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

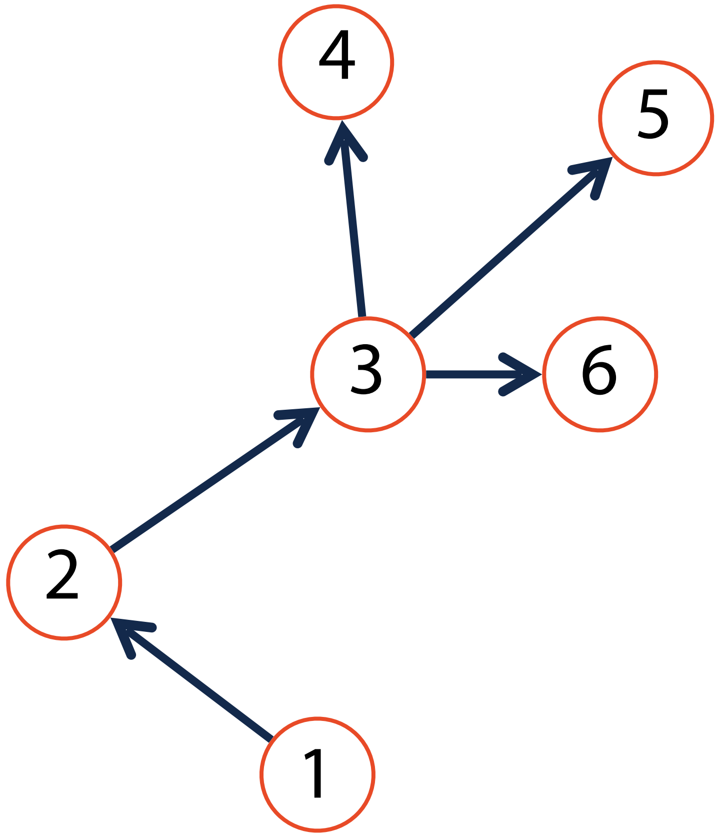
Trees

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:

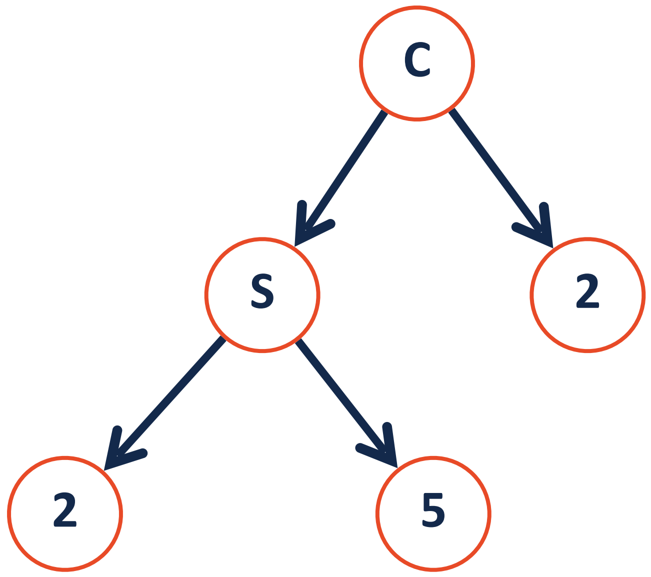
1) Acyclic — No path from node to itself

2) Rooted — A specific node is labeled root



Binary Tree

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

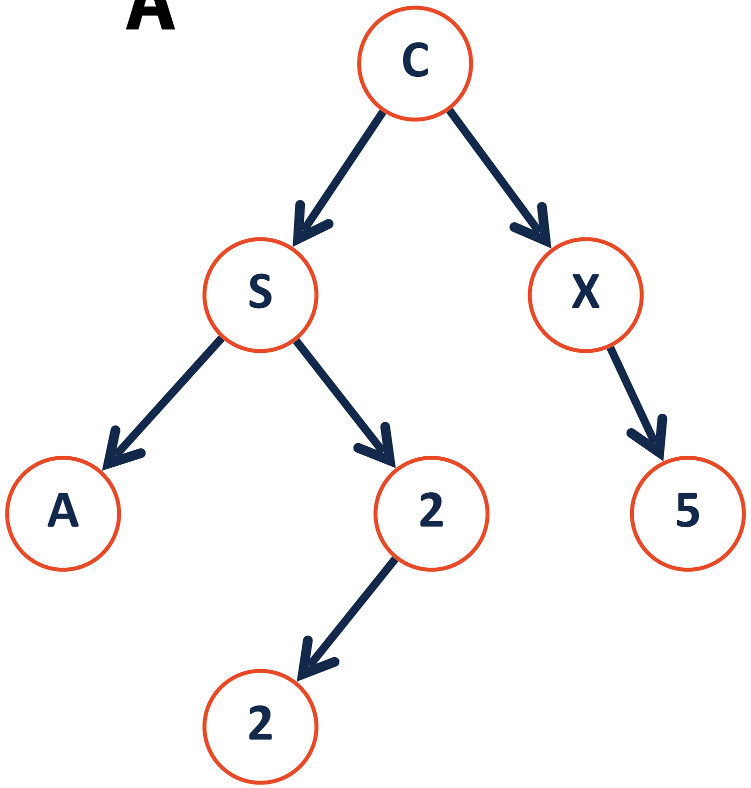


1. $T = \emptyset$

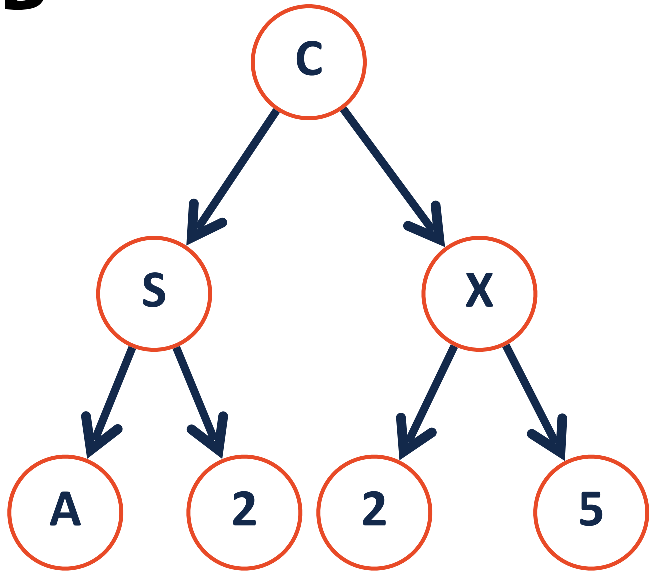
2. $T = (data, T_L, T_R)$

Which of the following are binary trees?

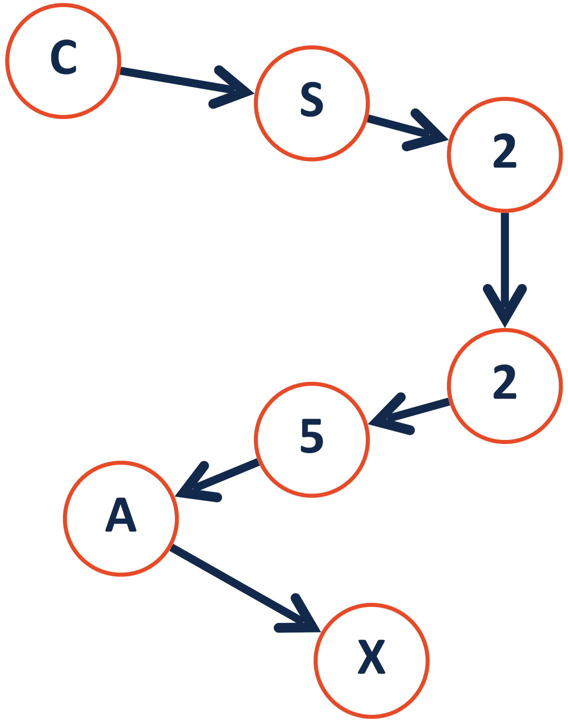
A



B



C



Binary Tree

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

1.

2.

3.

Binary Tree: full

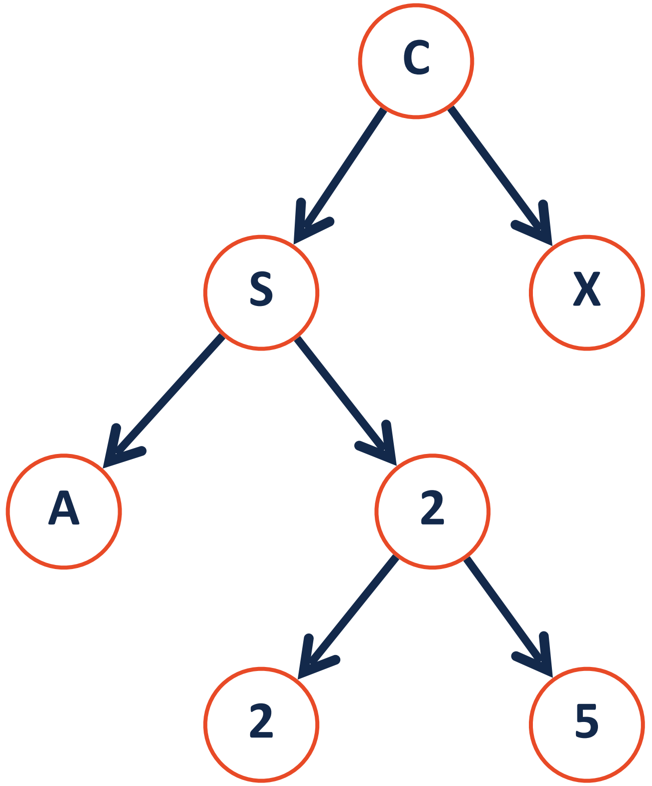
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1.

2.

3.



Binary Tree: perfect

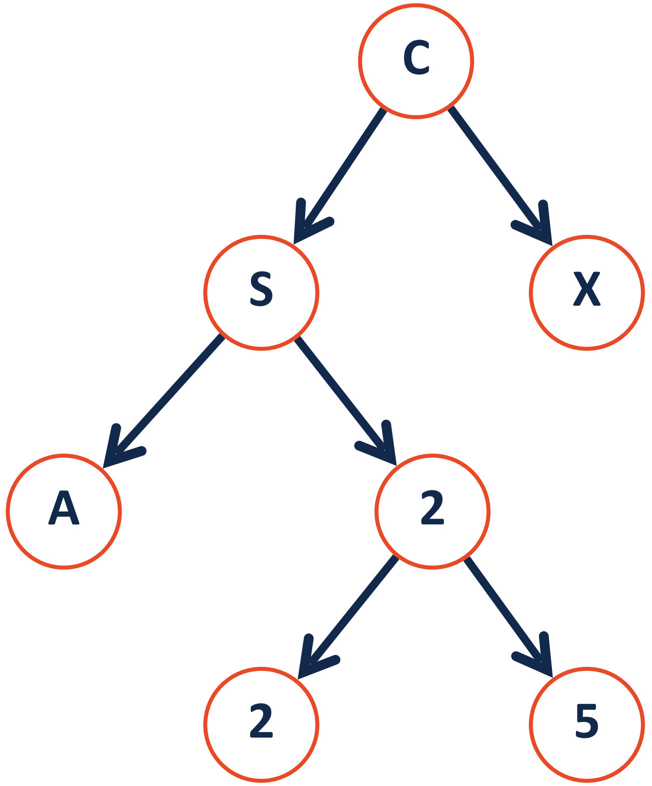
A **perfect tree** is a binary tree where…

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: complete

A **complete tree** is a B.T. where…

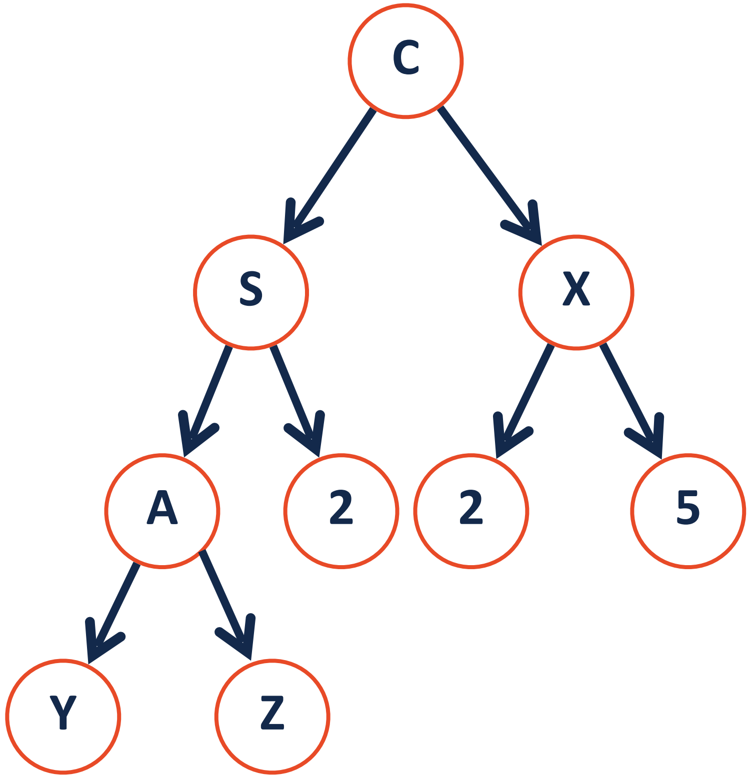
All levels are completely filled except the last (which is pushed to left)

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

1.

2.

3.



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.

Induction Step:

Tree ADT

BinaryTree.h

```
#pragma once

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

  private:
















};
```

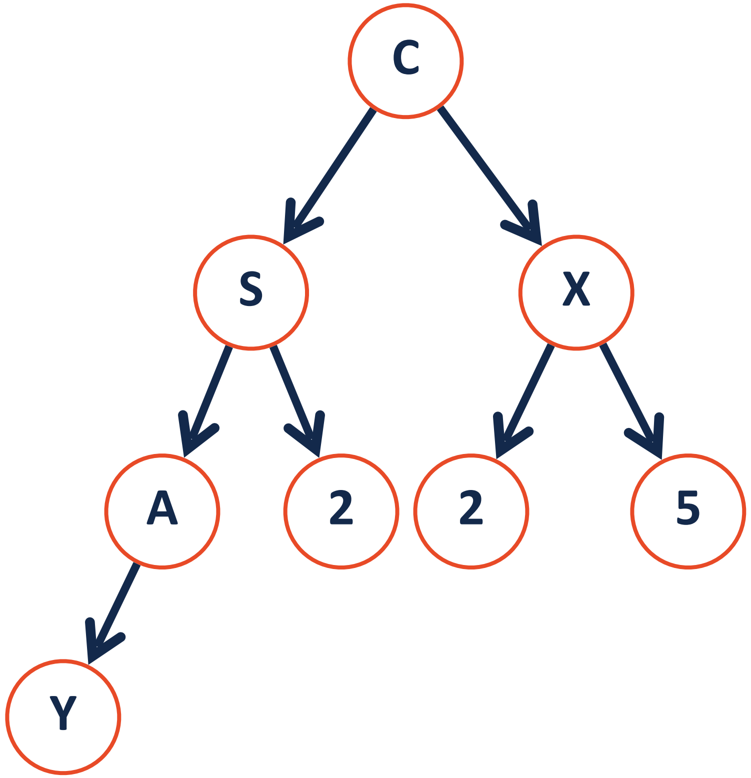
List.h

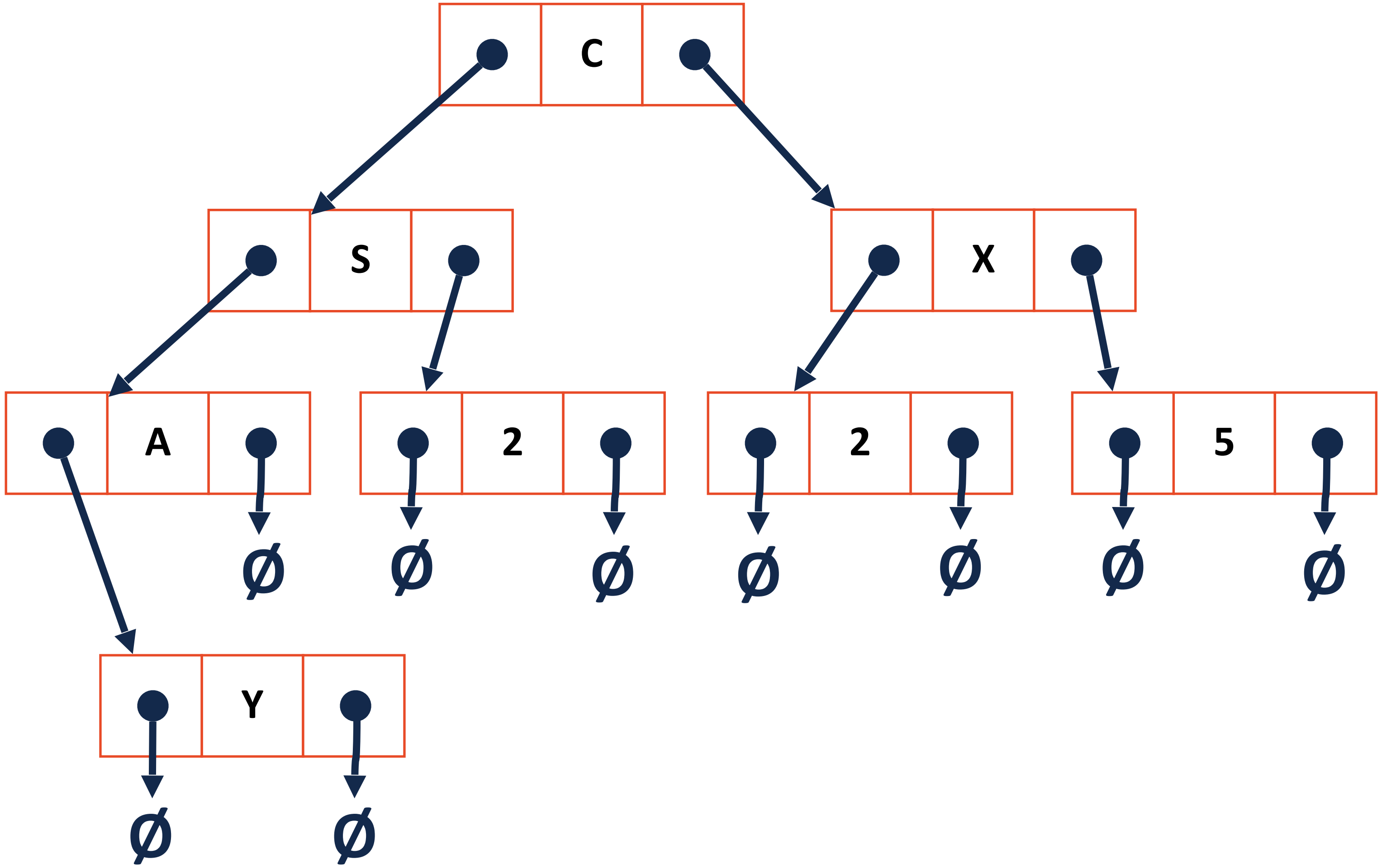
```
#pragma once

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_;
    /* ... */
};
```

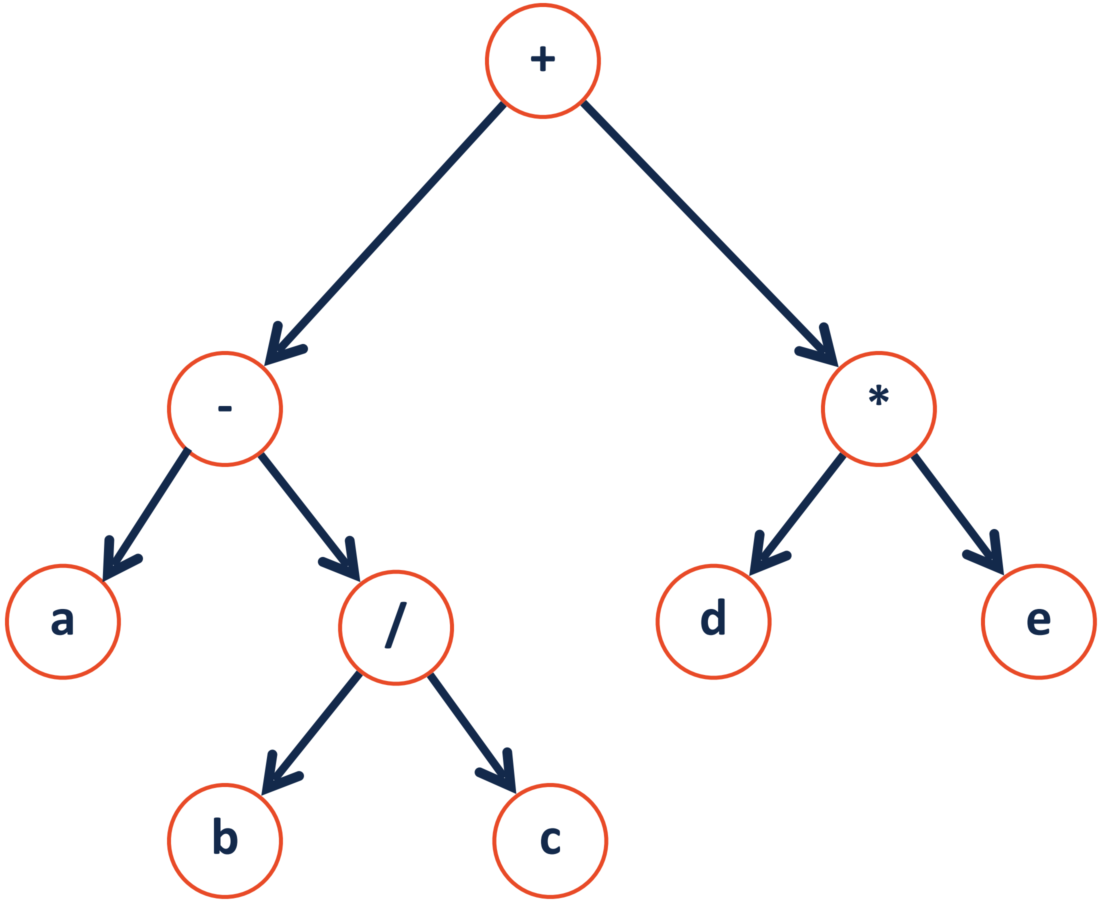
Visualizing trees



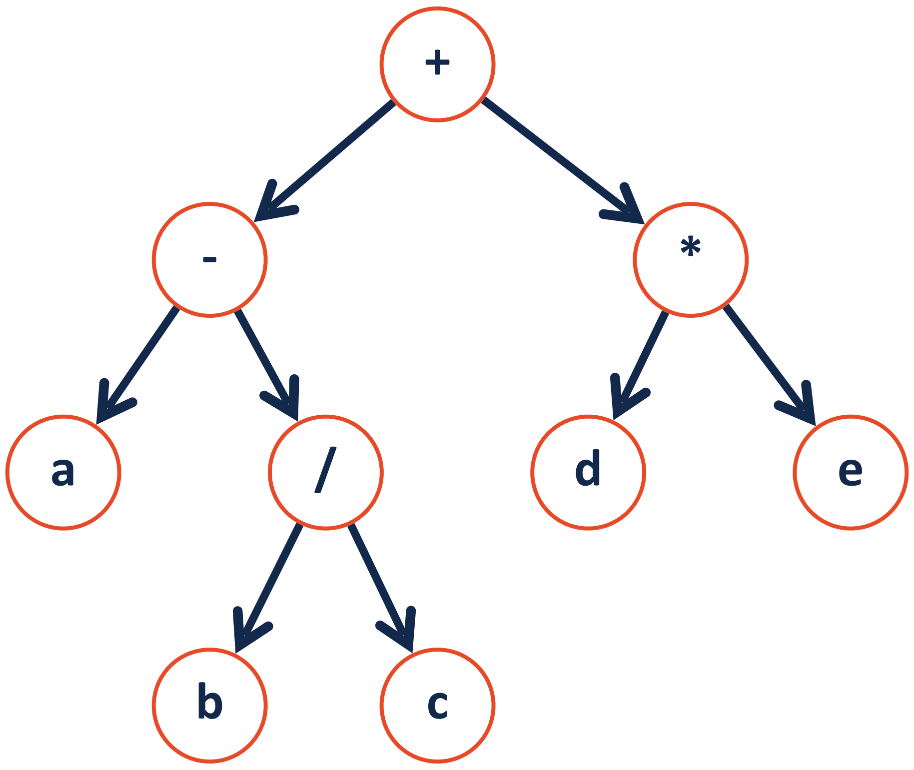


Tree Traversal

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

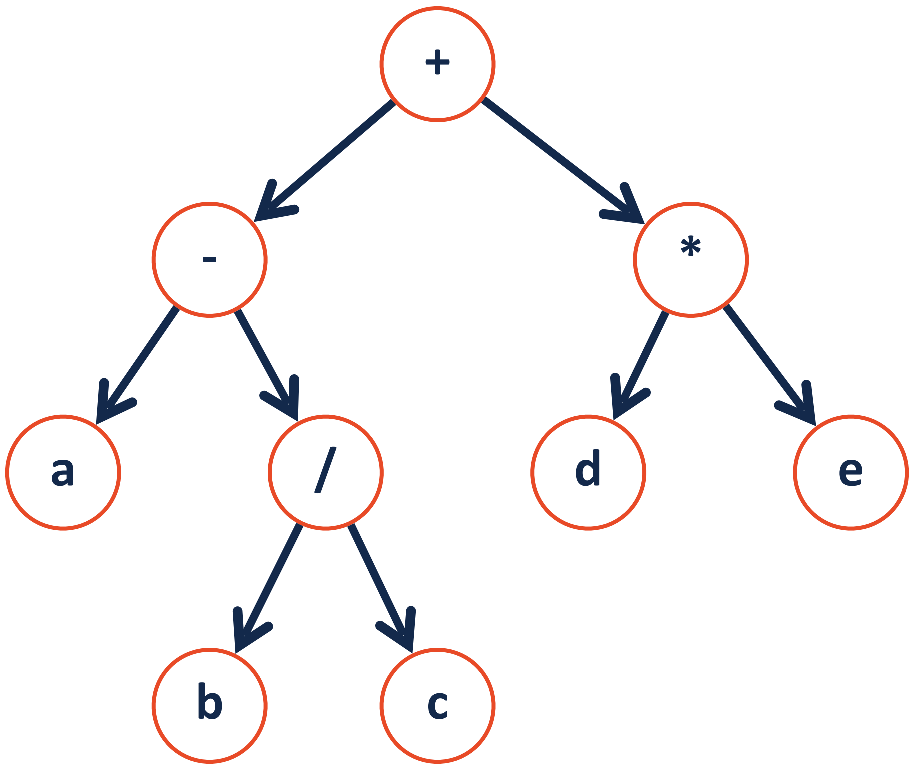


Traversals



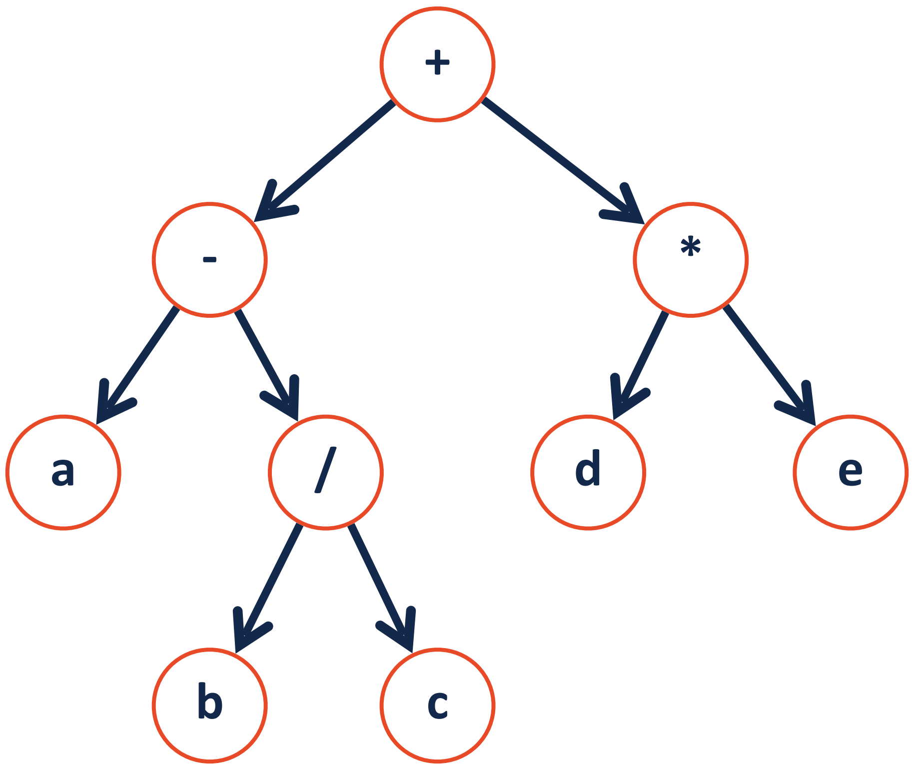

```
template<class T>
void BinaryTree<T>::_____Order(TreeNode * root)
{
















}
```

Traversals



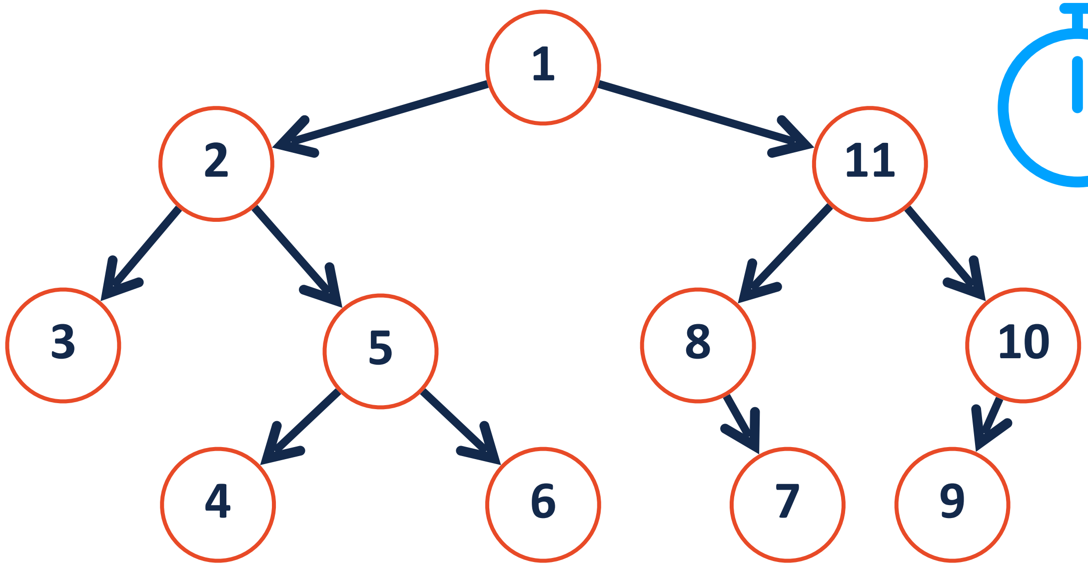

```
template<class T>
void BinaryTree<T>::_____Order(TreeNode * root)
{

  if (root) {

      ______________________;

      _____Order(root->left);

      ______________________;

      _____Order(root->right);

      ______________________;

  }



}
```

Traversals




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

  if (root) {

      ______________________;

      _____Order(root->left);

      ______________________;

      _____Order(root->right);

      ______________________;

  }



}
```

Tree Traversals



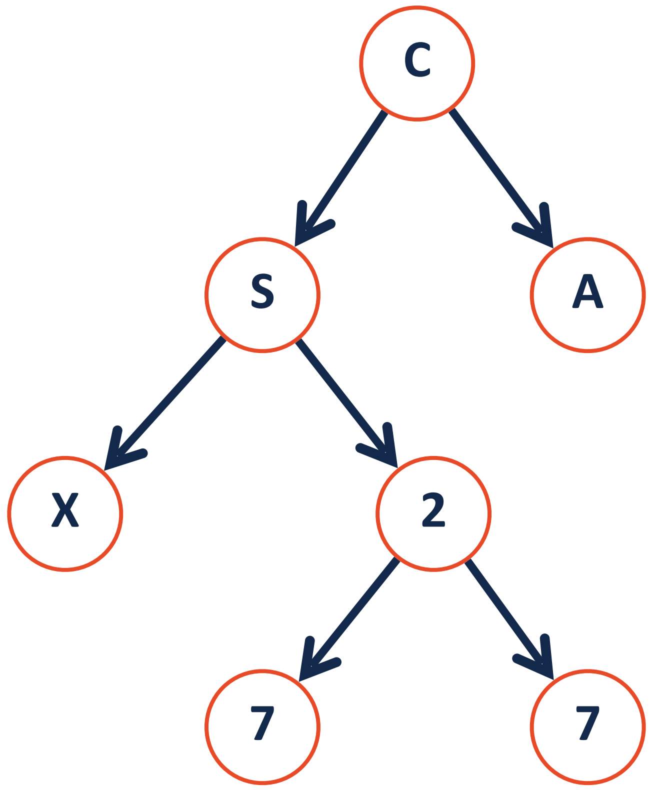
Pre-order:

In-order:

Post-order:

Tree Traversals

Pre-order: Ideal for copying trees

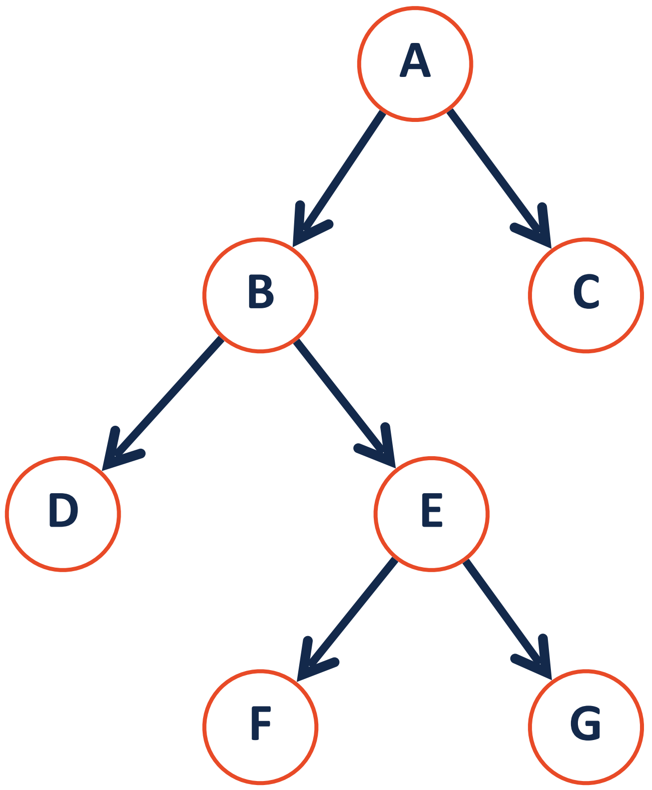


Post-order: Ideal for deleting trees

Traversal vs Search

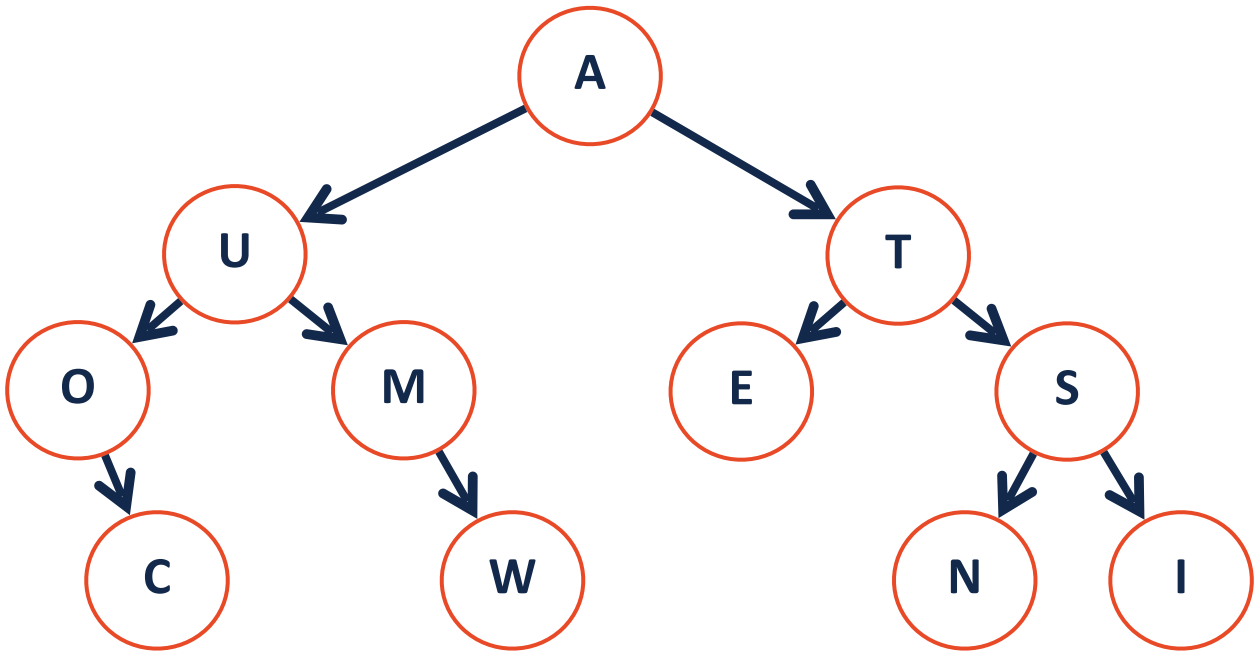
Traversal

Search



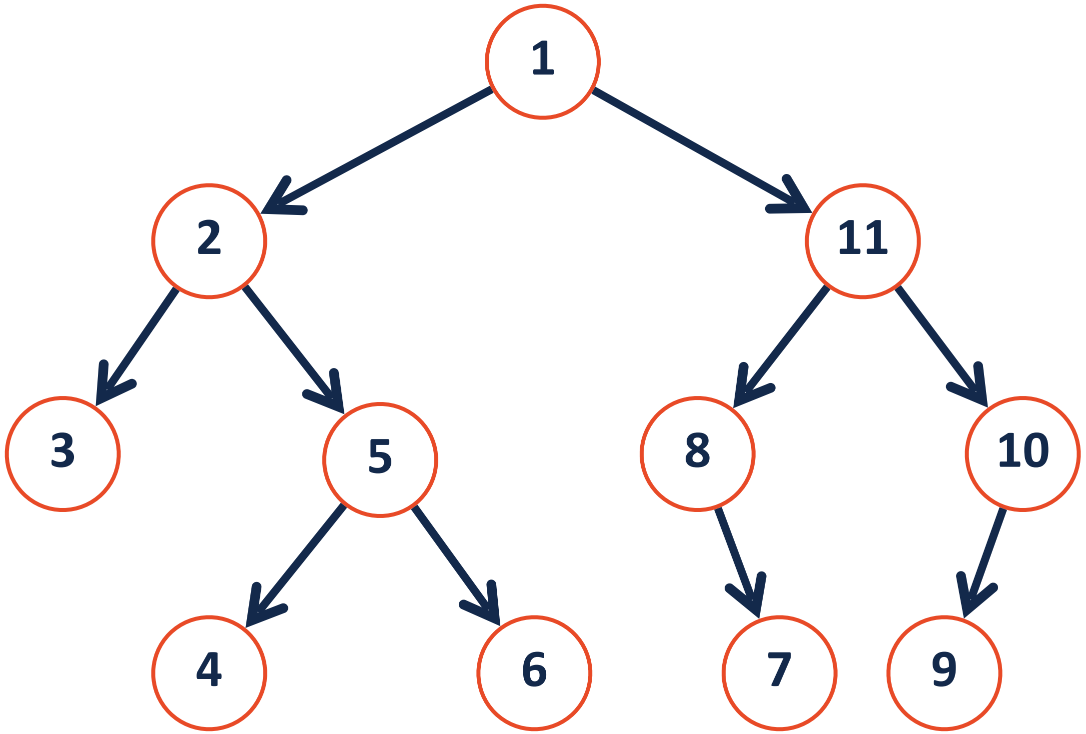
Tree Search

There are two main approaches to searching a binary tree:



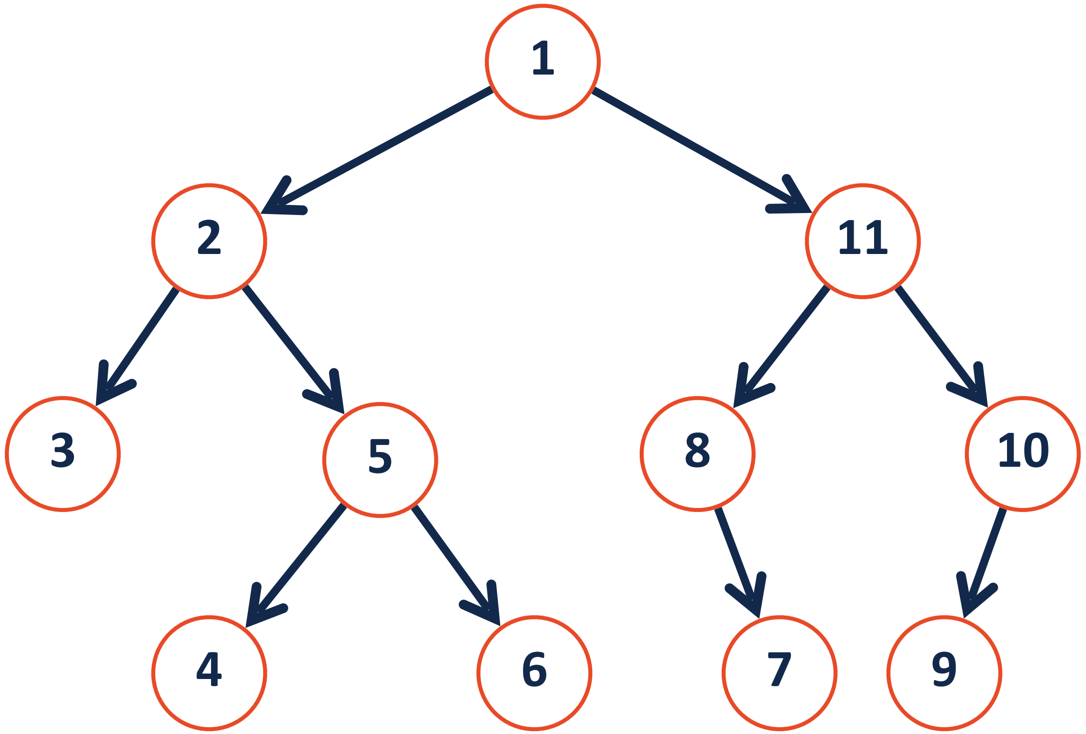
Depth First Search

Explore as far along one path as possible before backtracking

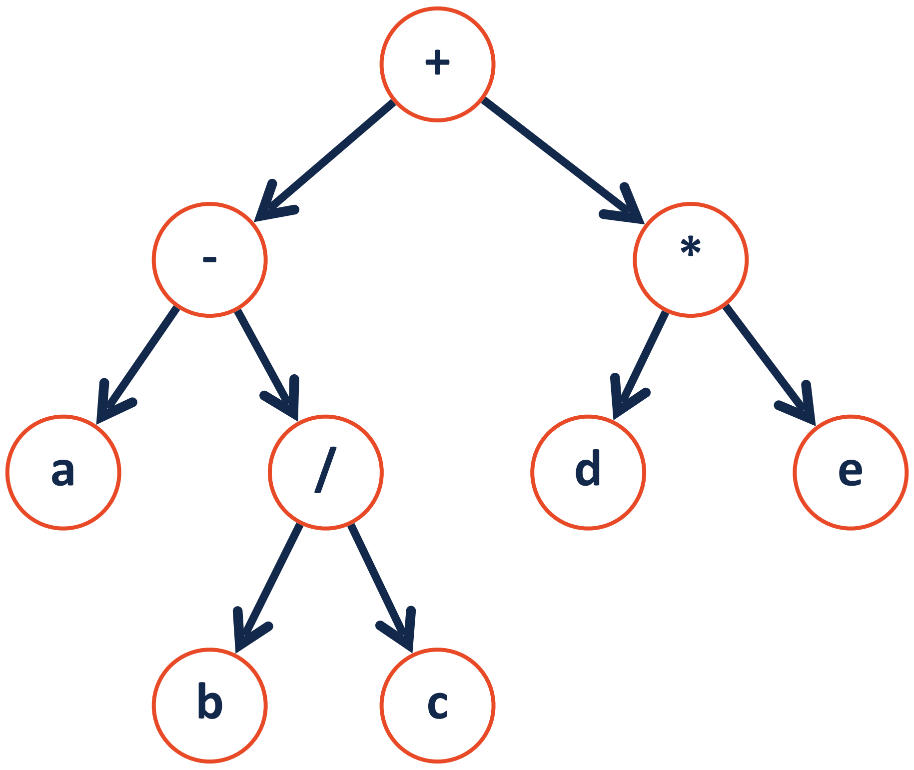


Breadth First Search

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?