Data Structures

Tree Traversal

CS 225

Brad Solomon

September 16, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

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 4 points

Learning Objectives

Review and expand on foundational tree terminology

Discuss the tree ADT

Explore tree implementation details

Binary Tree

Join Code: 225

Lets review our understanding of binary tree terminology!

How many leaves are there in a perfect tree of height?

A) h leaves

C) 2^{h-1} leaves

B) 2^h leaves

D) h^2 leaves

Binary Tree

Join Code: 225

Lets review our understanding of binary tree terminology!

What is the lowerbound on # of nodes in a height h complete tree?

A) $2^{h+1} + 1$ nodes

C) $2^{h} + 1$ nodes

B) 2^{h} nodes

D) $2^{h+1} - 1$ nodes

Binary Tree

Join Code: 225

Lets review our understanding of binary tree terminology!

What is the upperbound on # of nodes in a height h complete tree?

A) $2^{h+1} + 1$ nodes

C) $2^{h} + 1$ nodes

B) $2^{h} - 1$ nodes

D) $2^{h+1} - 1$ nodes

Tree ADT

Insert

Remove

Traverse

Find

Constructor

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
9
10
11
12
13
14
15
16
17
18
19     };
20
21     TreeNode *root_;
22     /* ... */
23 };
```

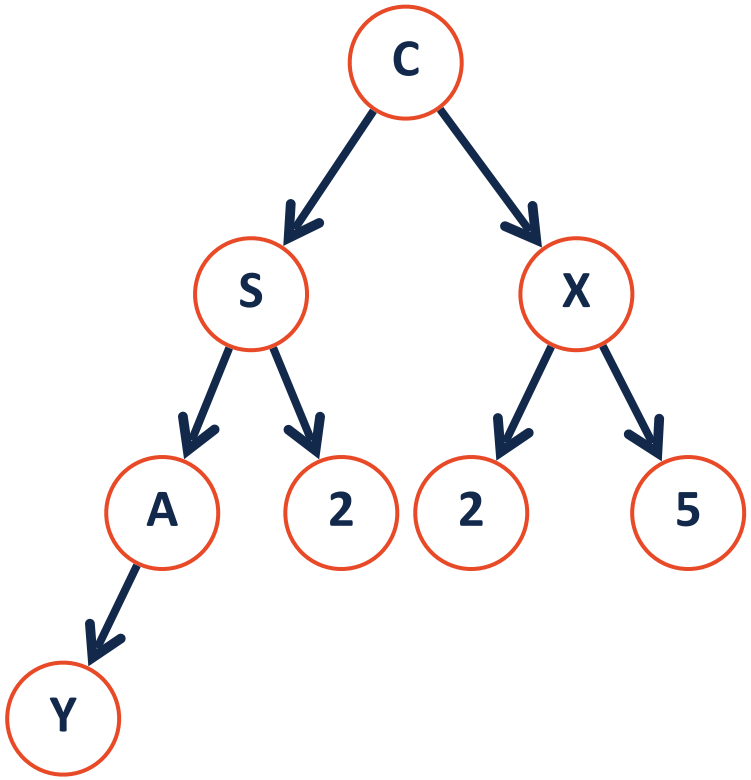
List.h

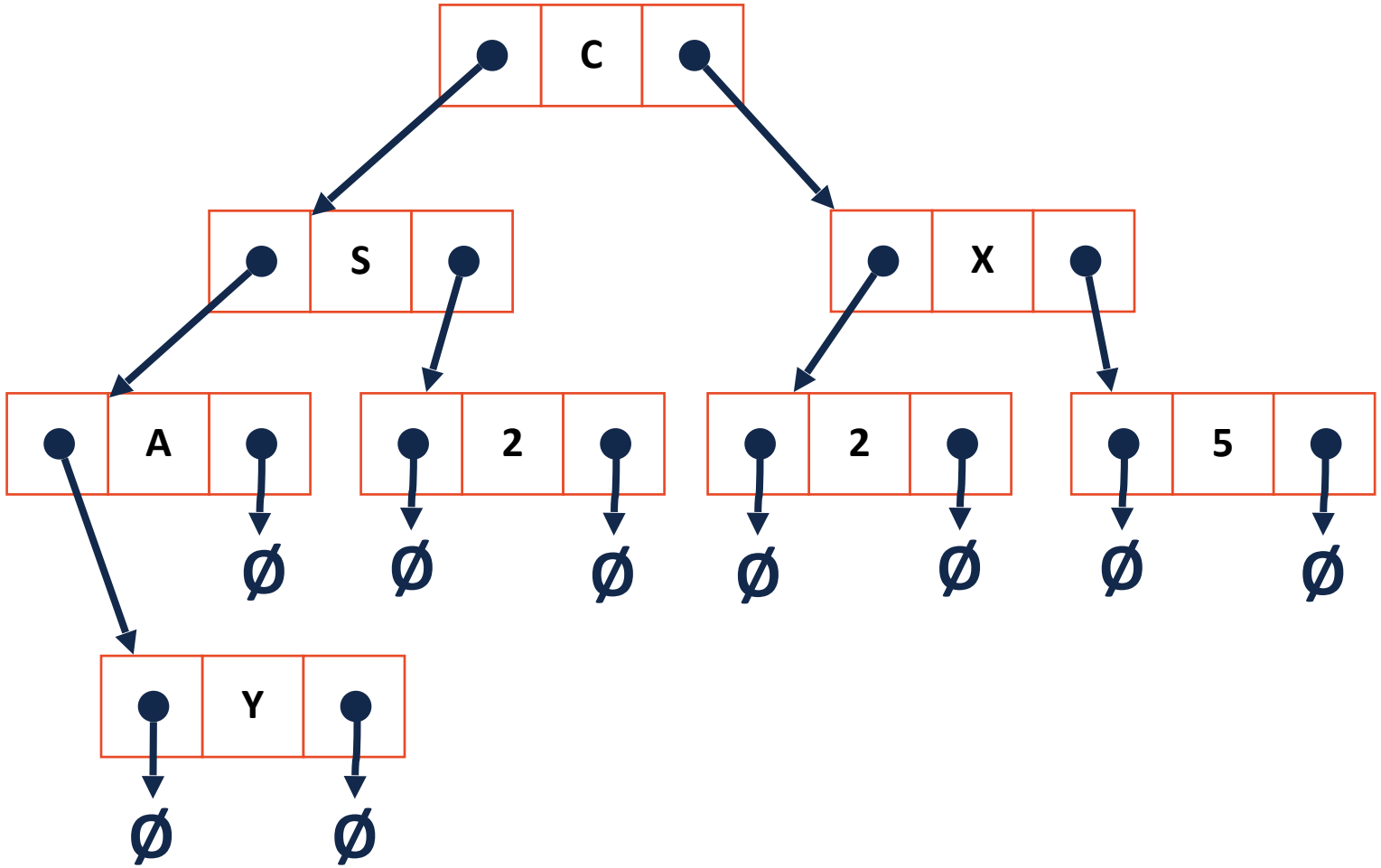
```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
6      /* ... */
7    private:
8      class ListNode {
9        T & data;
10
11       ListNode * next;
12
13
14
15       ListNode(T & data) :
16        data(data), next(NULL) { }
17     };
18
19
20
21     ListNode *head_;
22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

Tree.h

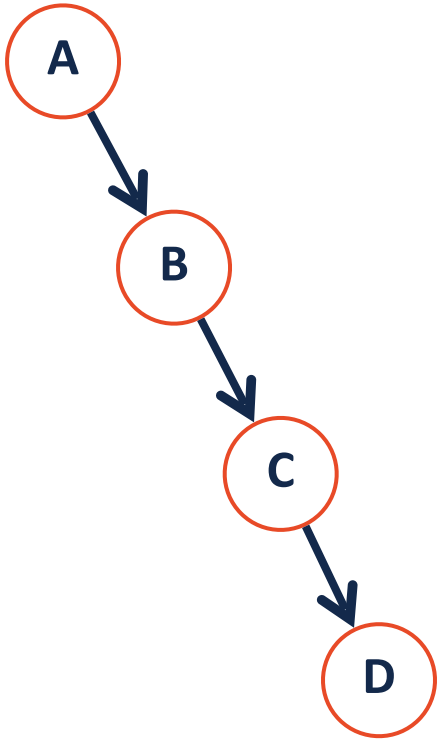
```
1  #pragma once
2
3  template <typename T>
4  class BinaryTree {
5    public:
6      /* ... */
7    private:
8      class TreeNode {
9        T & data;
10
11       TreeNode * left;
12
13       TreeNode * right;
14
15       TreeNode(T & data) :
16        data(data), left(NULL),
17 right(NULL) { }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

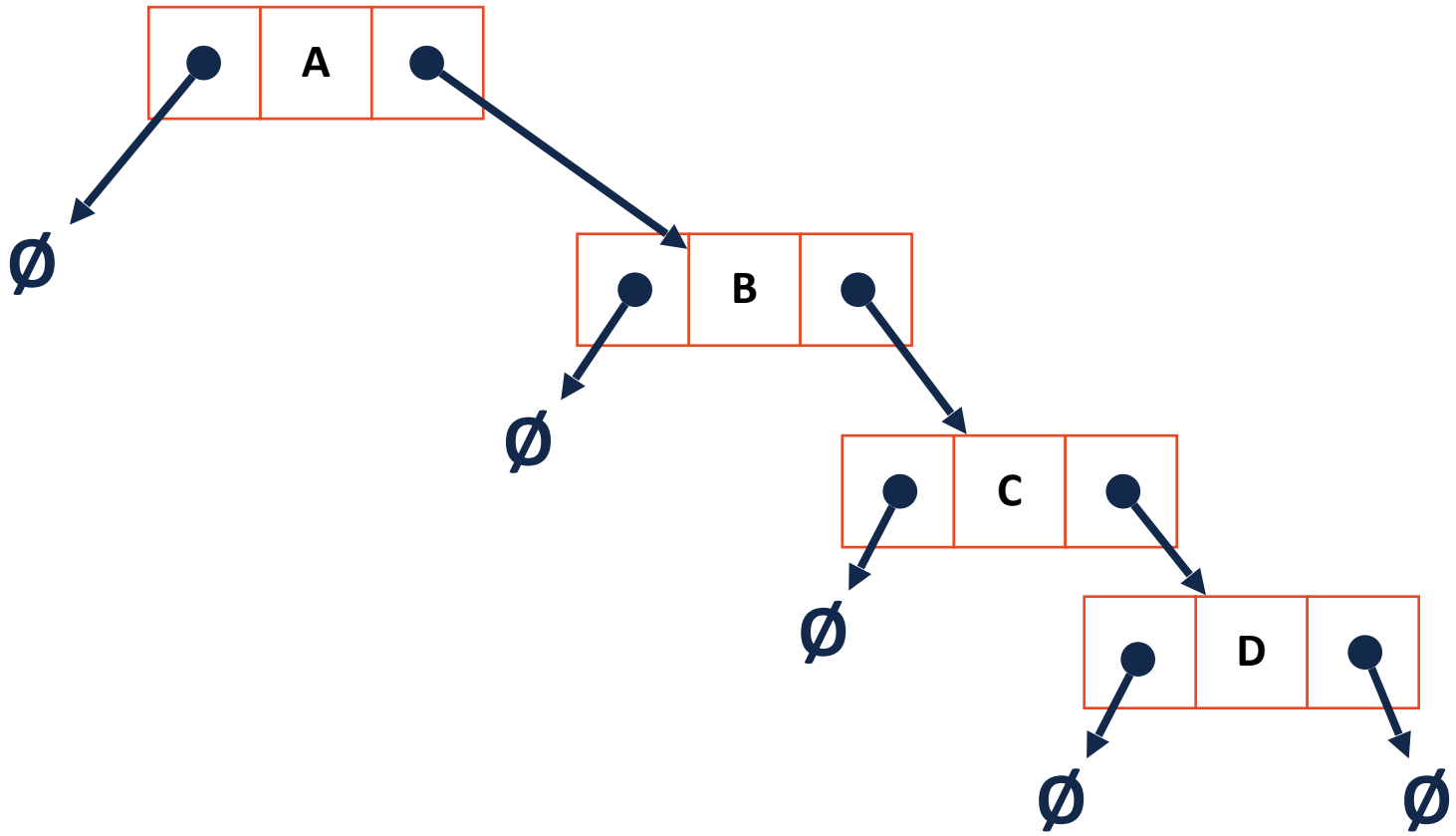
Visualizing trees





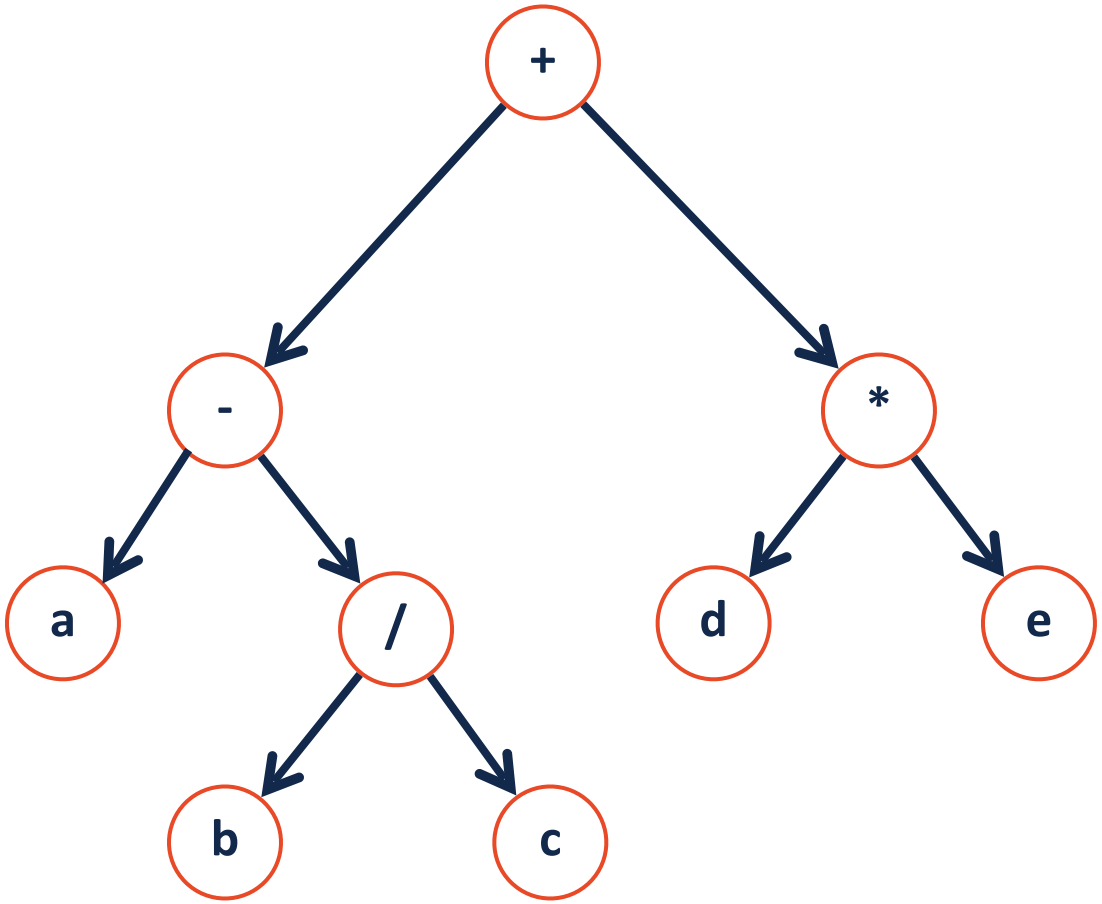
Tree Insert / Remove acts like Linked List



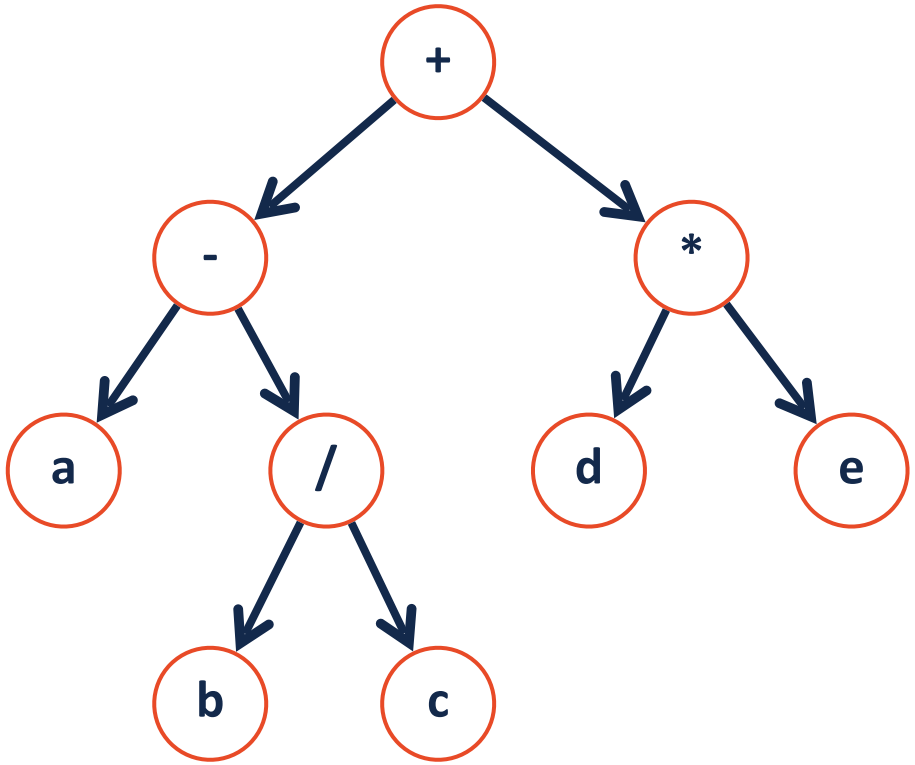


Tree Traversal

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

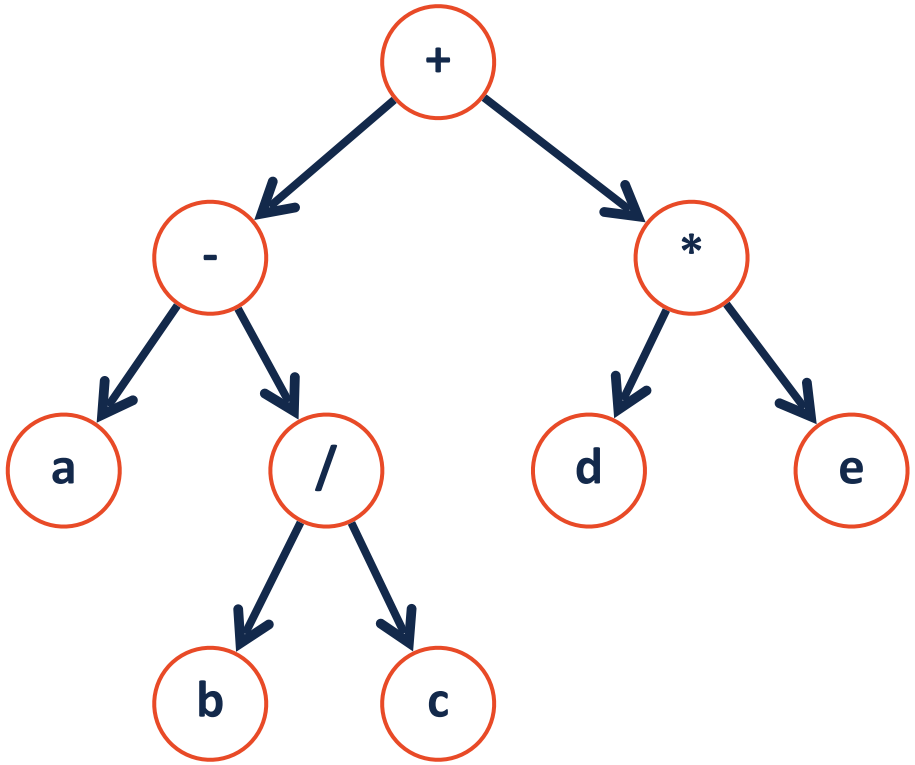


Traversals



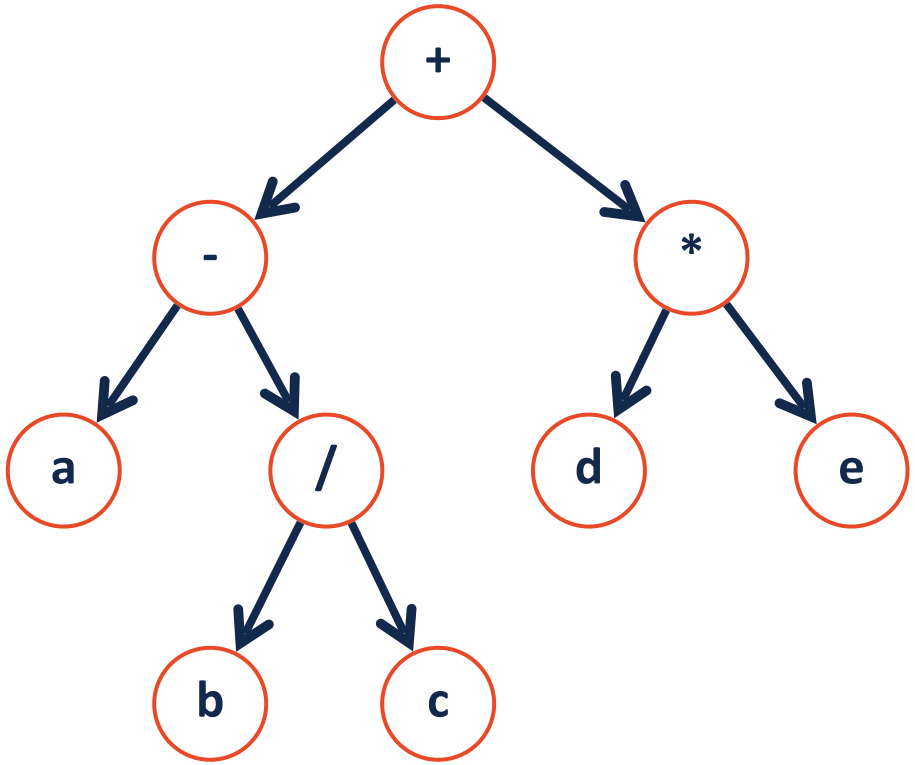

```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 }
```

Traversals



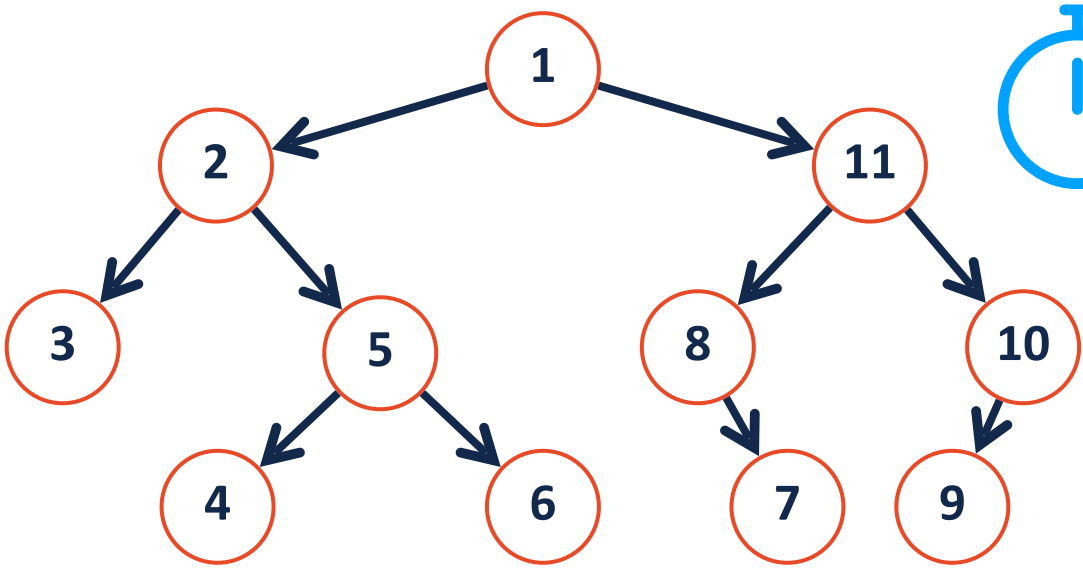

```
1  template<class T>
2  void BinaryTree<T>::_____Order(TreeNode * root)
3  {
4
5    if (root) {
6
7        ____________________________;
8
9        _____Order(root->left);
10
11       ____________________________;
12
13       _____Order(root->right);
14
15       ____________________________;
16
17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

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

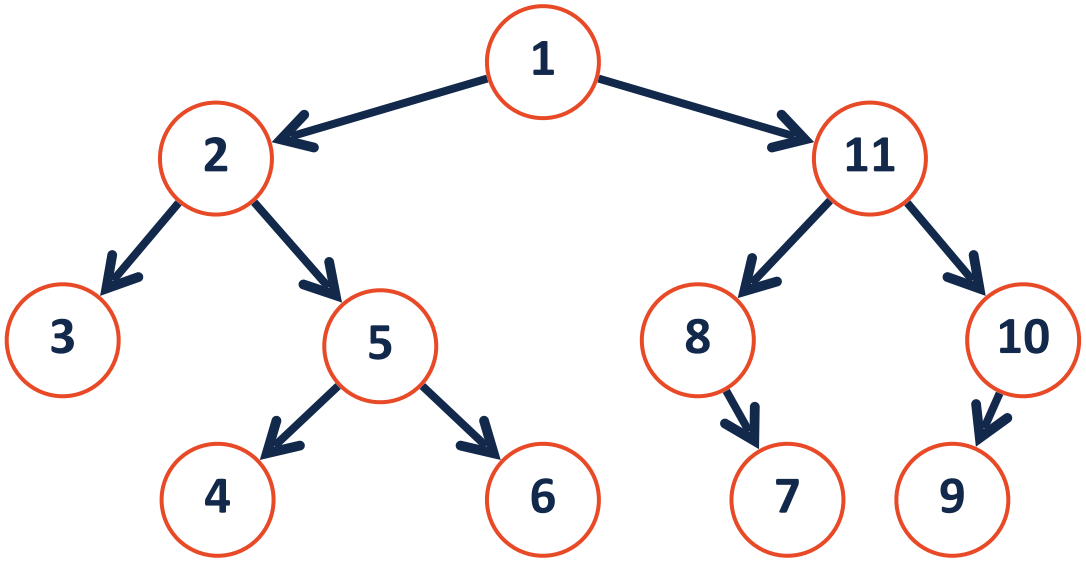


Pre-order:

In-order:

Post-order:

Tree Traversals (Solution)



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

Tip: Preorder always starts with root!

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

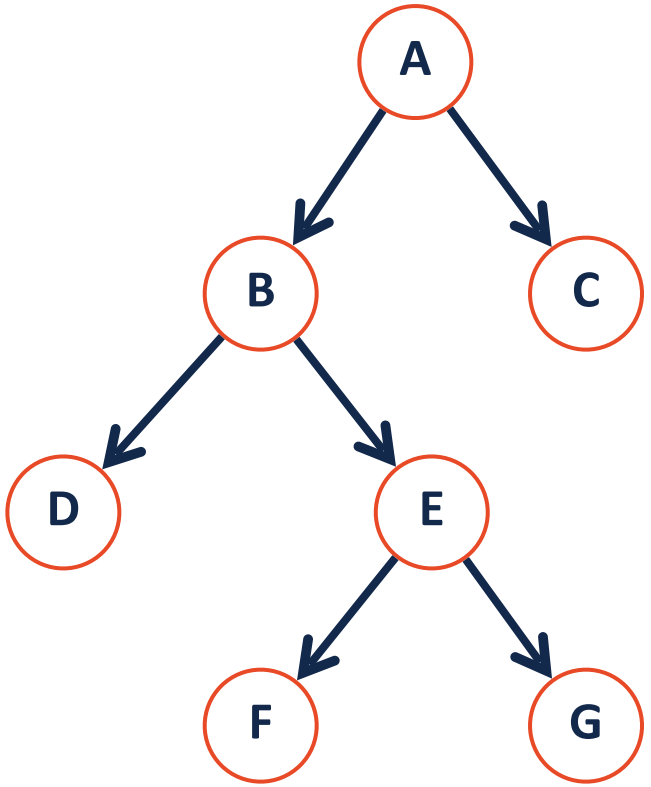
Tip: Inorder always starts with leftmost node. Root is after all left nodes!

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

Tip: Post always starts with leftmost node. Root is always LAST node!

Traversal vs Search

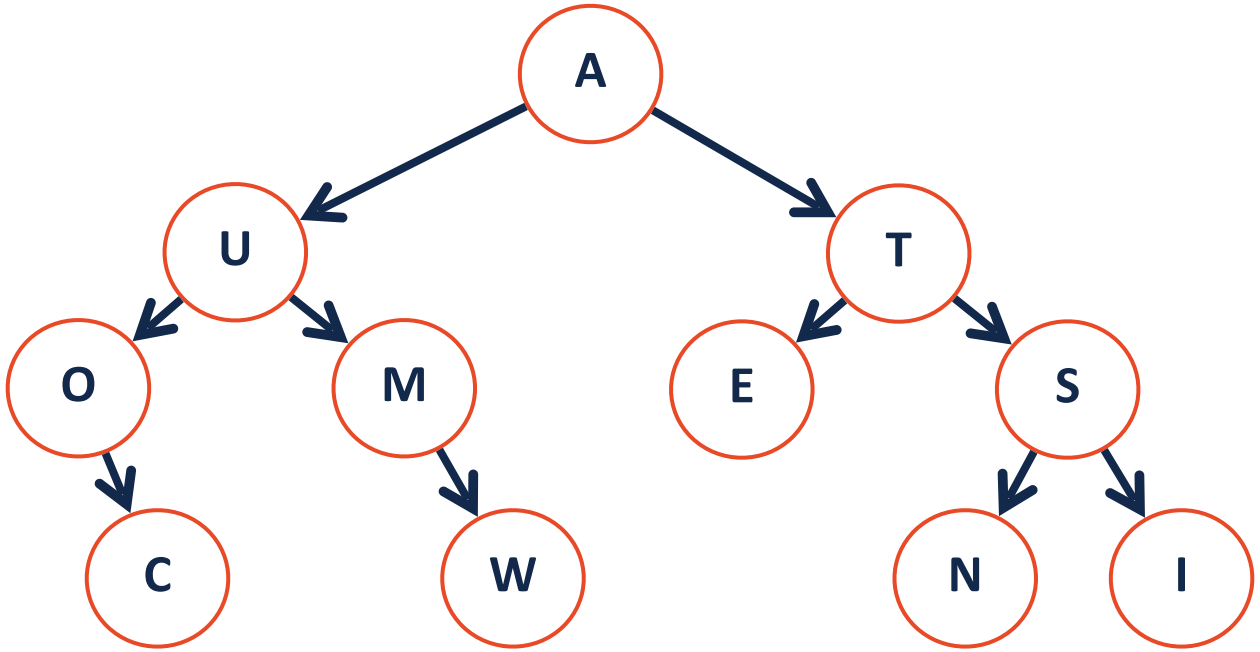
Traversal



Search

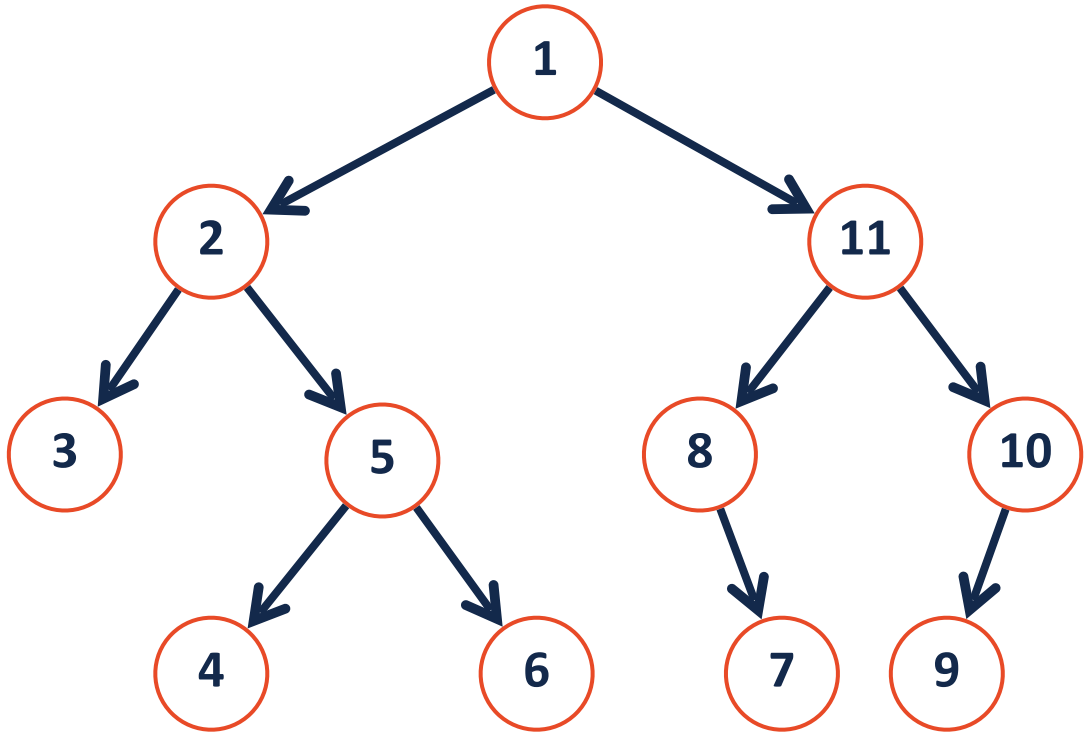
Tree Search

There are two main approaches to searching a binary tree:



Depth First Search

Explore as far along one path as possible before backtracking



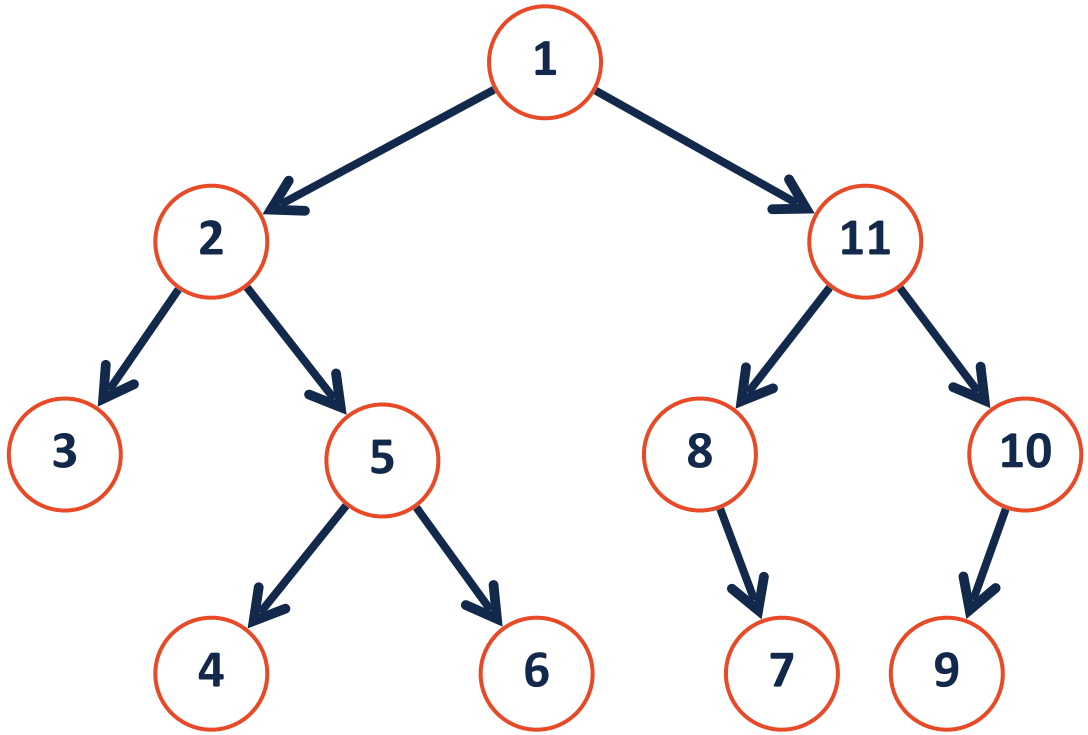
Depth First Search

Explore as far along one path as possible before backtracking

Make a stack initialized with root

While stack isn't empty:

- Pop top element (as `tmp`)
- Check if tmp is query (visual: 'print')
- Push `tmp->right` to stack
- Push `tmp->left` to stack



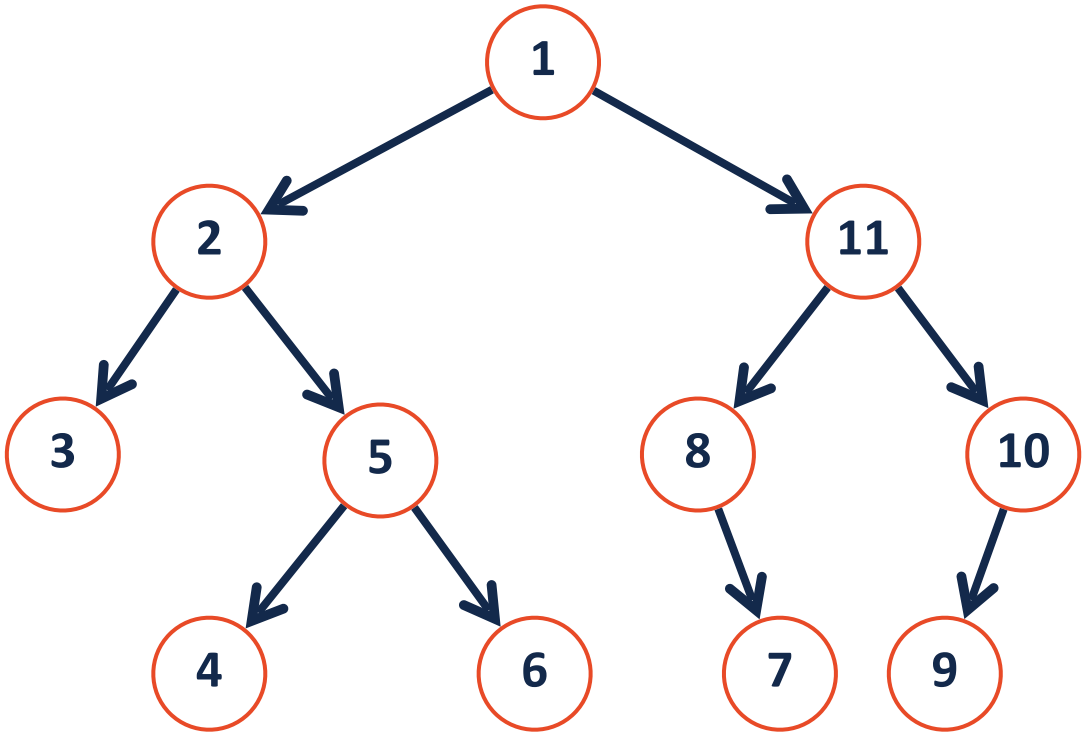
Stack: 1, 11, 2, 5, 3, 6, 4, 10, 8, 7, 9

Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

Depth First Search

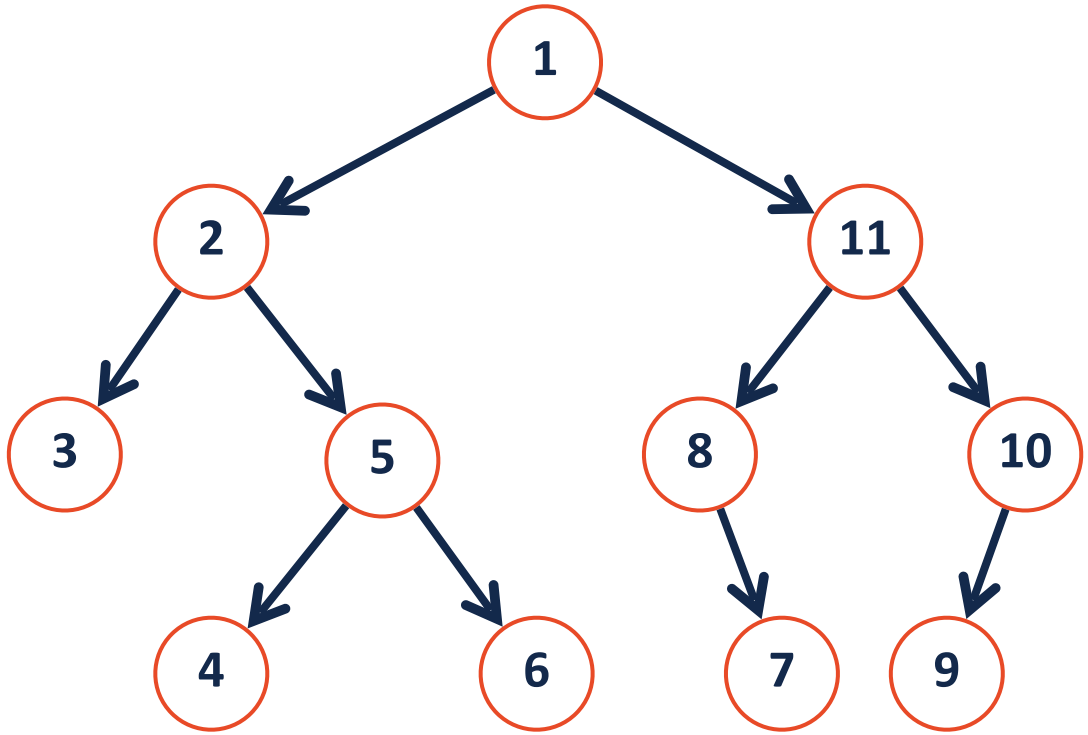
Explore as far along one path as possible before backtracking

```
1  template<class T>
2  bool BinaryTree<T>::DFS_search(TreeNode*
   root, T& query)
3  {
4      if (root) {
5          if (process(root->data, query)) {
6              return true;
7          }
8          if (DFS_search(root->left, query)) {
9              return true;
10         }
11         if (DFS_search(root->right, query)){
12             return true;
13         }
14     }
15     return false;
16 }
17
18
```



Breadth First Search

Fully explore level i before exploring level i+1



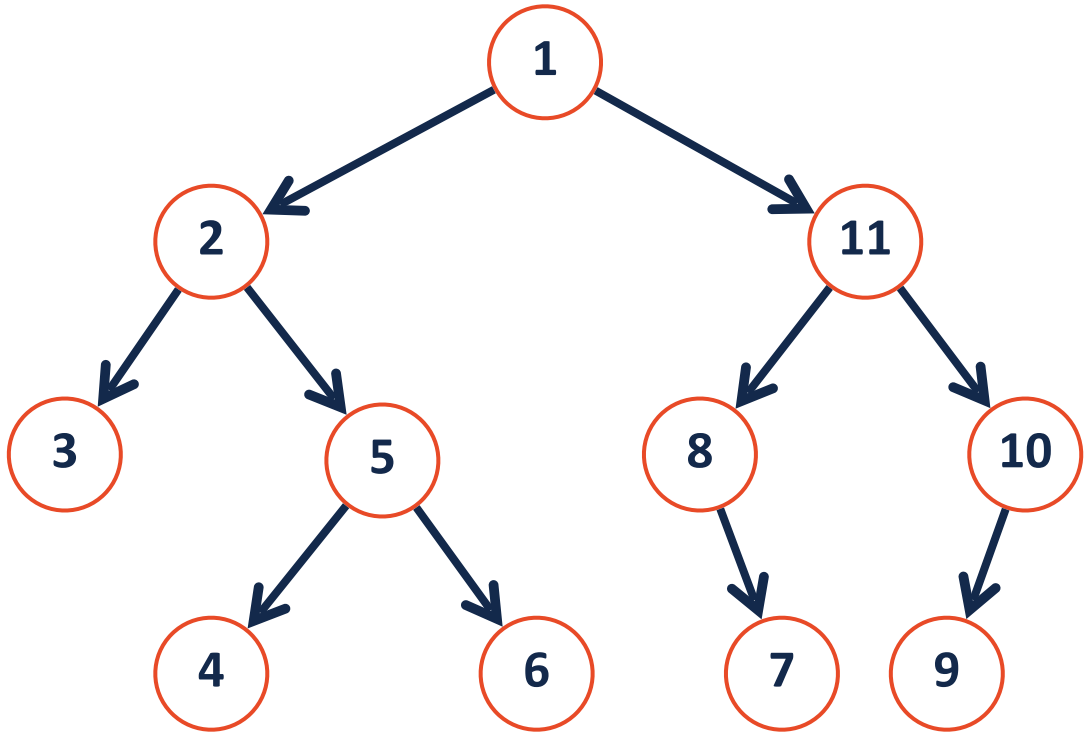
Breadth First Search

Fully explore level i before exploring level i+1

Make a queue initialized with root

While queue isn't empty:

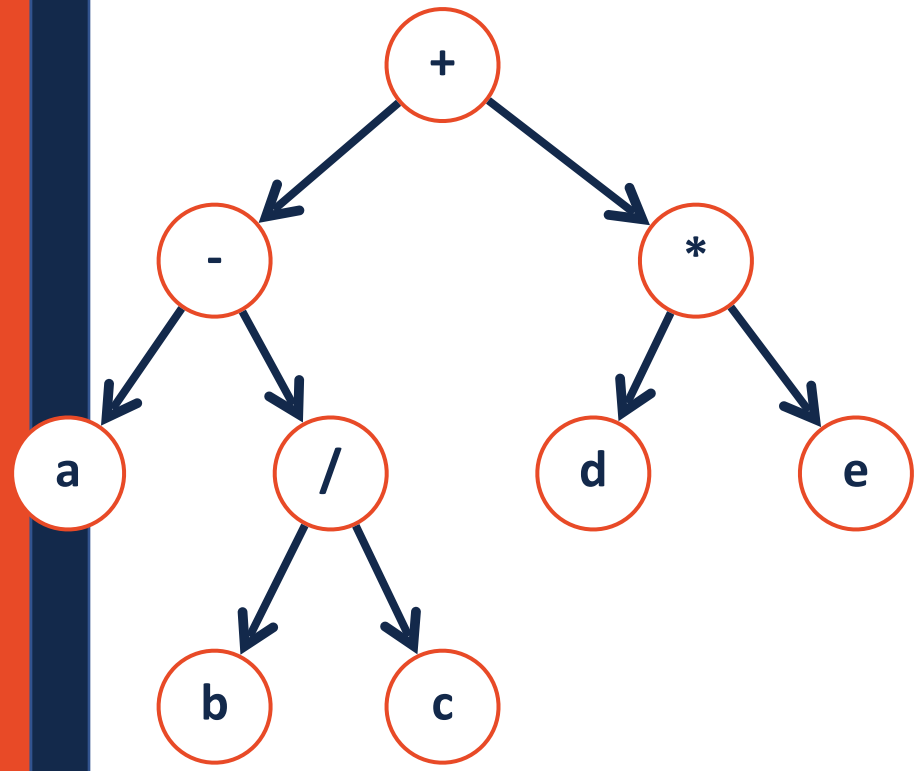
- Dequeue front element (as `tmp`)
- Check if tmp is query (visual: 'print')
- Enqueue `tmp->left`
- Enqueue `tmp->right`



Queue: 1, 2, 11, 3, 5, 8, 10, 4, 6, 7, 9

Print: 1, 2, 3, 5, 4, 6, 11, 8, 7, 10, 9

Level-Order Traversal




```
 1 template<class T>
 2 bool BinaryTree<T>::BFS_search(TreeNode* root, T&
   query)
 3 {
 4     if (!root) return false;
 5
 6     std::queue<TreeNode*> q;
 7     q.push(root); // enqueue == push
 8
 9     while (!q.empty()) {
10         TreeNode* temp = q.front();
11         q.dequeue();
12
13         if (proc(temp->data, query)) return true;
14
15         if (temp->left)  q.enqueue(temp->left);
16         if (temp->right) q.enqueue(temp->right);
17     }
18
19     return false;
20 }
```

Tree Search Tradeoffs

How can we improve our ability to search a binary tree?

What do we trade in order to do so?

Tree Search

How can we improve our ability to search a binary tree?

What do we trade in order to do so?