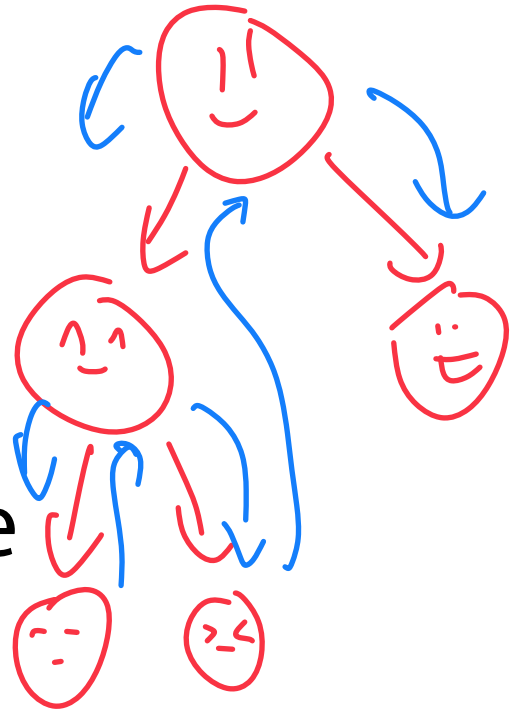
Data Structures

Tree Traversal

CS 225
Brad Solomon

September 18, 2023



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 8 points

Learning Objectives

Discuss the tree ADT

Explore tree implementation details

Tree ADT

Insert — add item to tree

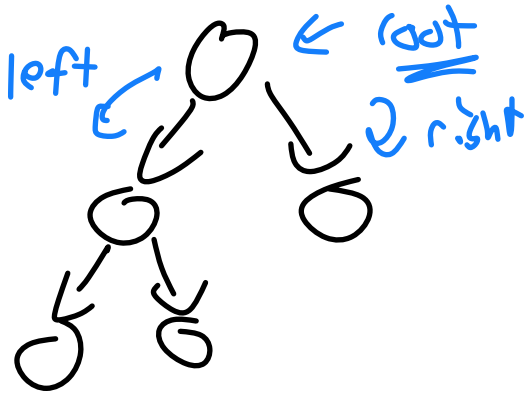
Remove — specific object/node/value

Traverse — iterate over a tree (How to visit all nodes)

Find — search for value/object/node

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 };
```

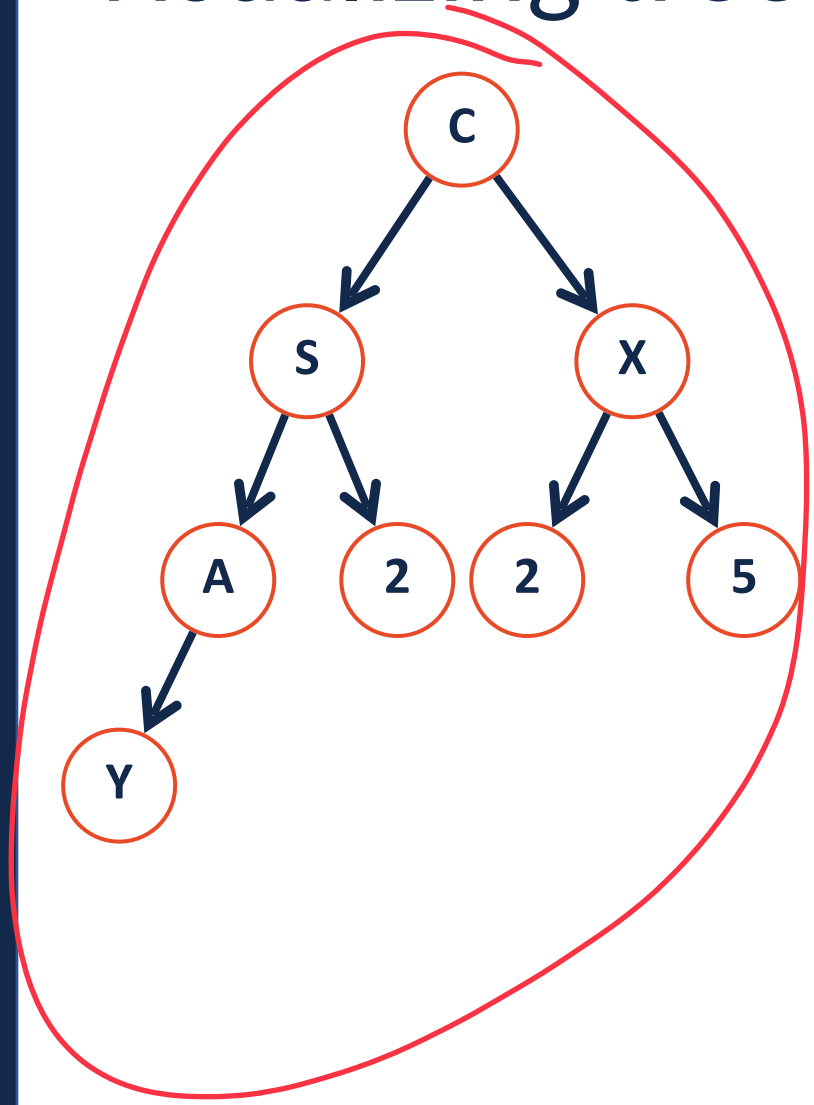
List.h

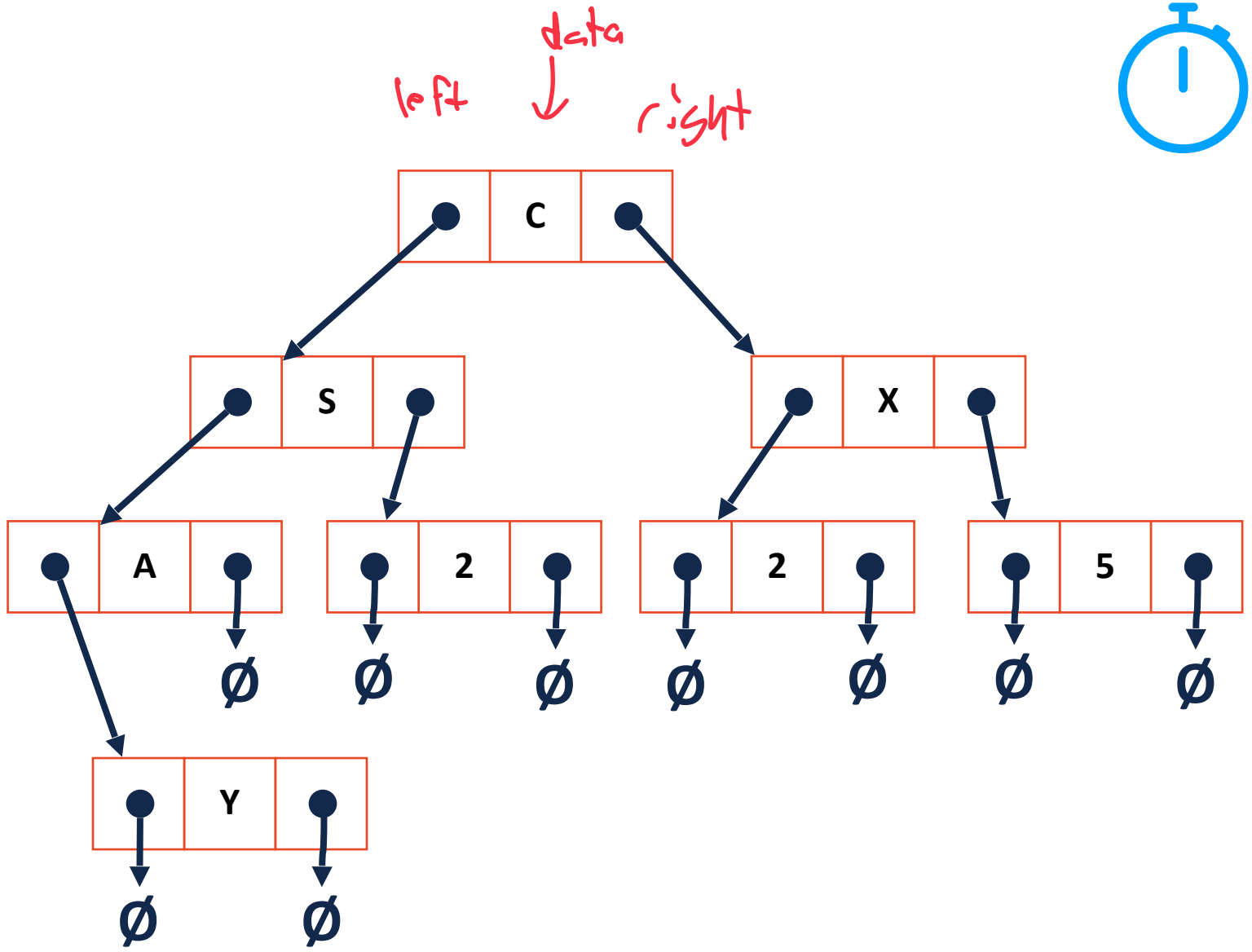
```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
6      /* ... */
7    private:
8      class ListNode {
9        T & data;
10
11       ListNode * next;
12
13
14
15       ListNode(T & data) :
16        data(data), next(NULL) { }
17     };
18
19
20
21     ListNode *head_;
22     /* ... */
23 };
```

Tree.h

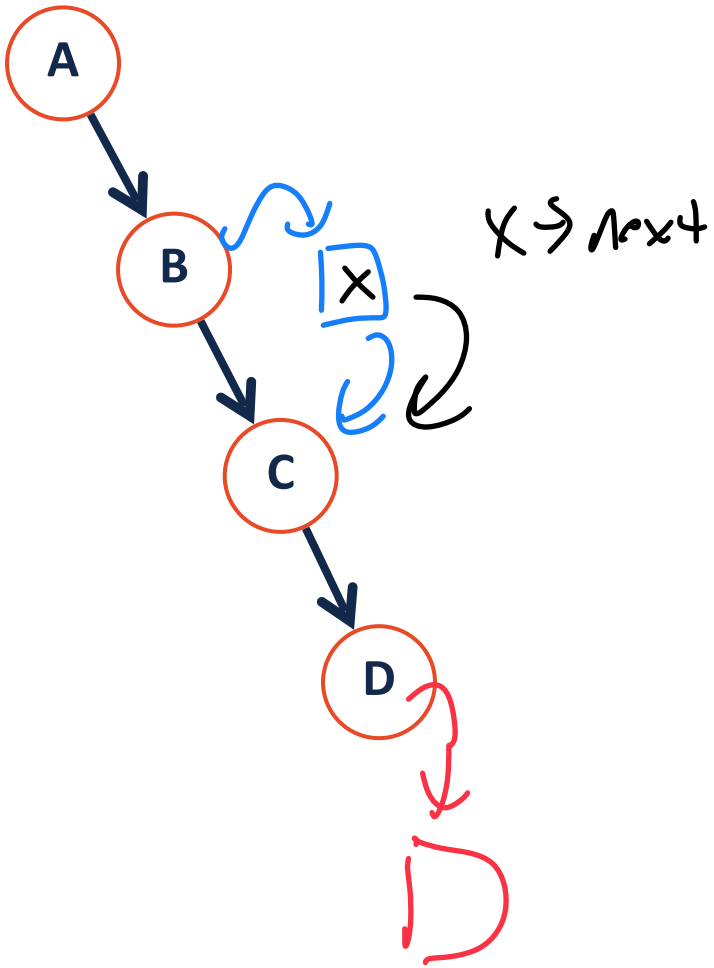
```
1  #pragma once
2
3  template <typename T>
4  class BinaryTree {
5    public:
6      /* ... */
7    private:
8      class TreeNode {
9        T & data;
10
11       TreeNode * left;
12
13       TreeNode * right;
14
15       TreeNode(T & data) :
16        data(data), left(NULL),
17 right(NULL) { }
18
19     };
20
21     TreeNode *root_;
22     /* ... */
23 };
```

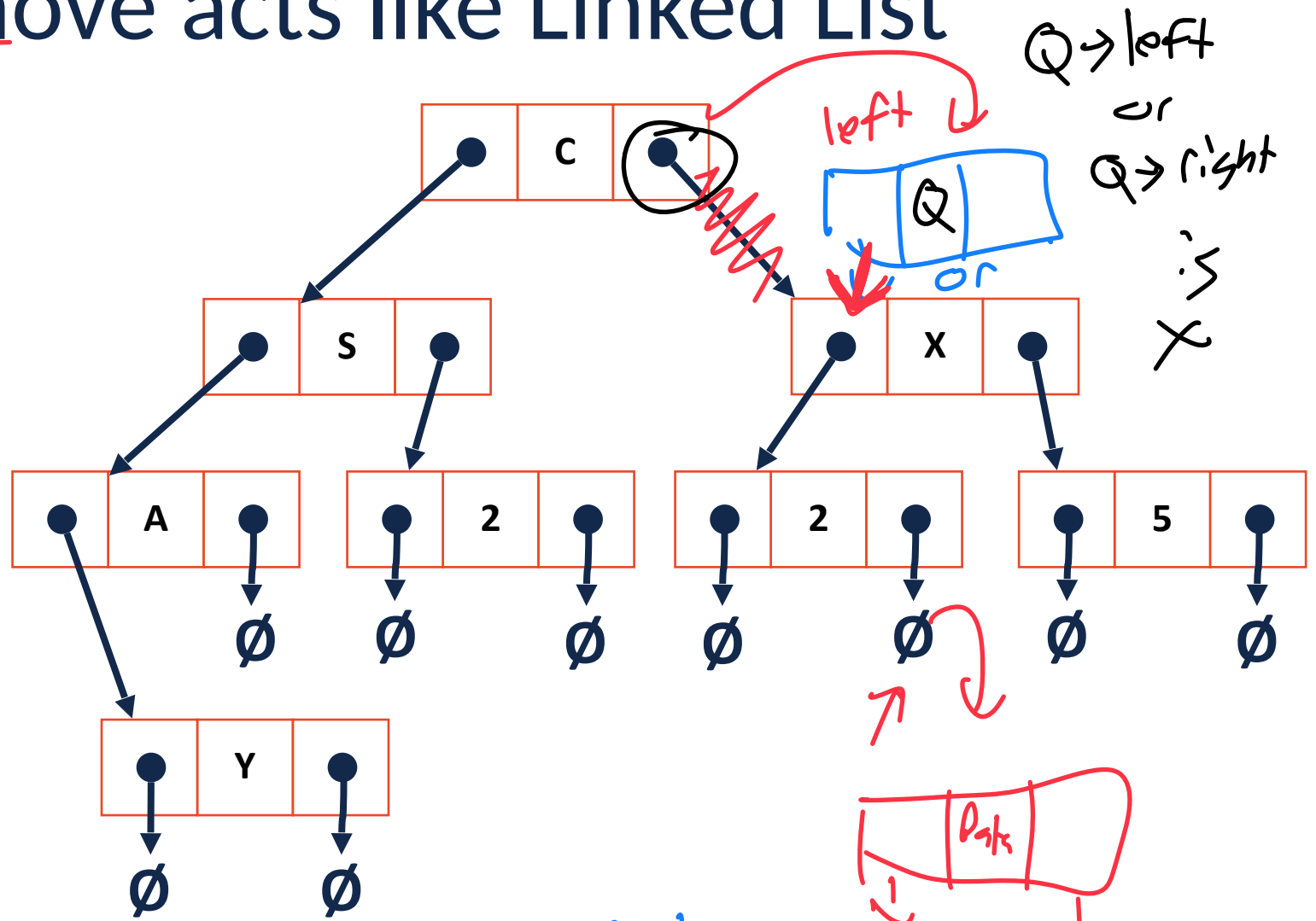
Visualizing trees



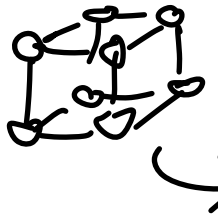


Tree Insert / Remove acts like Linked List





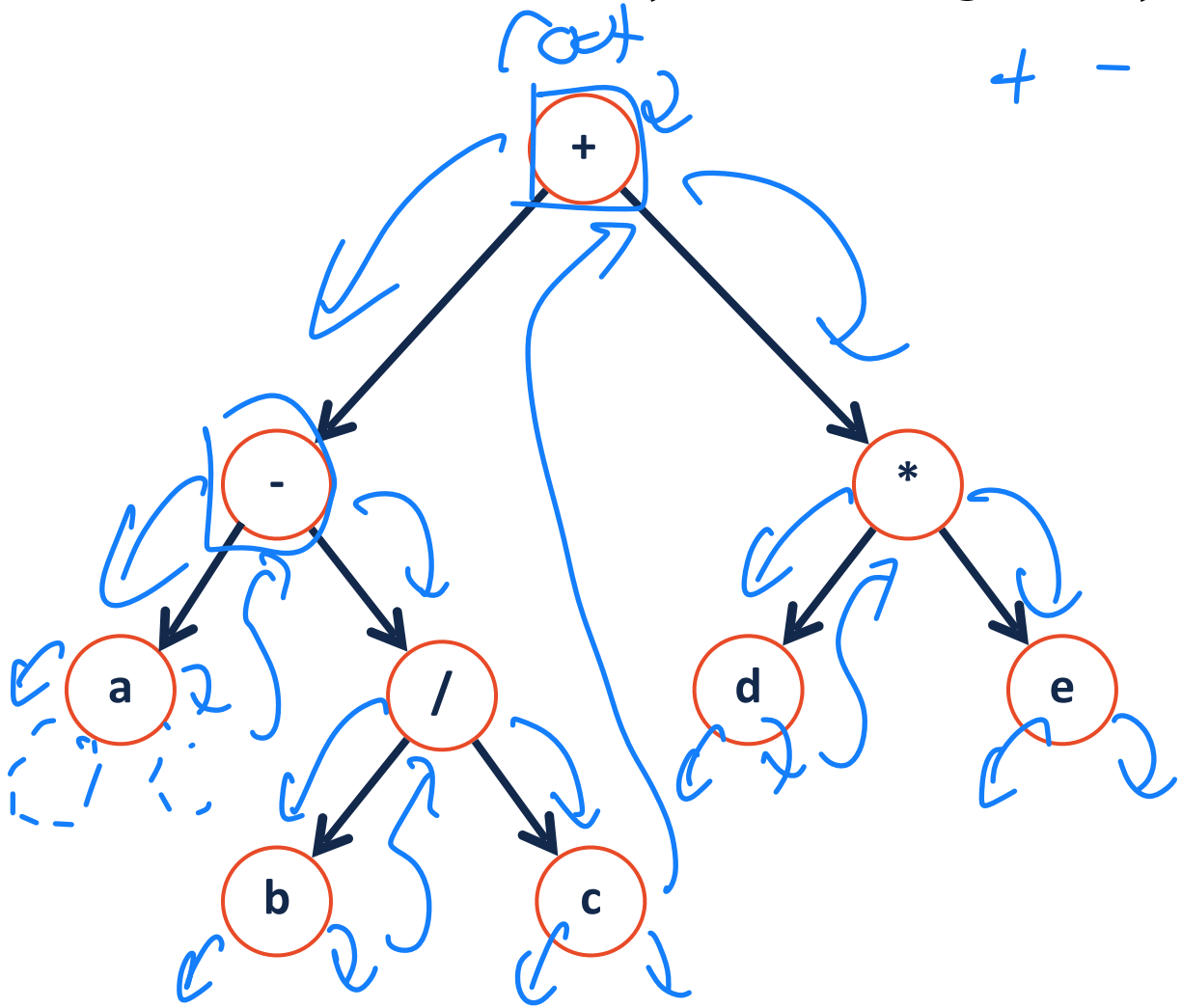
Tree Traversal



Iterator

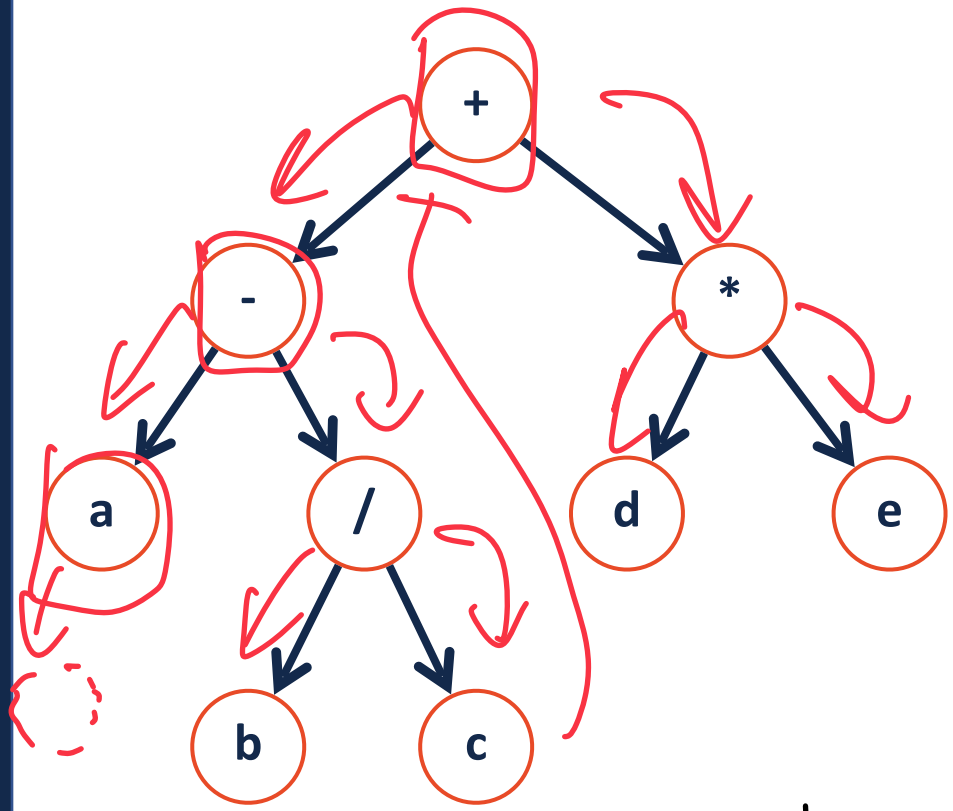
++i;

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



+ - a / b c * d e

Traversals



1 "process" node ← print

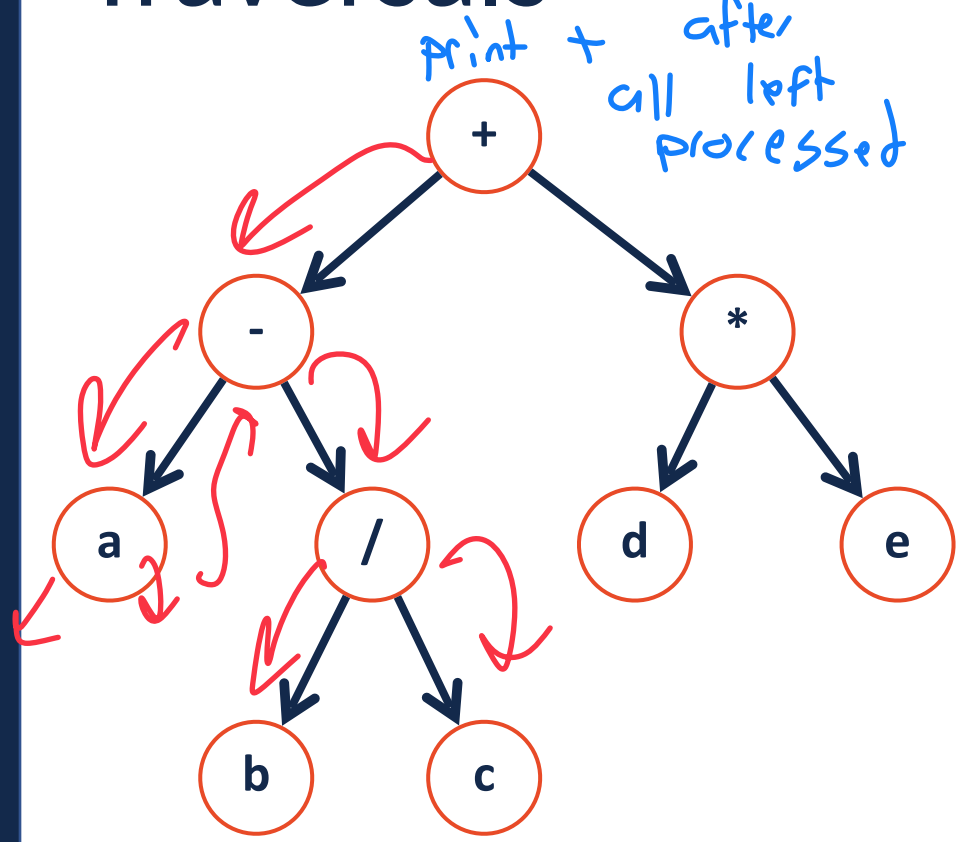
2 recurse left

3 recurse Right.

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

+ − a / b c * d e

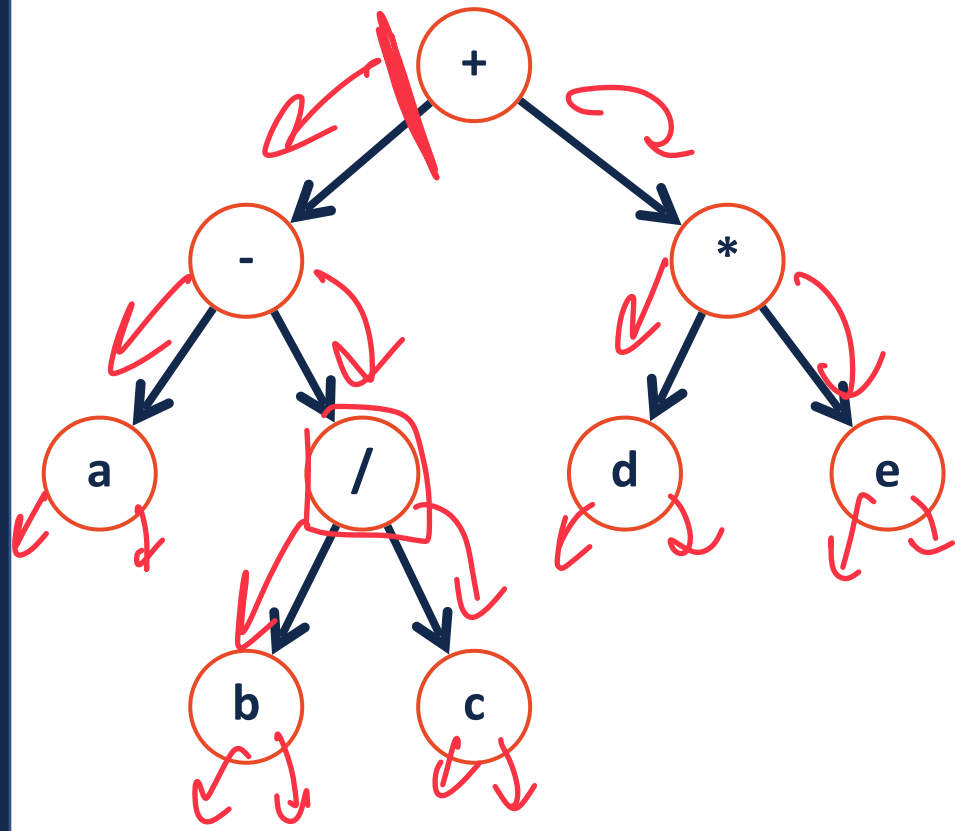
Traversals




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

a - b / c + d * e

Traversals



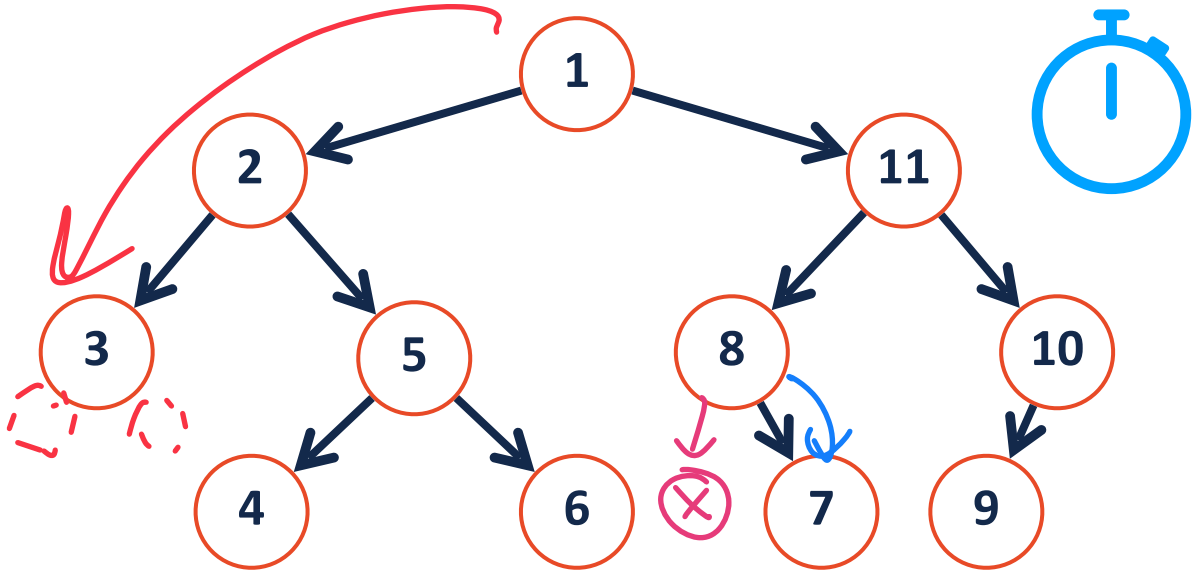

```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5    if (root) {
6
7        ____________________________;
8
9        _____Order(root->left);
10
11       ____________________________;
12
13       _____Order(root->right);
14
15       ____________________________;
16
17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

a b c / - d e * f

Tree Traversals

left

right



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

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

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

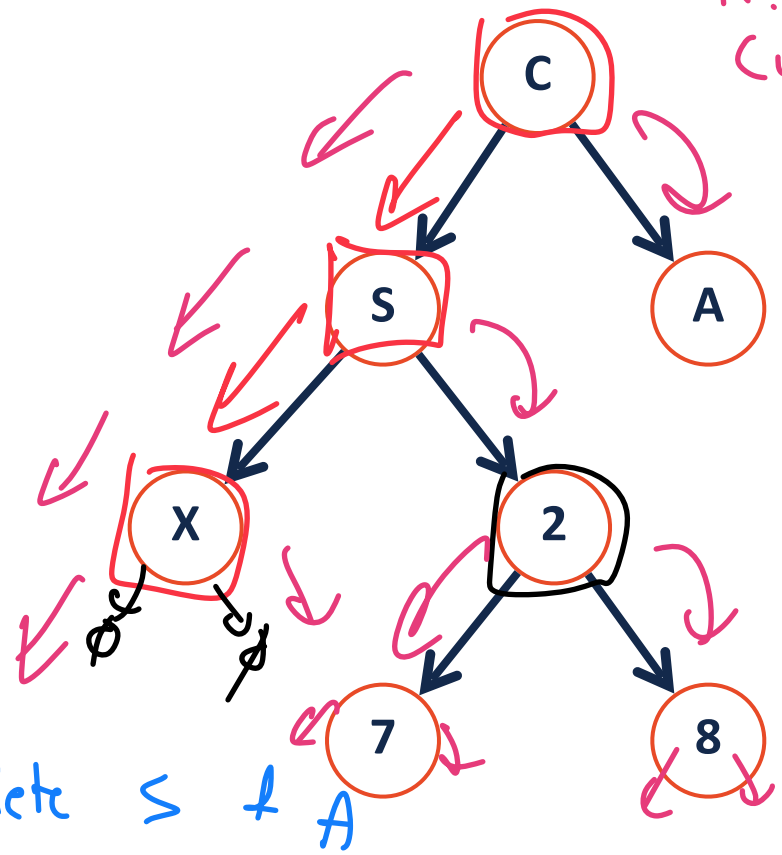
Tree Traversals

$T = \{root, T_L, T_R\}$

Post order:
Left
Right
Current

Pre-order: Ideal for copying trees

{C, {S, {X, ∅, ∅}, {2, {7, 8}}}, A}



Post-order: Ideal for deleting trees

↳ Before I delete root, I must delete T_L & T_R

↳ Before T_L

$x \in T_2$

↙ ↘

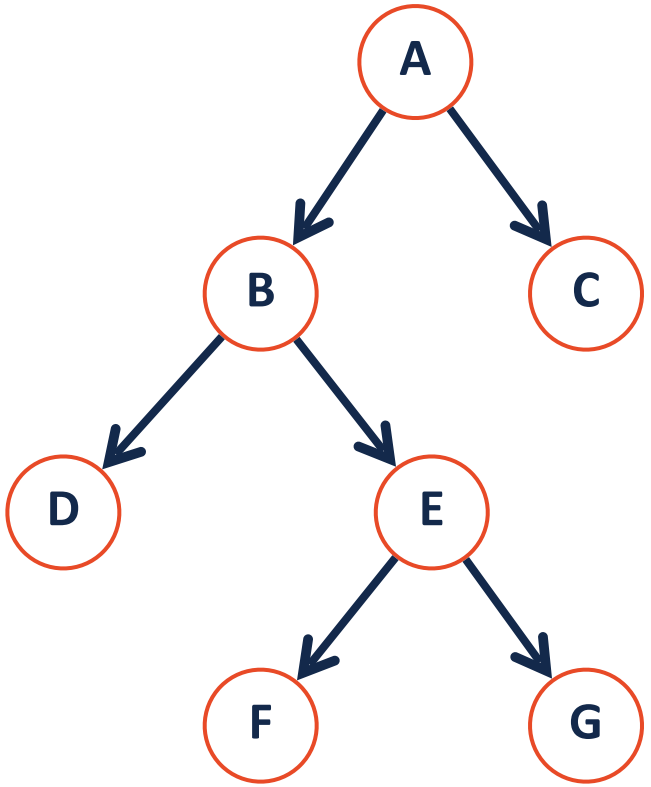
7 8

X 7 8 2 S A C

Traversal vs Search

Traversal – will visit every node once

↳ O(n) always



Search – Best way currently is traversal

↳ O(n)

↳ Actual time anywhere from 1 to N

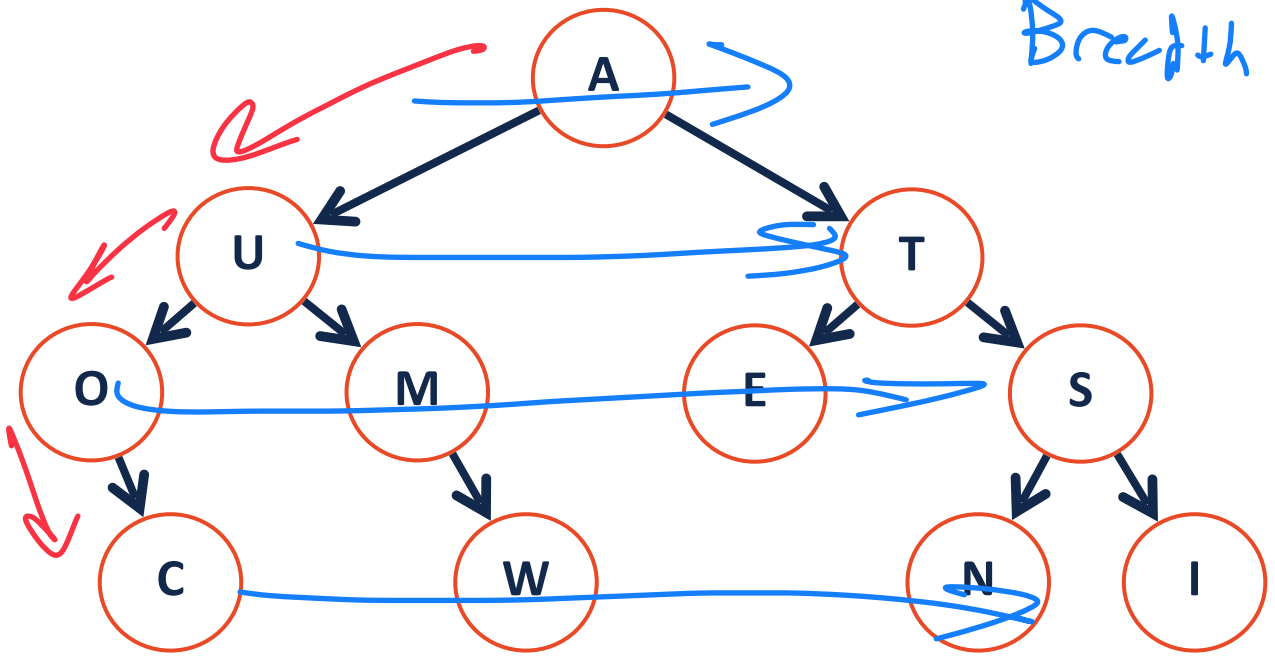
↳ Search can be done faster w/ tradeoff

Tree Search

There are two main approaches to searching a binary tree:

Depth first search

Breadth first search



Depth First Search

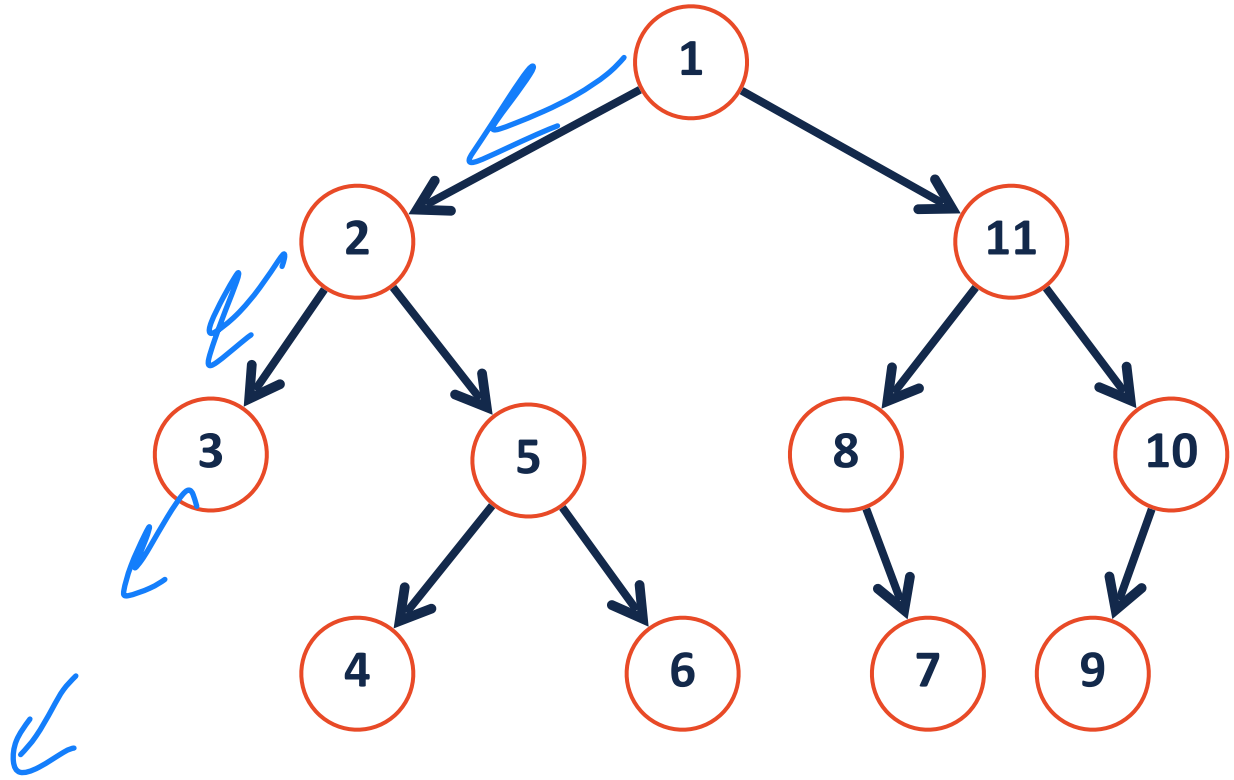
Explore as far along one path as possible before backtracking

↳ Pre order

↳ in order

↳ post order

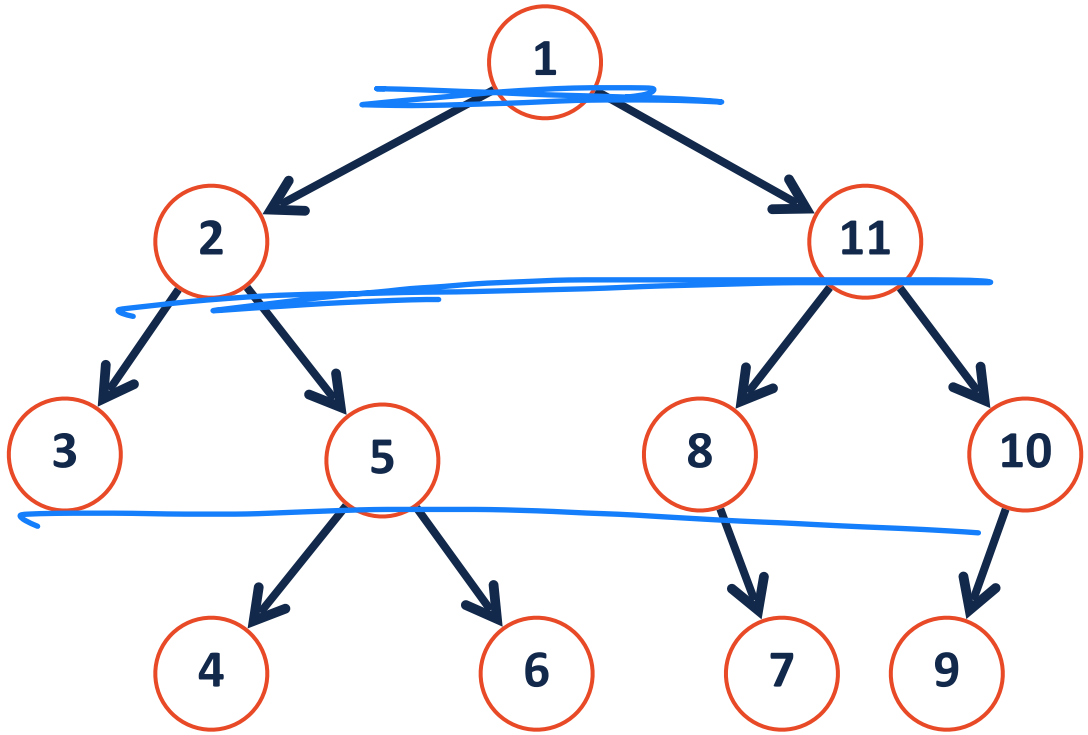
Stack!



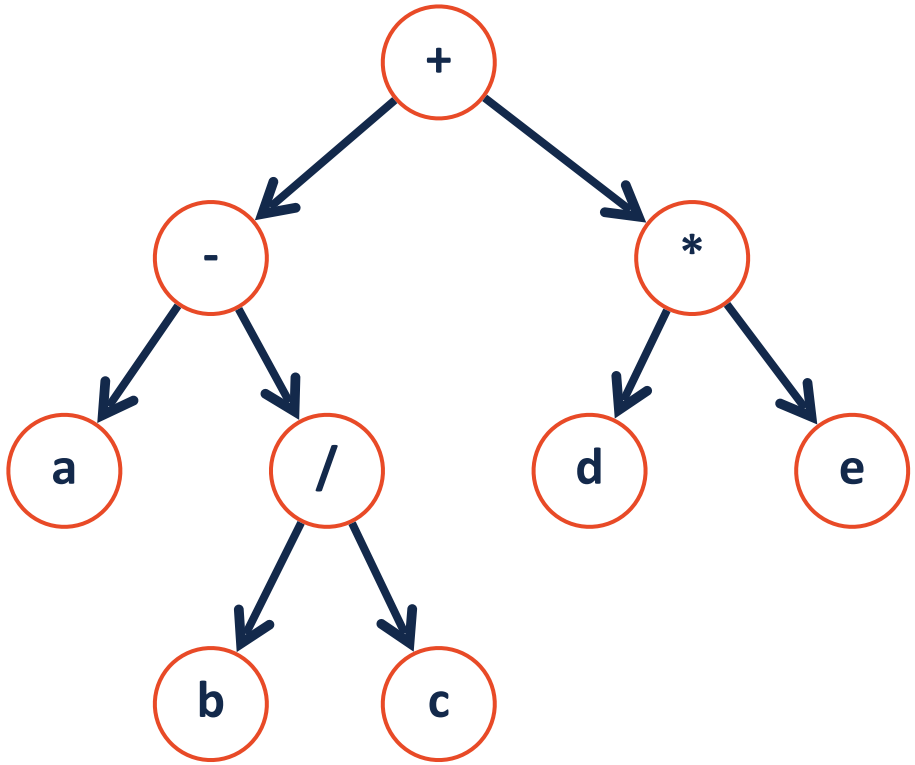
Breadth First Search

Fully explore depth i before exploring depth i+1

Queue!



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?