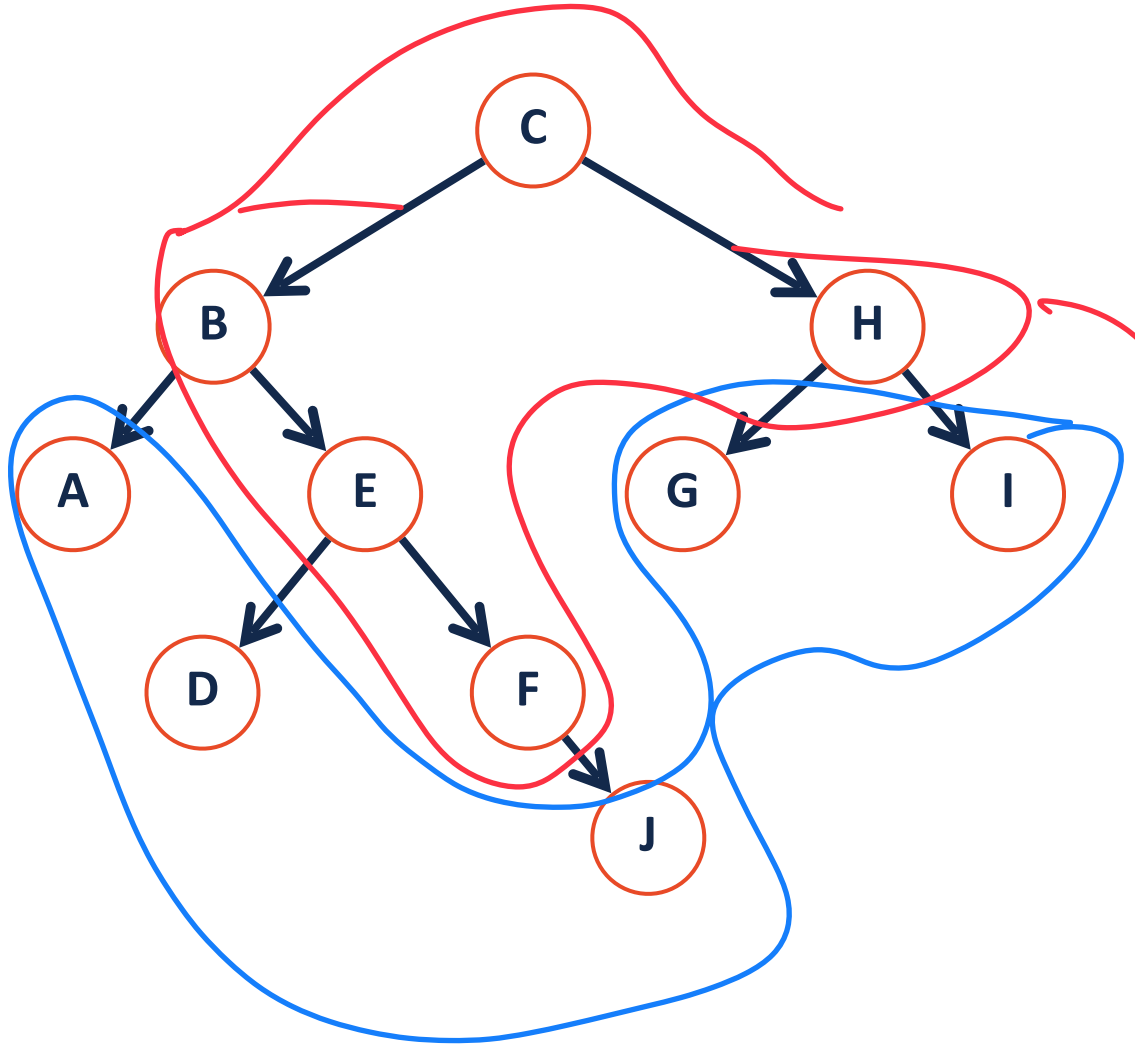
Parent: The precursor node to the current node is the ‘parent’

Child: The nodes linked by the current node are it’s ‘children’

Neighbor: Parent or child

Degree: The number of children for a given node

Tree Terminology

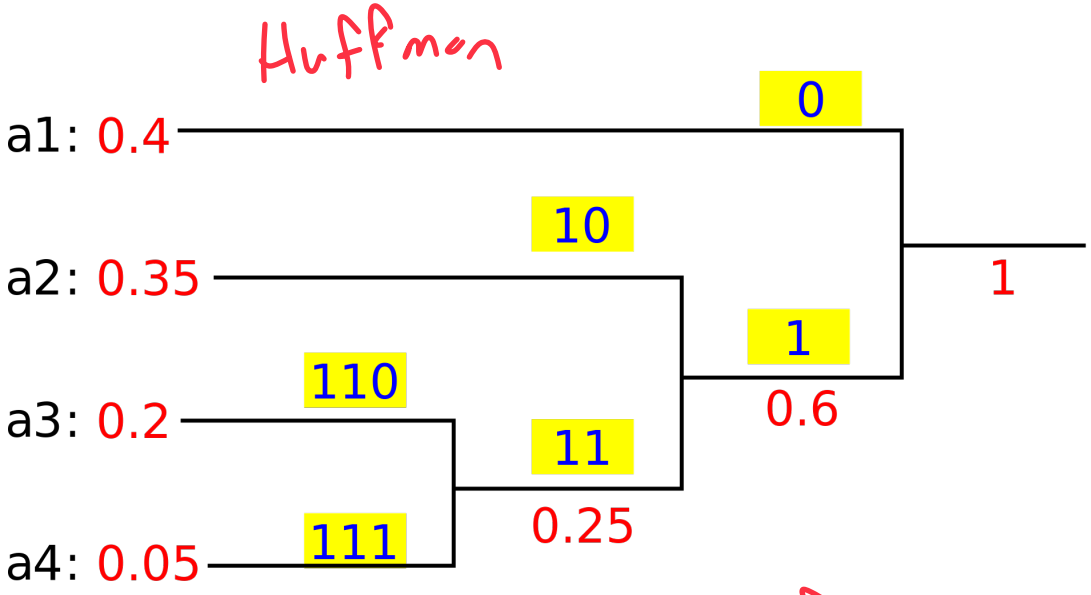


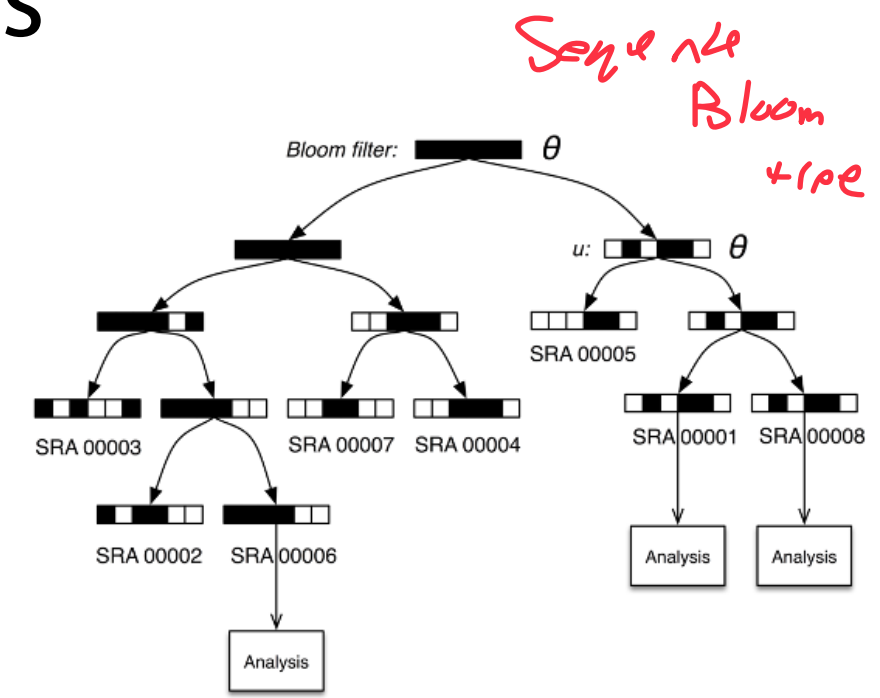
Root: The start of a tree (the only node with no parent).

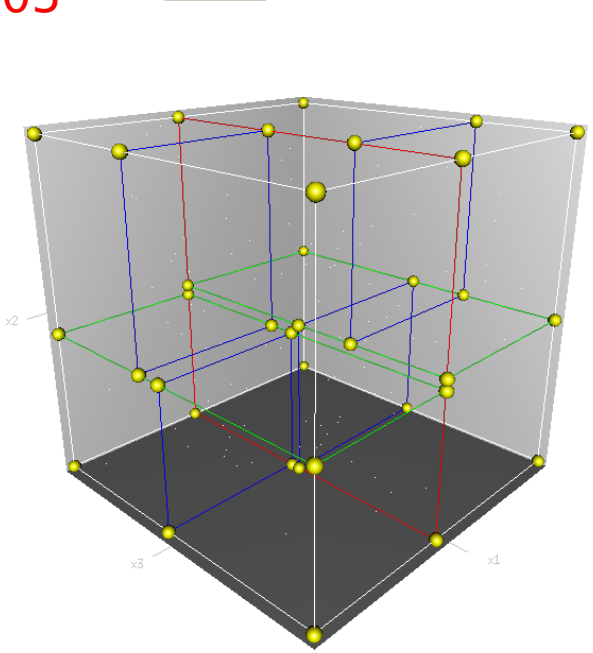
Leaf: The terminating nodes of a tree (have no children)

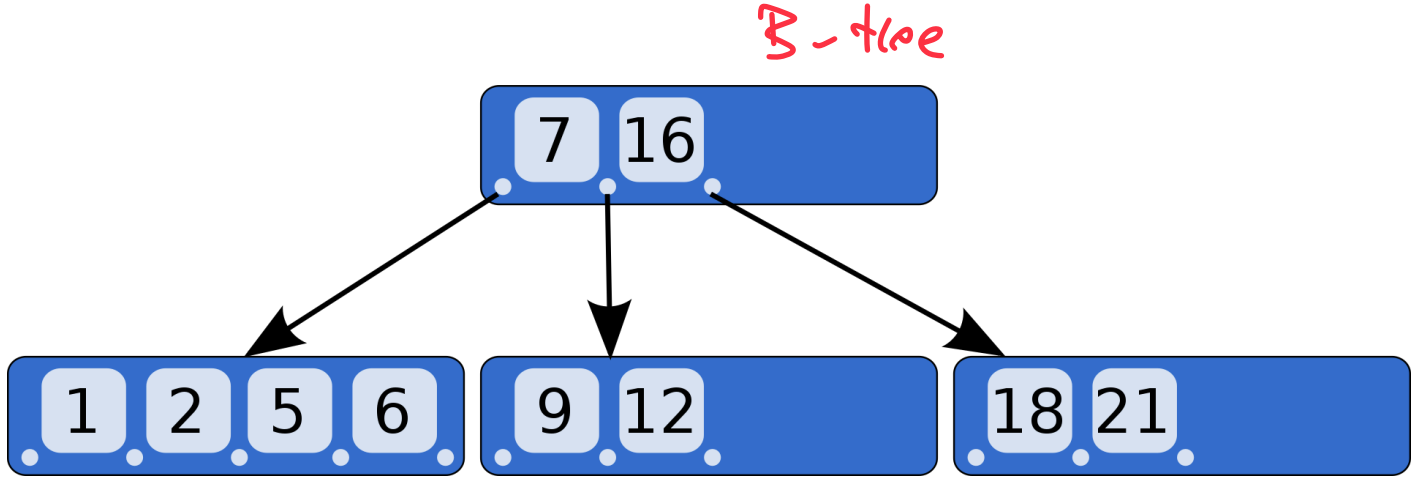
Internal: A node with at least one child

There are many *types* of trees









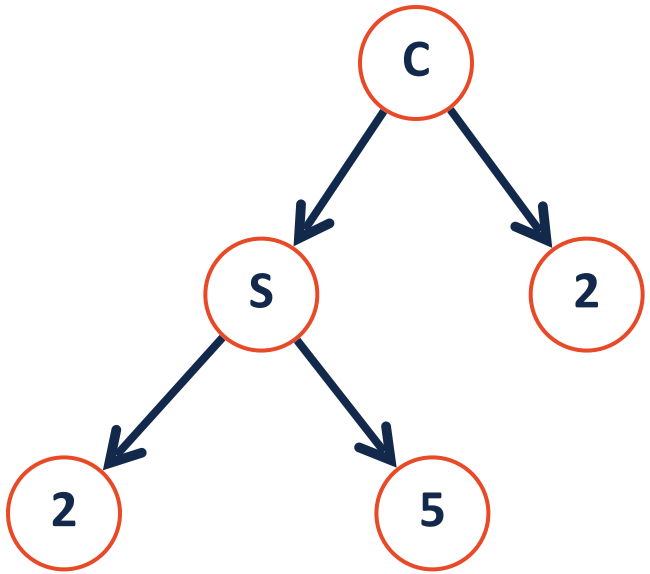
Binary Tree

Recursion

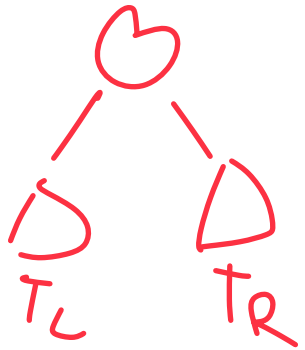
A **binary tree** is a tree T such that:

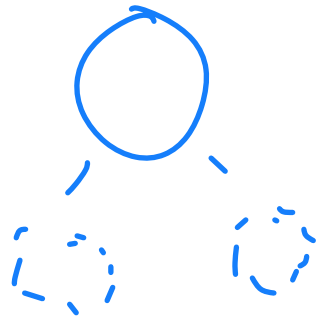
1. $T = \emptyset$

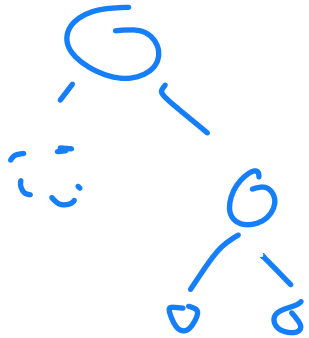
empty tree (zero nodes)



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



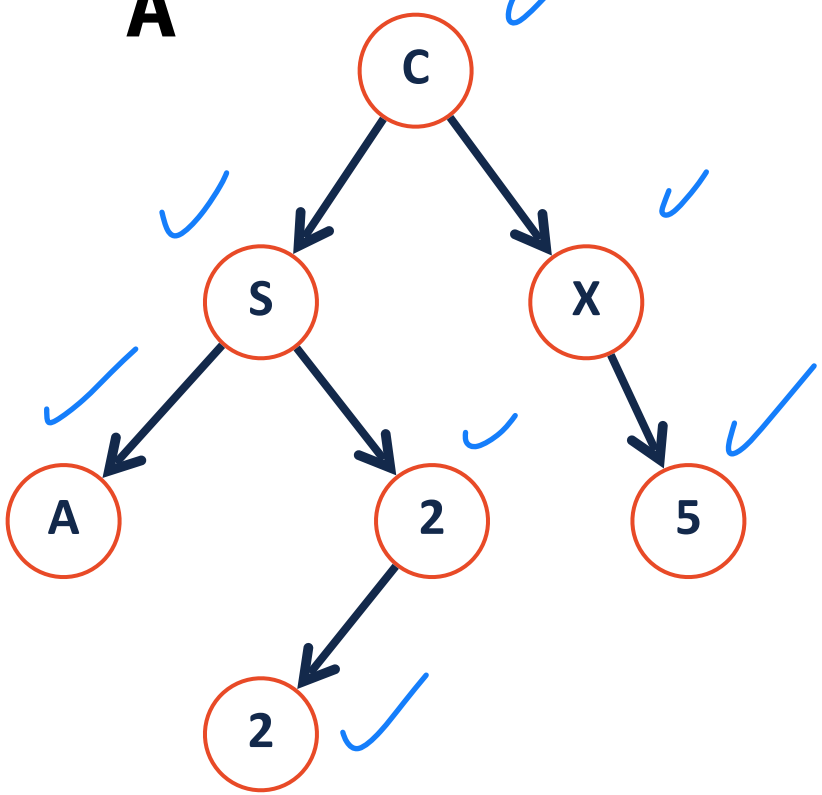




Which of the following are binary trees?

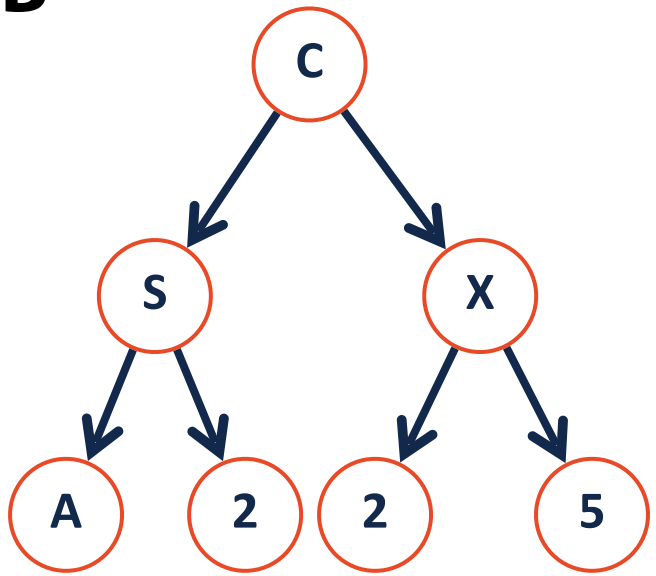
Join Code: 225

A



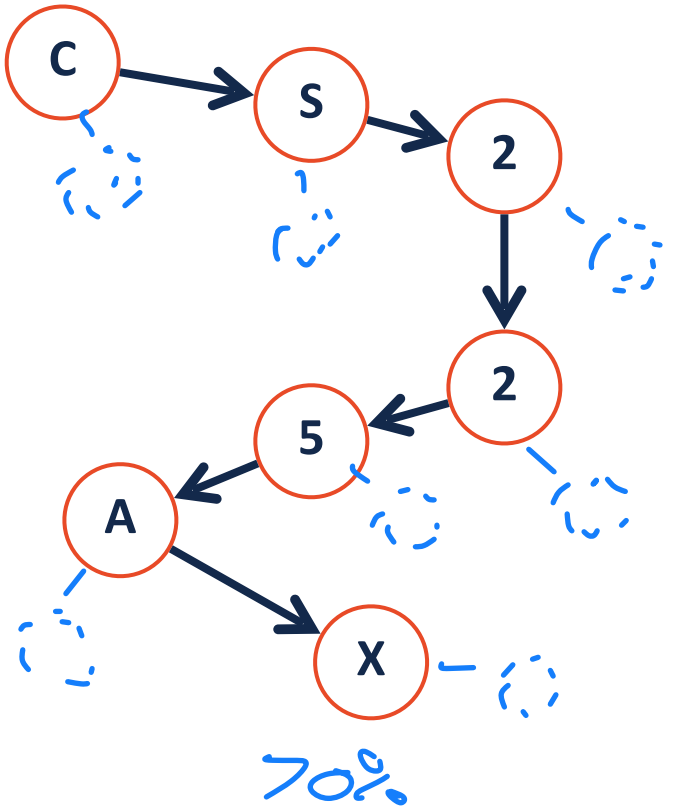
B

A perfect binary tree



100%

C

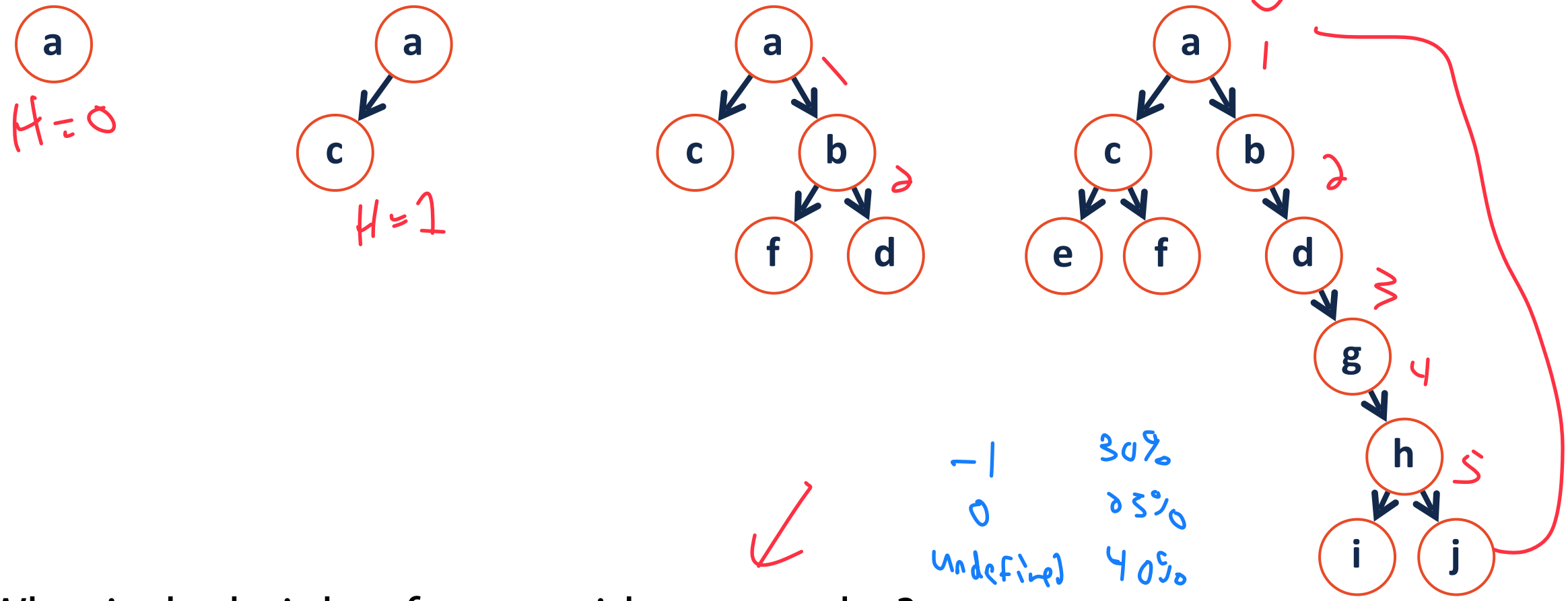


70%

Invisible null children do exist!

Binary Tree Height

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



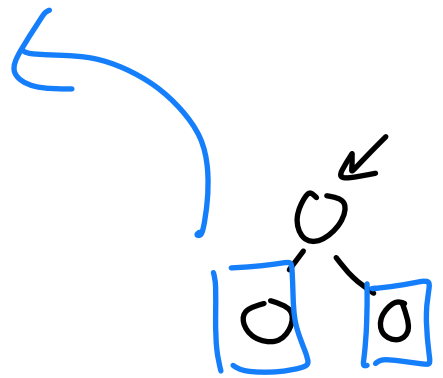
What is the height of a tree with ***zero*** nodes?

Binary Tree Height

`height(T)` = $1 + \max(\text{height}(T_l), \text{height}(T_r))$

Base Case: If 1 node, height = 0

If 0 node, height = -1



Recursive Step:

height(null) = -1

Combining: Max part

Binary Tree Height

$$height(T) = 1 + max(height(T_L), height(T_R))$$

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. full (tree)

2. perfect (tree)

3. complete (tree)

Binary Tree: full

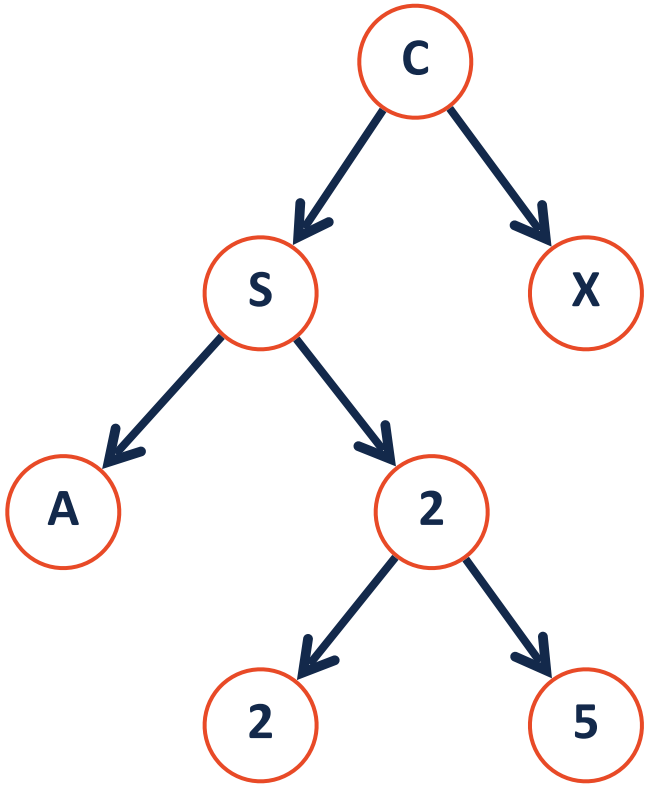
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1. $F = \emptyset$ ∴

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

3. $F = (root, F_L, F_R)$



Binary Tree: full

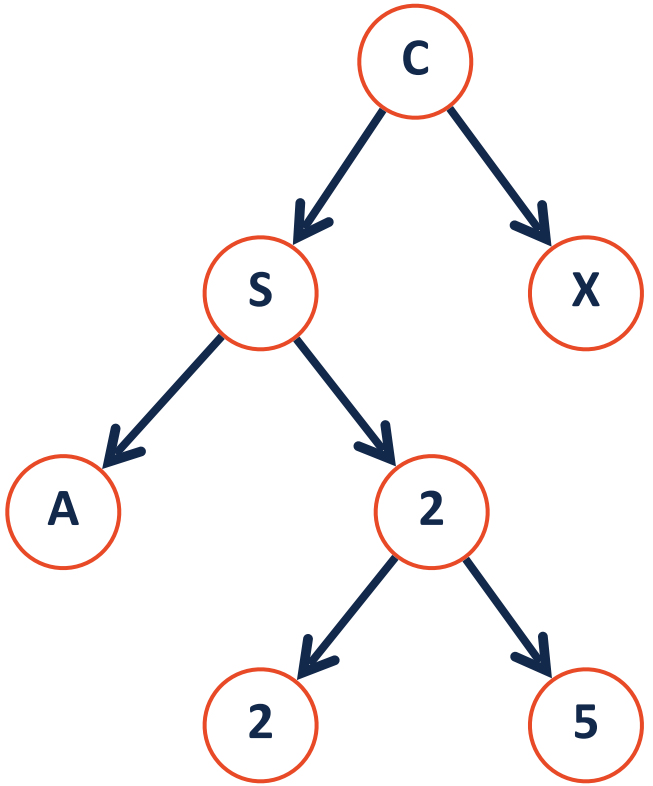
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1. $F = \emptyset$

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

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



Binary Tree: full

Join Code: 225

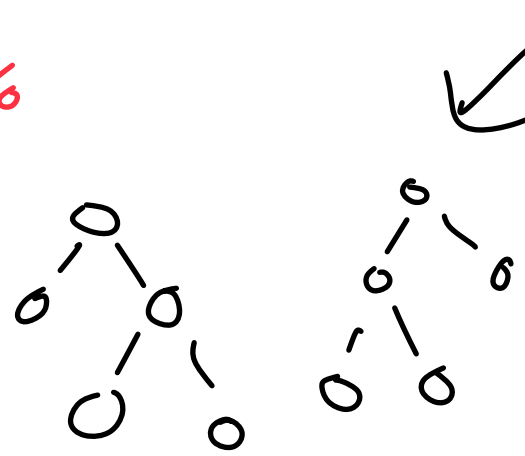
Which of the following are possible sizes (# of nodes) for…

… a full binary tree?

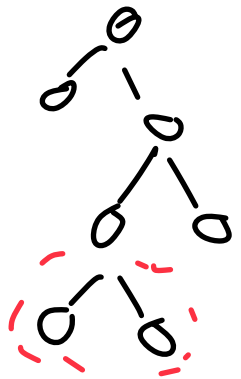
A) 2 Nodes

Not full!

B) 5 Nodes



C) 7 Nodes



D) 8 Nodes

Trivia: Cannot have even #s on full tree!

Binary Tree: perfect

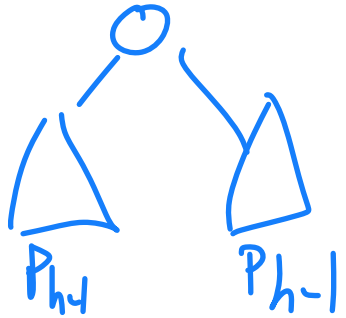
(max capacity for height h)

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

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

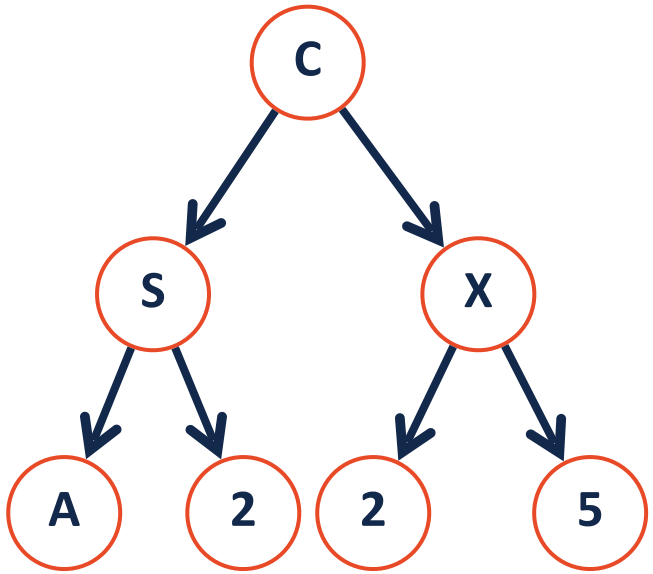
A tree **P** is **perfect** if and only if:

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



2. $P_0 = (root, \phi, \phi)$

3. $P_{-1} = \phi$



Binary Tree: perfect

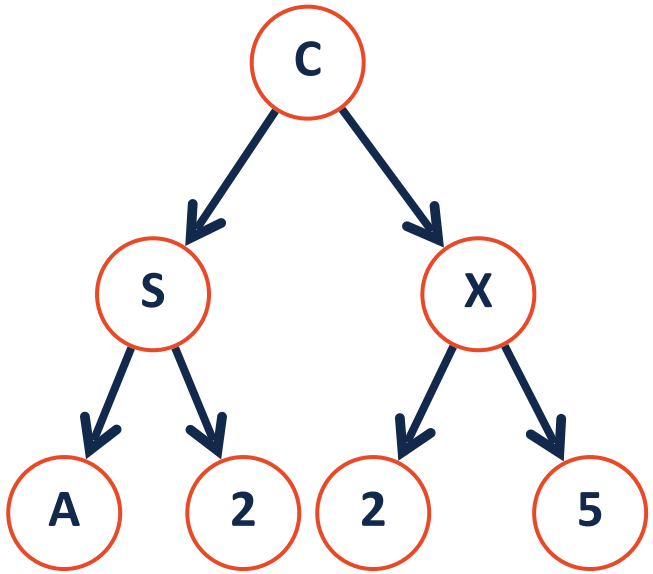
A **perfect tree** is a binary tree where…

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

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

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

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



Binary Tree: perfect

Join Code: 225

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

… a perfect binary tree?

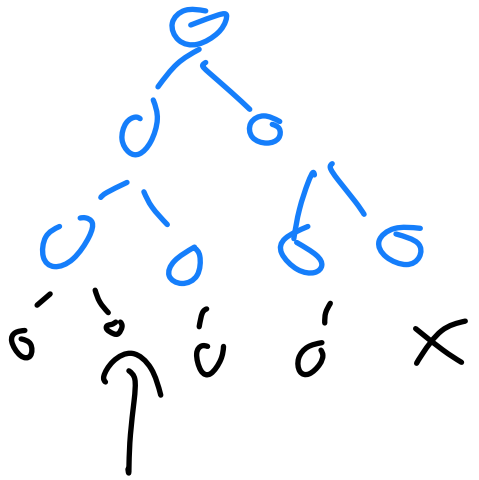
# of nodes in perfect tree of height h: $2^{(h+1)} - 1$

A) 2 Nodes

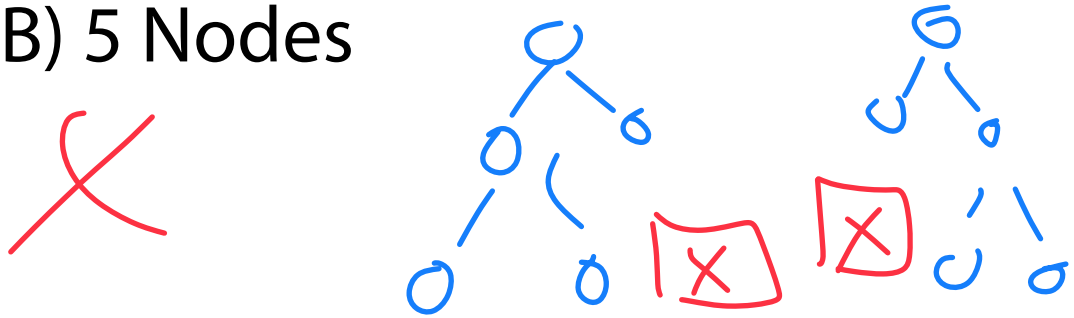
h = 0 $2^1 - 1 = 1$ ✓

h = 1 $2^2 - 1 = 3$ ✓

C) 7 Nodes



B) 5 Nodes



D) 8 Nodes

✗ add +1 breaks perfect

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)

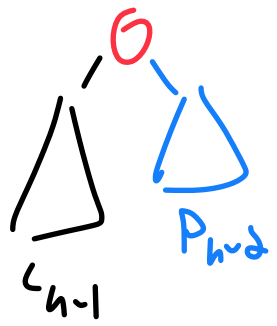
A tree **C** is **complete** if and only if:

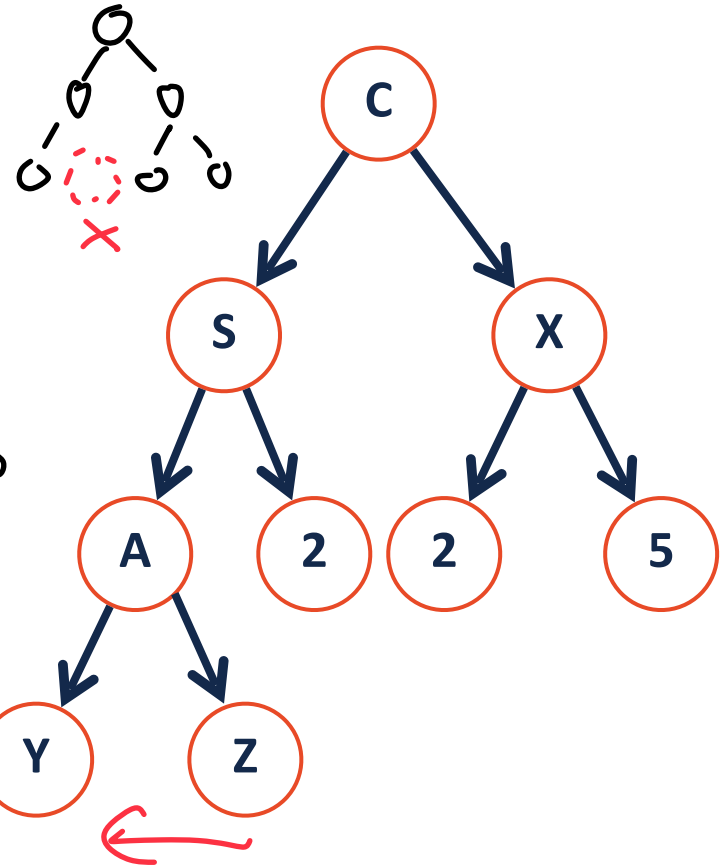
1. $C_h = (root, C_{h-1}, P_{h-2})$

   Left subtree must be larger than right o/ ....

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

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





Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

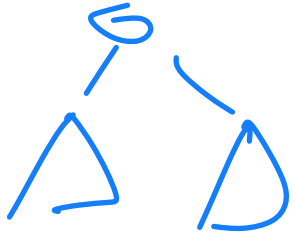
The last level contains at least one node (and is pushed to left)

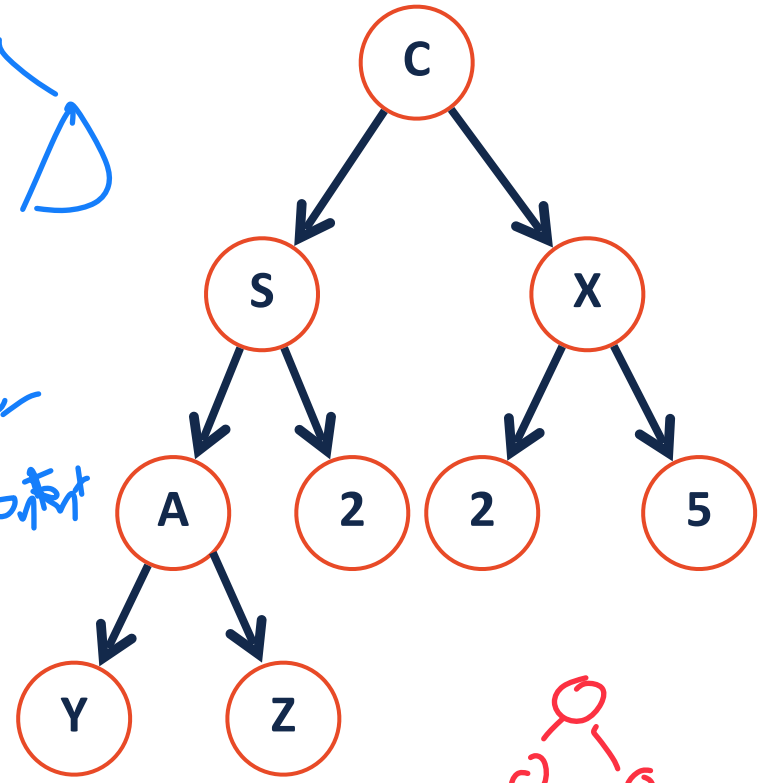
A tree **C** is **complete** if and only if:

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

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

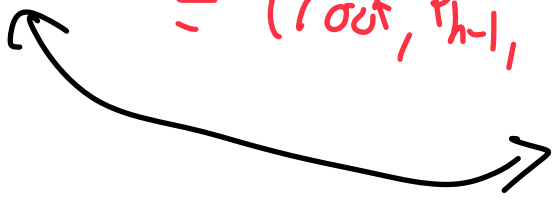
3. $C_{-1} = \emptyset$





Can be helpful for future theory

$\equiv (root, P_{h-1}, P_{h-2})$



Both a complete & perfect tree

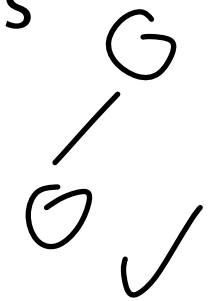
Binary Tree: ~~perfect~~ complete

Join Code: 225

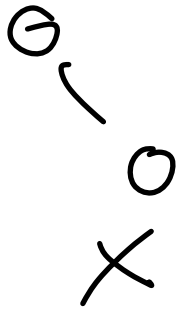
Which of the following are possible sizes (# of nodes) for…

… a ~~perfect~~ complete binary tree?

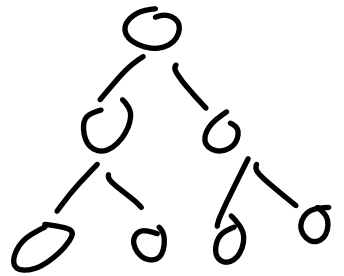
A) 2 Nodes



B) 5 Nodes



C) 7 Nodes



Is both complete & perfect

D) 8 Nodes

All perfect trees are complete, not all complete trees are perfect

Binary Tree

Why do we care?

1. Terminology instantly defines a particular tree structure

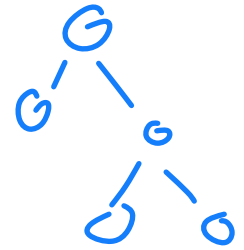
2. Understanding how to think ‘recursively’ is very important.

Binary Tree: Thinking with Types

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

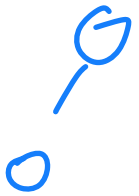
No!

Counter example:



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

No!



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

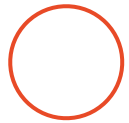
Theorem: If there are **n** objects in our representation of a binary tree, then there are **n+1** NULL pointers.

Base Case:

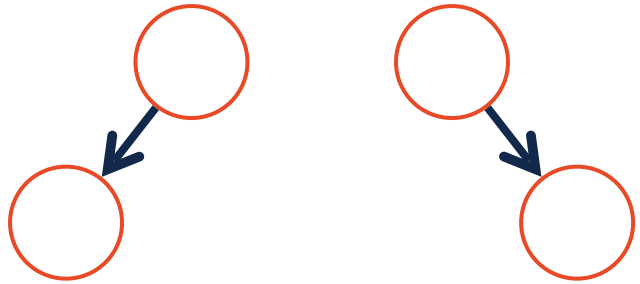
Let F(n) be the max number of NULL pointers in a tree of n nodes

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

```
 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 };
```

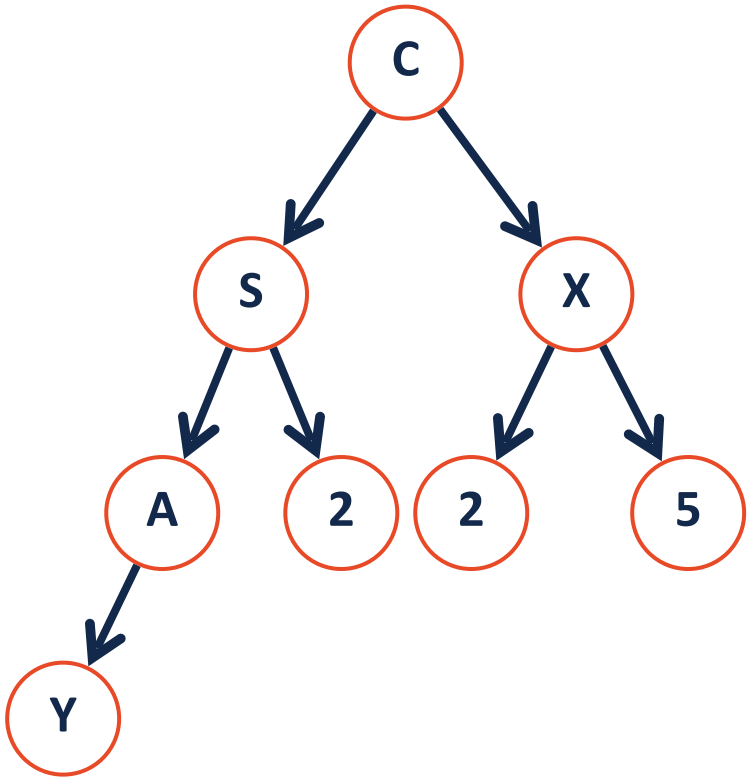
List.h

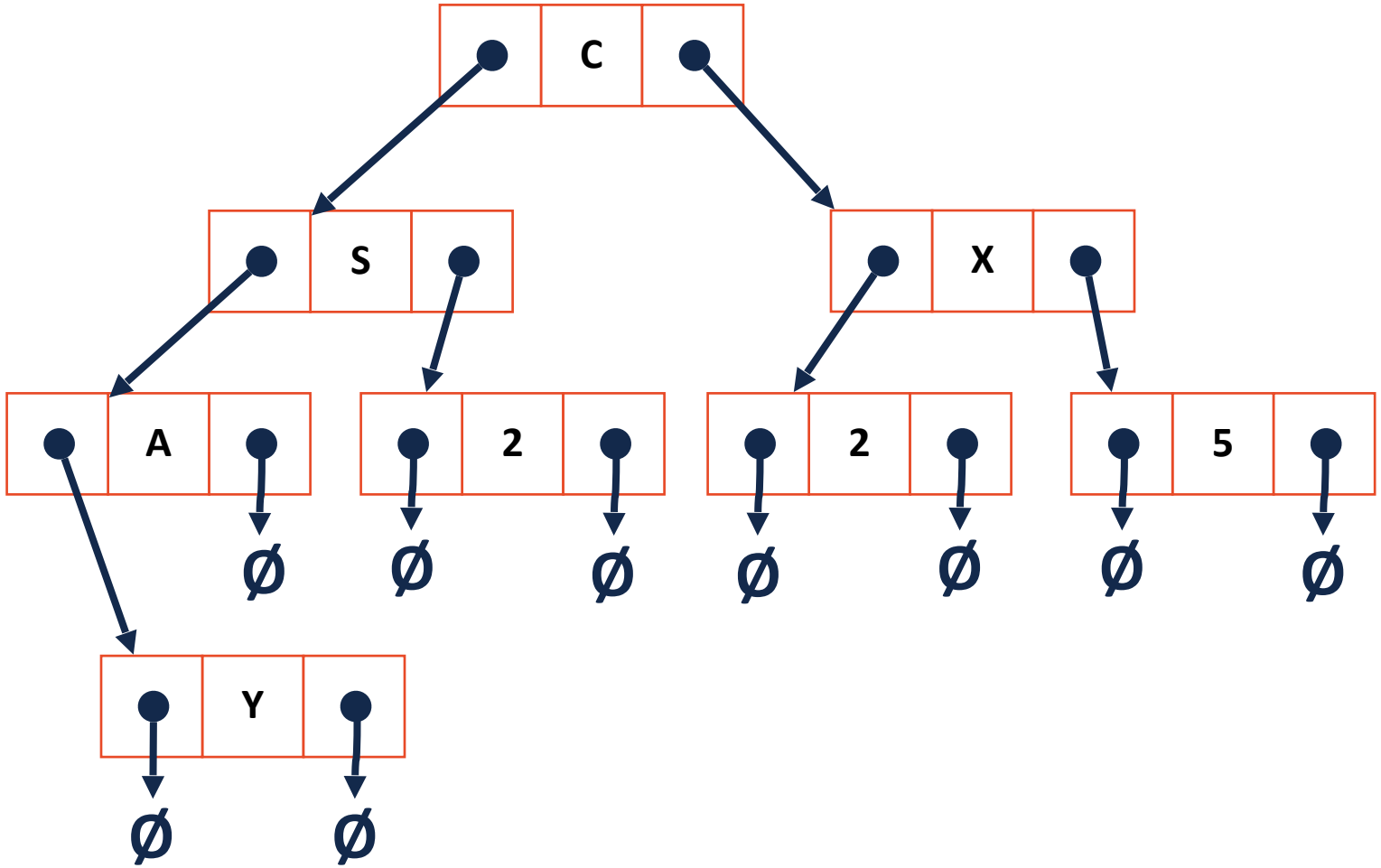
```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
6      /* ... */
7    private:
8      class ListNode {
9        T & data;
10
11       ListNode * next;
12
13
14
15       ListNode(T & data) :
16        data(data), next(NULL) { }
17     };
18
19
20
21     ListNode *head_;
22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

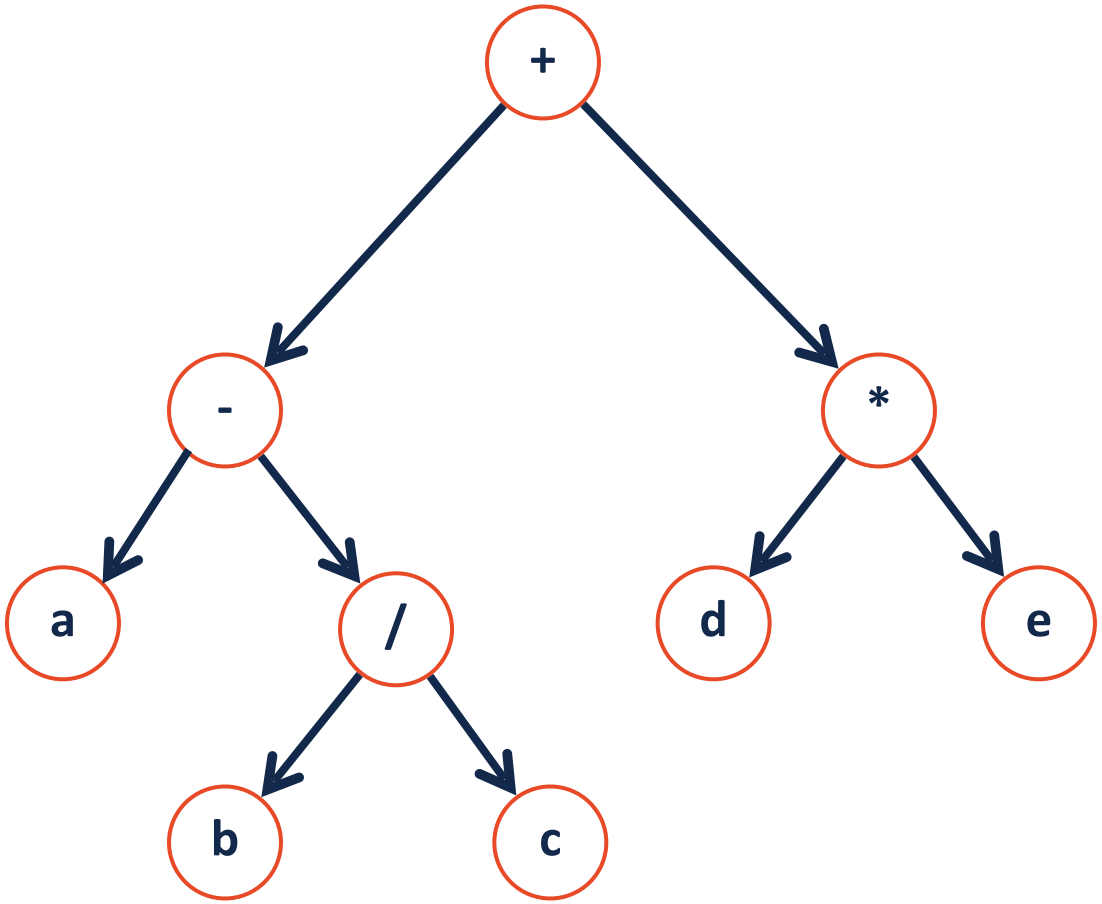
Visualizing trees



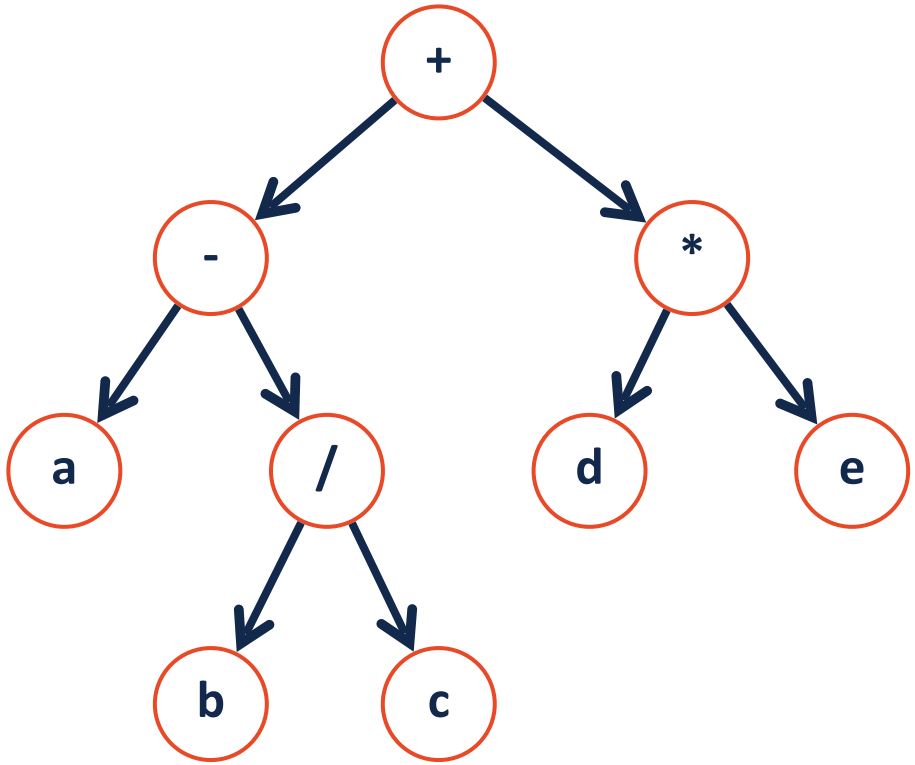


Tree Traversal

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)
{









}
```