Data Structures

Tree Traversal

CS 225

September 15, 2023

Brad Solomon & G Carl Evans

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

State of the class

Exam scores generally match expected trends!

Exam 0: 77%

Exam 1: 83%

Plagiarism is at an 'all-time' low!

Remember — do not share code!

State of assessments

Lab_debug

44% of class found lecture helpful for completing lab

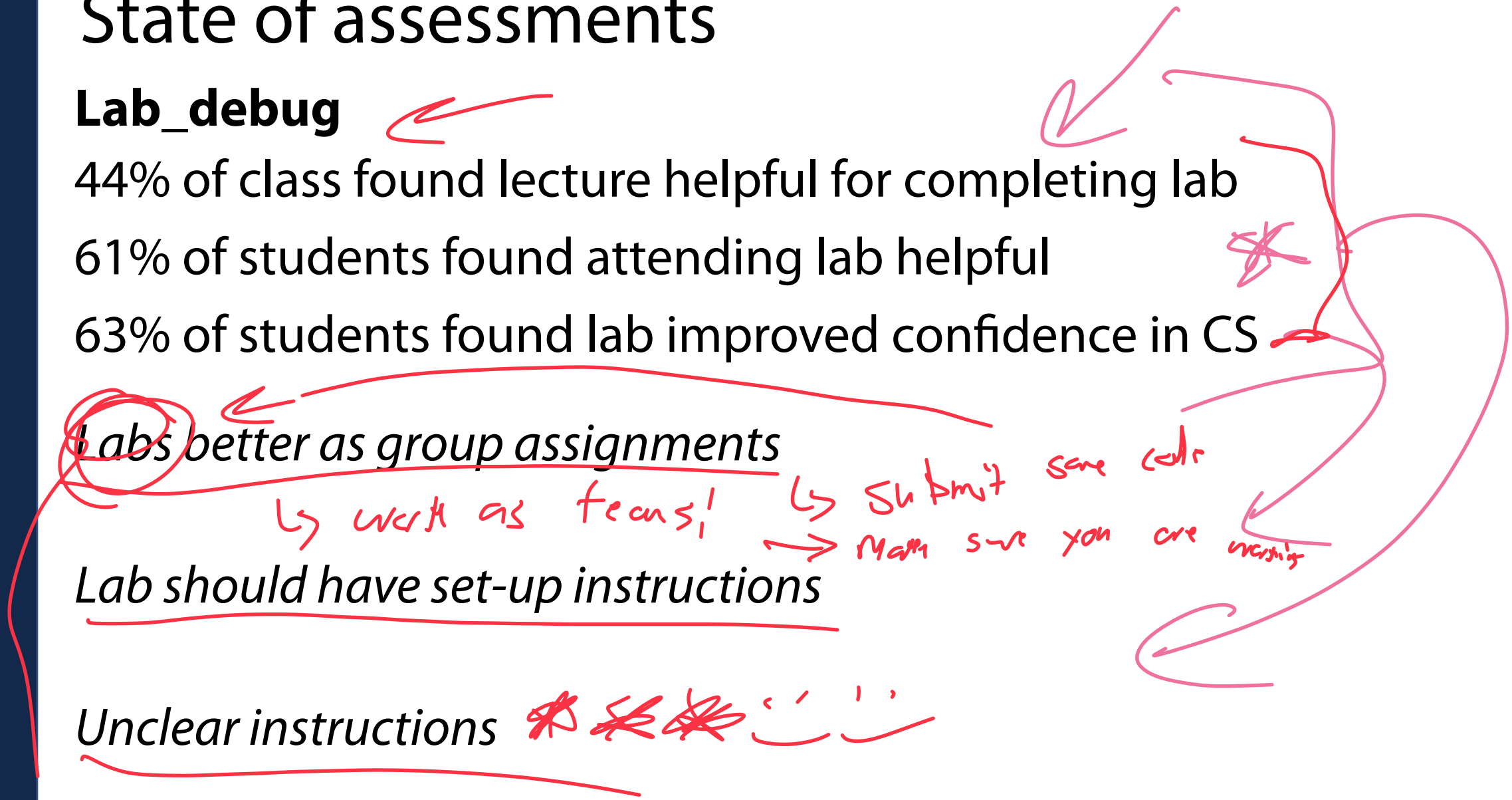
61% of students found attending lab helpful

63% of students found lab improved confidence in CS

Labs better as group assignments

Lab should have set-up instructions

Unclear instructions



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

No extra credit extensions

Queue Usage Reminder

Please include meaningful topic

Please include both your name and Discord username

Please be respectful to course staff

Learning Objectives

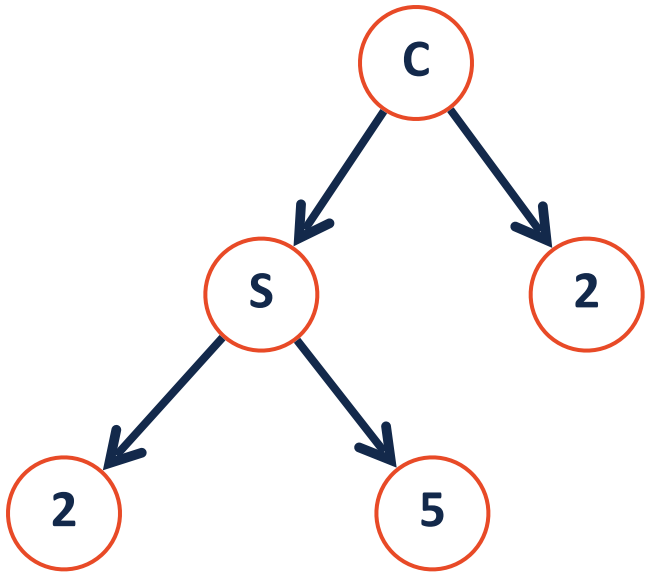
Discuss the tree ADT

Explore tree implementation details

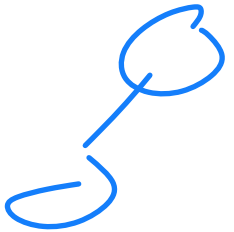
Binary Tree

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

1. $T = \emptyset$



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



Tree ADT

Insert – add data

Remove – remove data

GetData – Find / Search

Traverse – New?

Constructor – Construct empty tree

BinaryTree.h

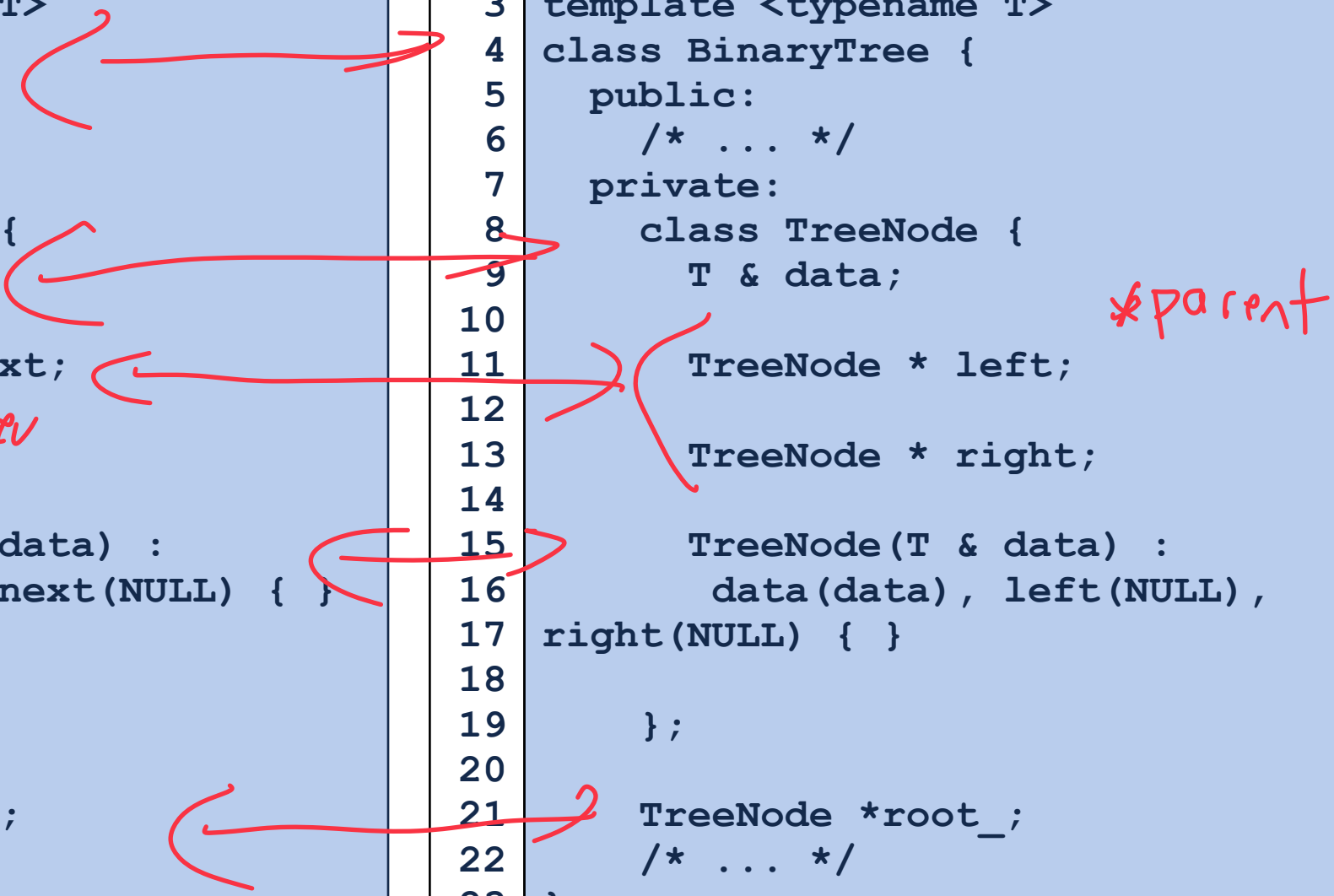
```
1  #pragma once
2
3  template <class T>
4  class BinaryTree {
5    public:
6      /* ... */
7
8    private:
9        class TreeNode {
10
11           T & data;
12
13           TreeNode *left, *right;
14
15           TreeNode(T & data): data(data), left(null), right(null) {}
16
17
18
19       };
20
21
22       TreeNode * root;
23
24
25 };
```

List.h

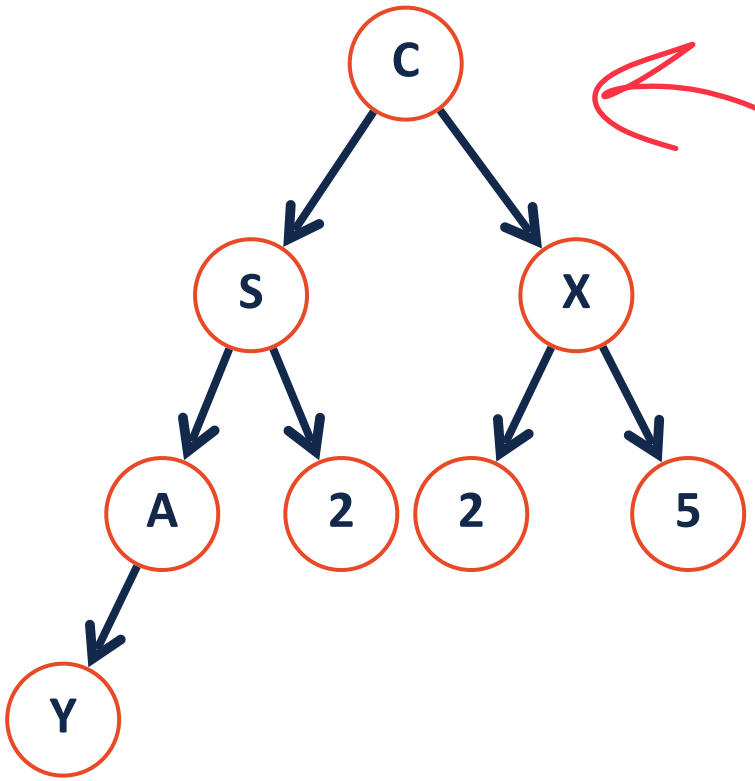
```
1  #pragma once
2
3  template <typename T>
4  class List {
5    public:
6      /* ... */
7    private:
8      class ListNode {
9        T & data;
10
11       ListNode * next;
12
13
14
15       ListNode(T & data) :
16        data(data), next(NULL) { }
17     };
18
19
20
21     ListNode *head_;
22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

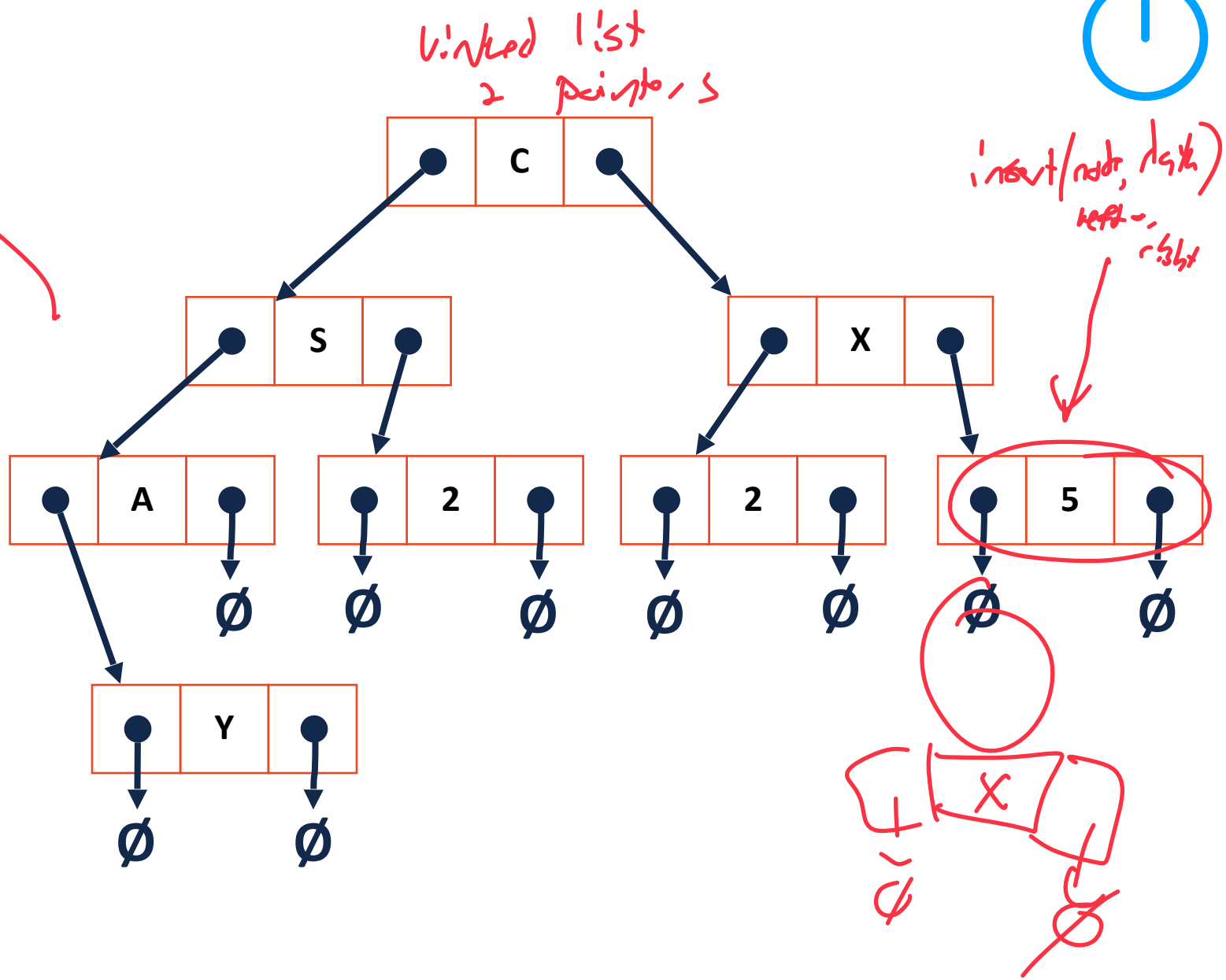
Tree.h

```
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

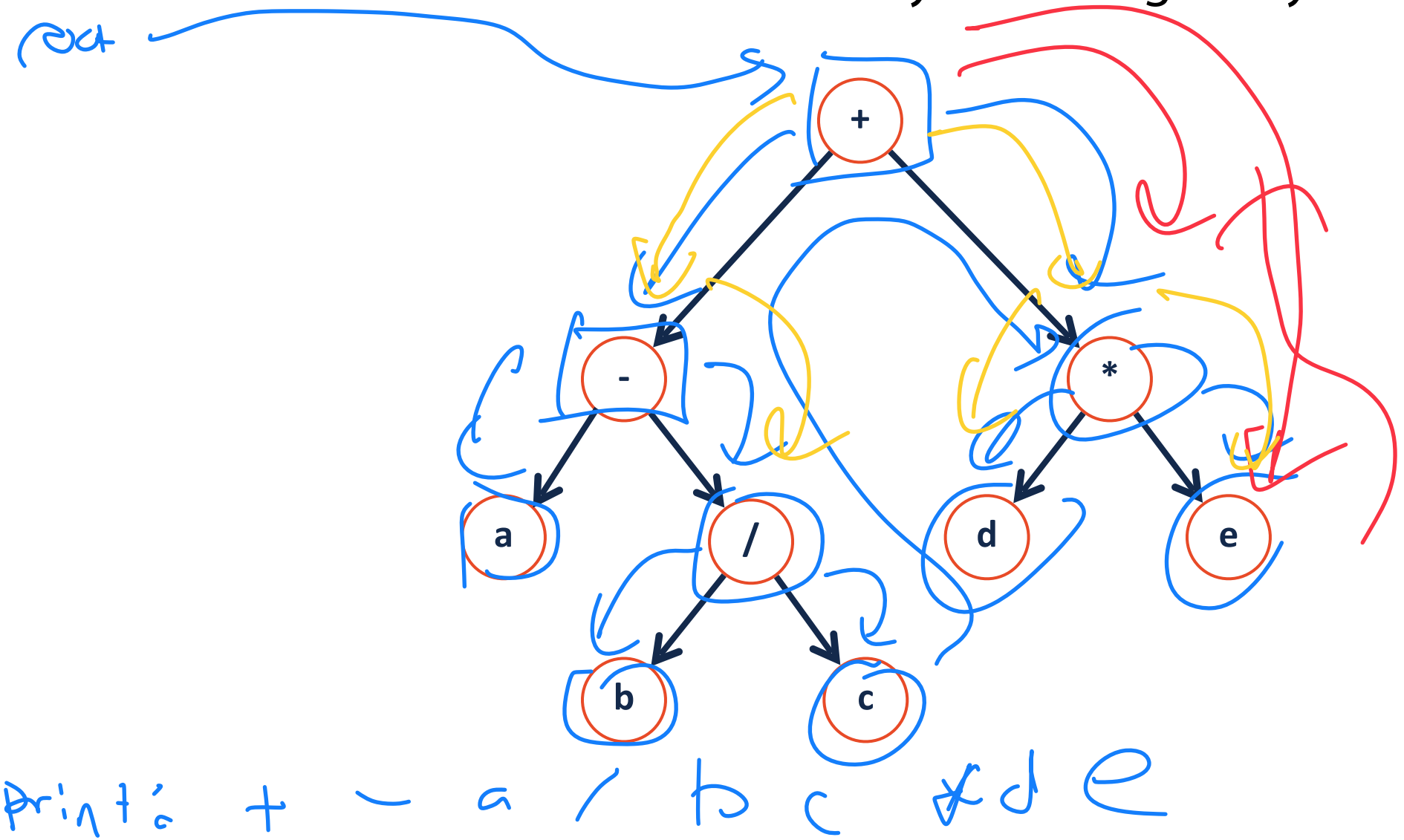




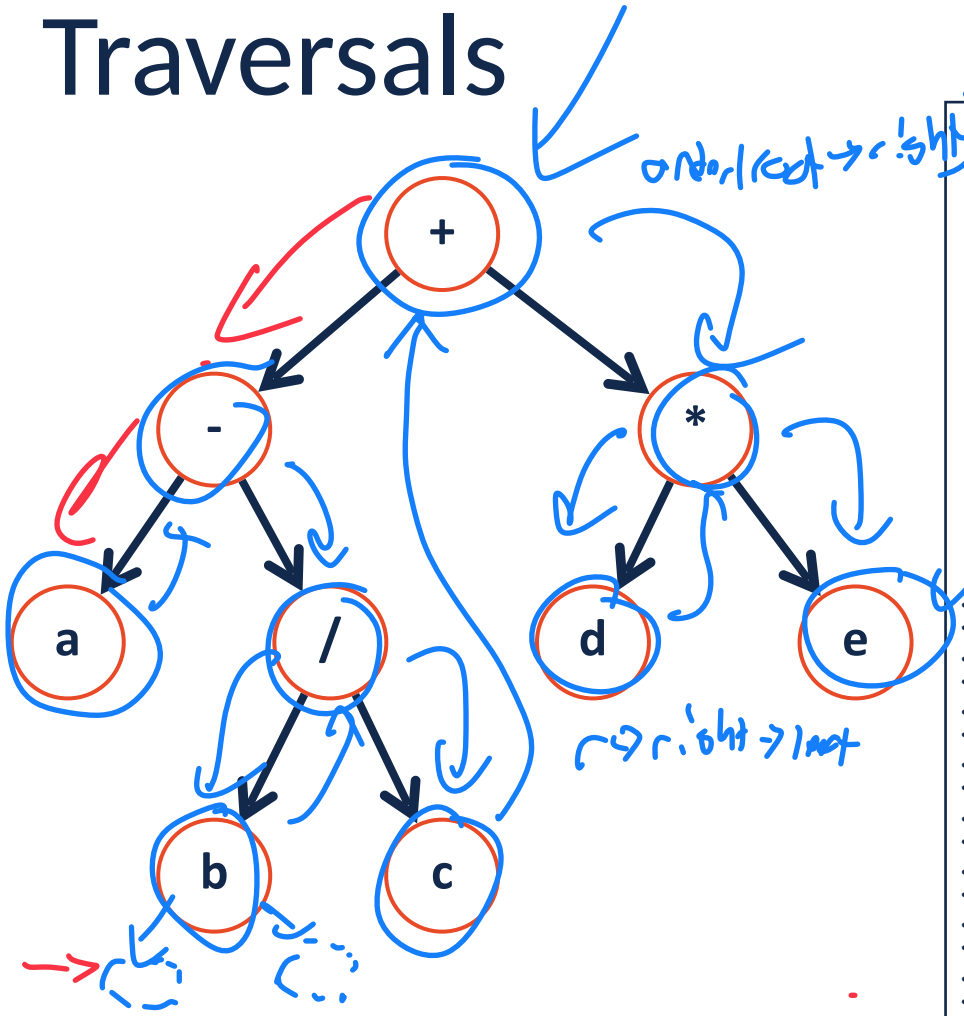
insert/remove/get Data

Tree Traversal

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



Traversals

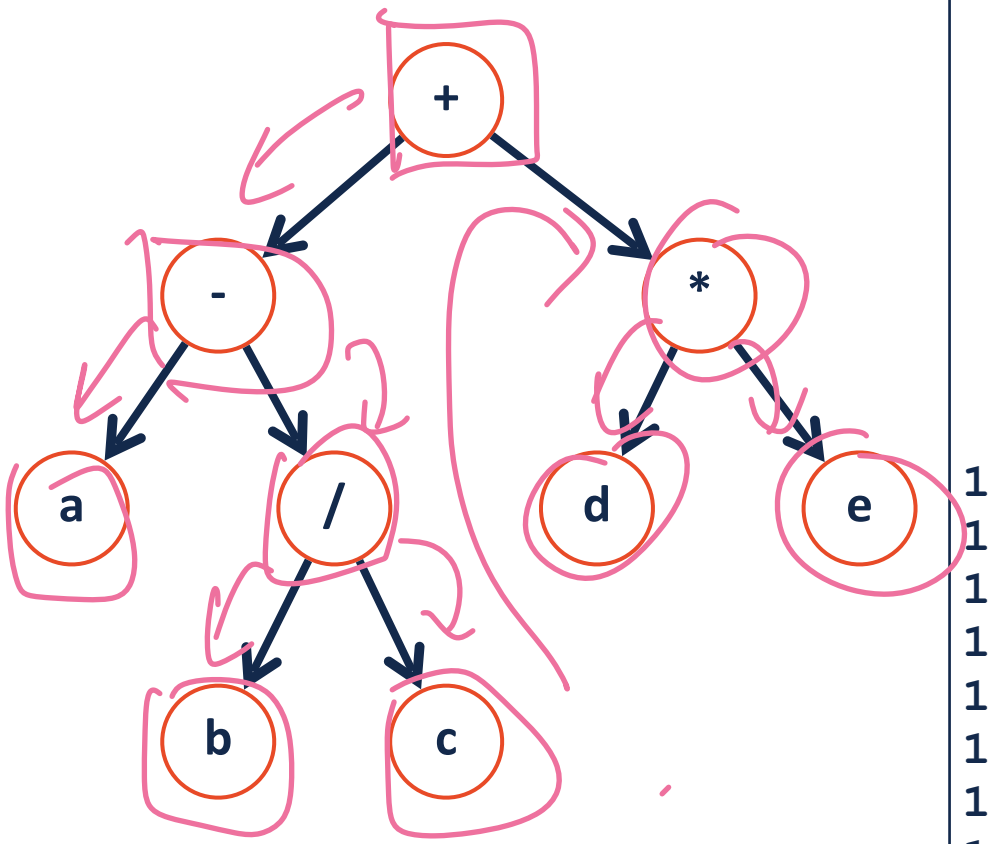



```
1  template<class T>
2  void BinaryTree<T>::____Order(TreeNode * root)
3  {
4   // Base case for empty list
5
6   if (root == nullptr) { return; }
7
8
9      in order (root -> left);
10
11     process (root); // handle current node
12
13
14     in order (root -> right);
15
16
17
18
19
20
21 }
```

1) visit left
2) process current.data
3) visit right

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 }
```

pre

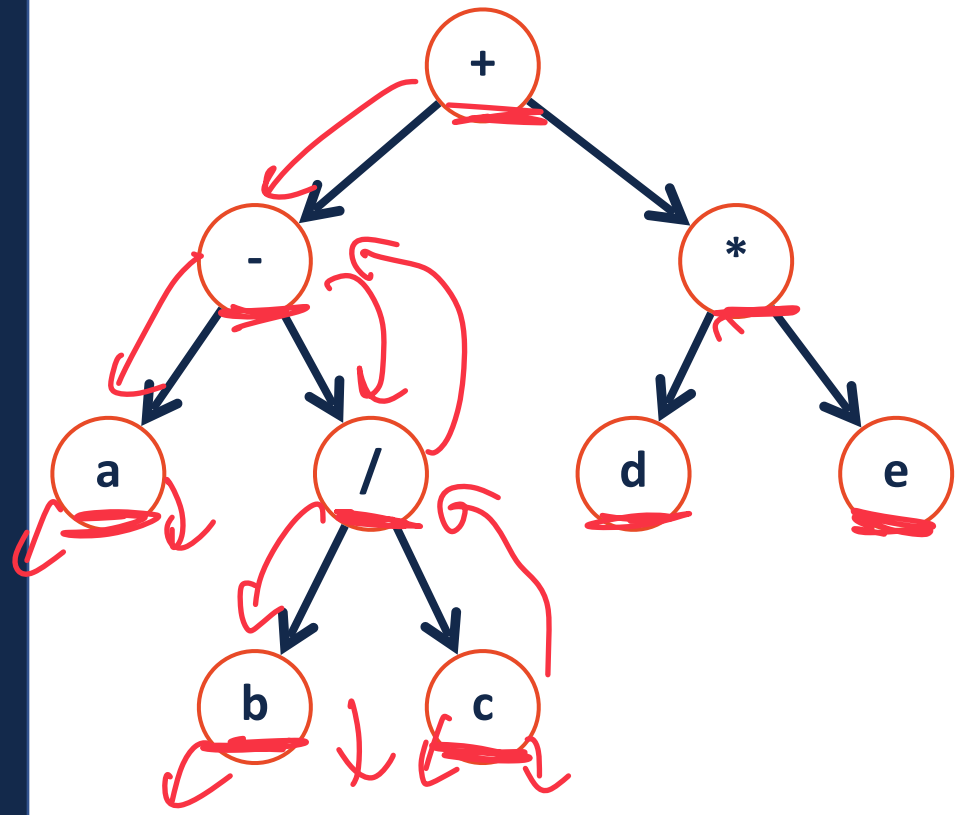
root is not nullptr

process (root)

in order

print: + - a / b c * d e

Traversals

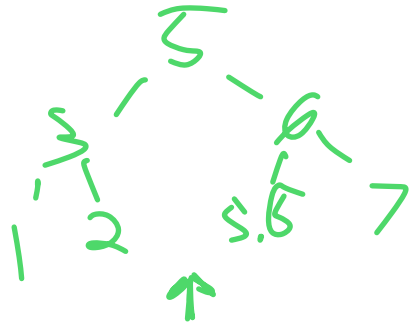


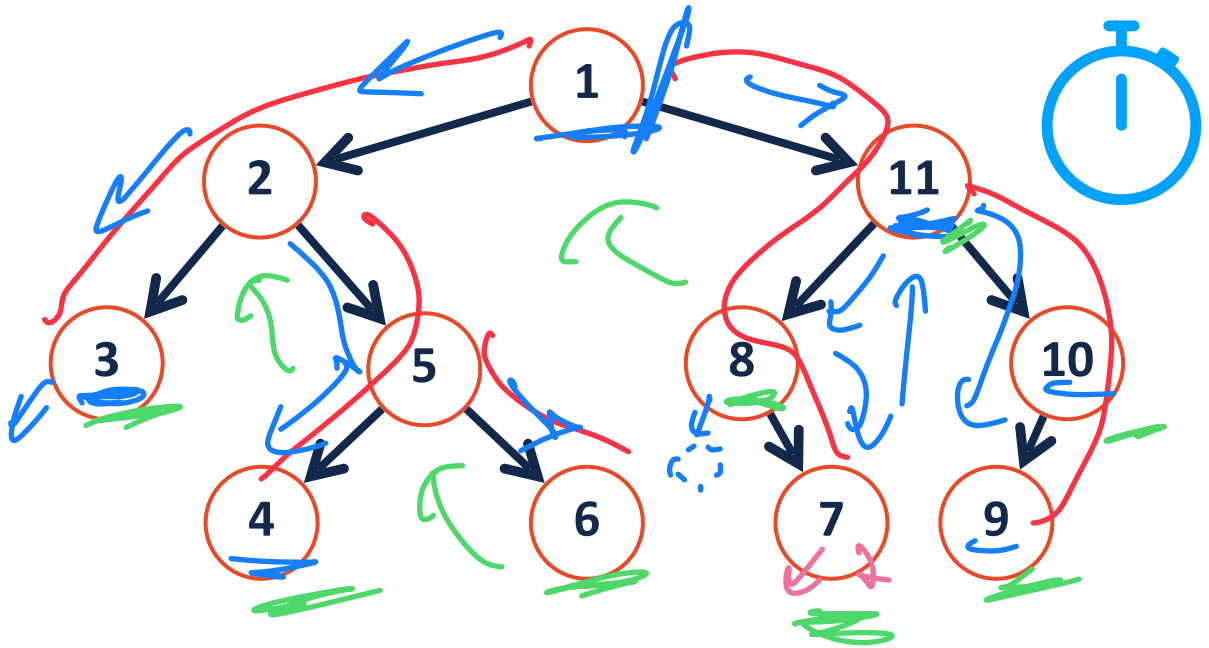
post_order

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

print: a b c / - d e * f

Tree Traversals





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

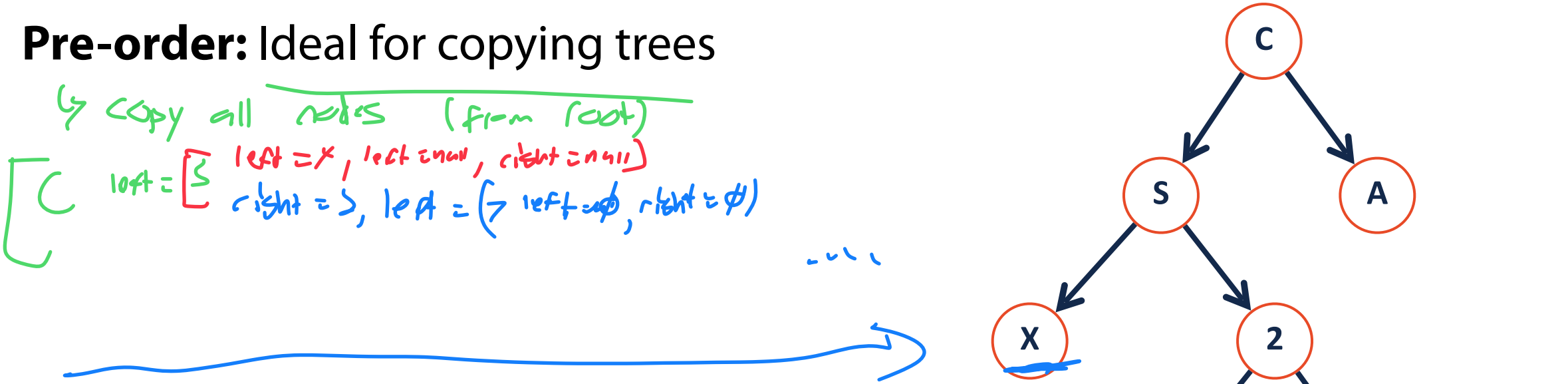
In-order: 3 2 4 5 6 1 8 7 11 9 10

left curr right

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

Tree Traversals

Pre-order: Ideal for copying trees



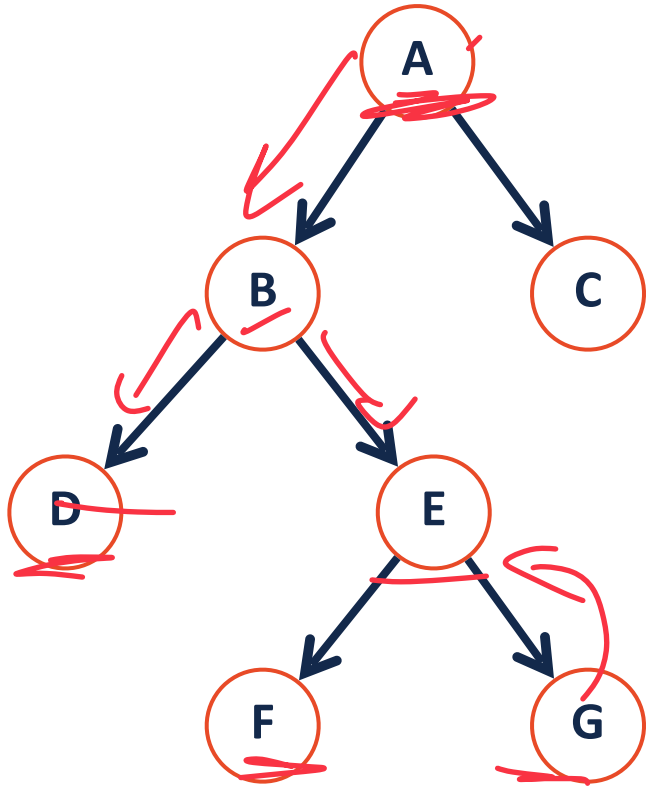
Post-order: Ideal for deleting trees

In order traversal is much more useful if tree is structured or ordered intelligently

Traversal vs Search

Traversal ← visit every node once

↳ Always O(n)



Search find(E)

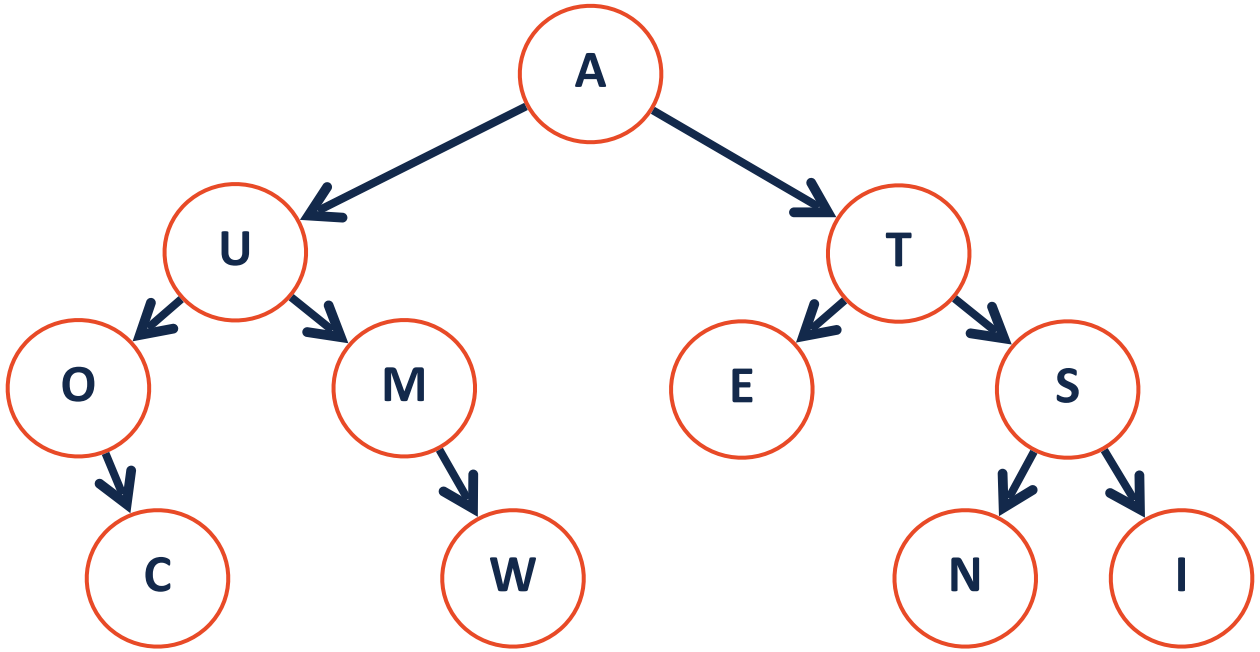
↳ Right now search is traversal O(n)

↳ can be O(h)

↳ up to ↑↑↑??

Tree Search

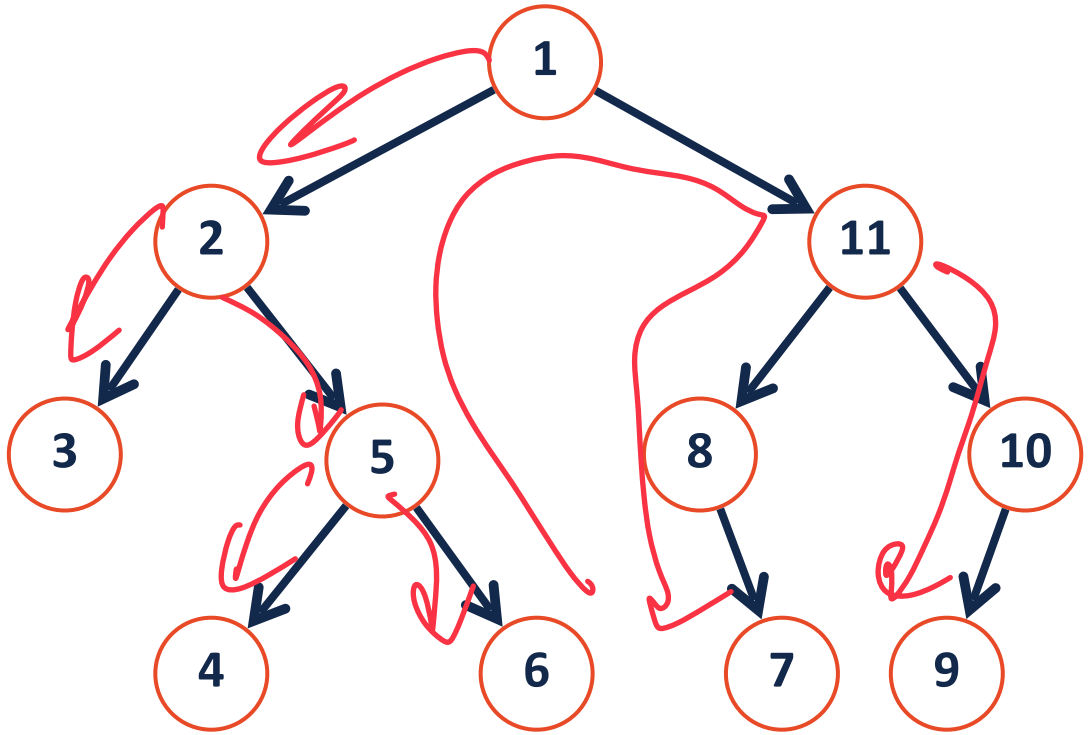
There are two main approaches to searching a binary tree:



Depth First Search

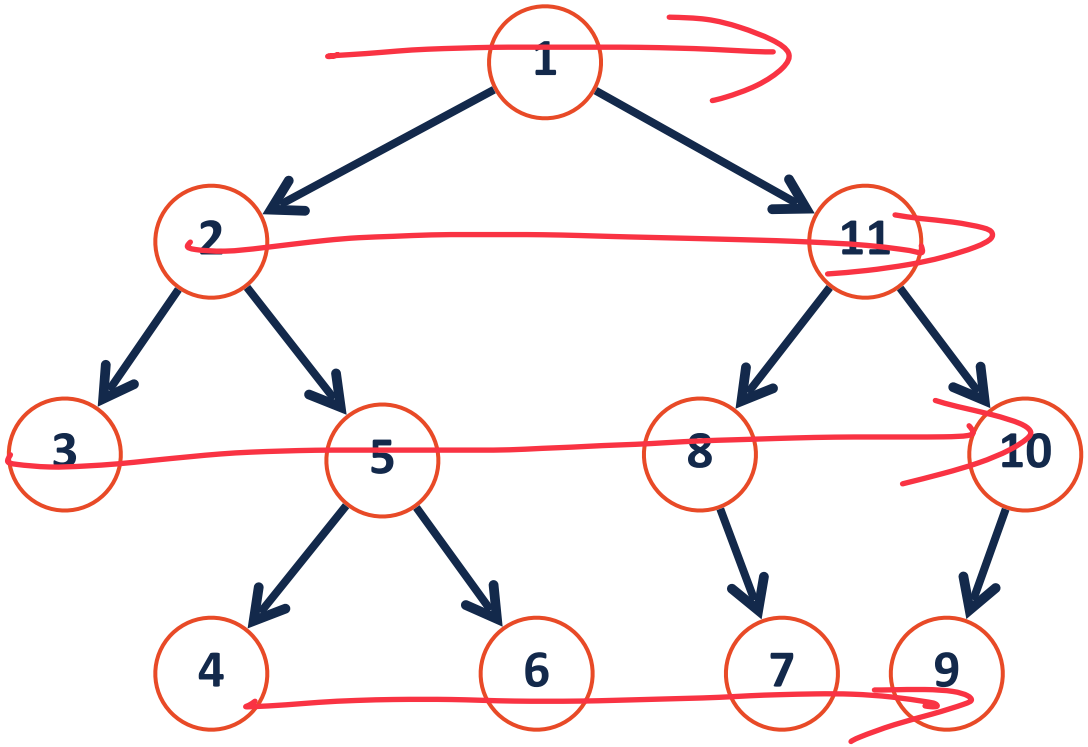
Explore as far along one path as possible before backtracking

This should look familiar

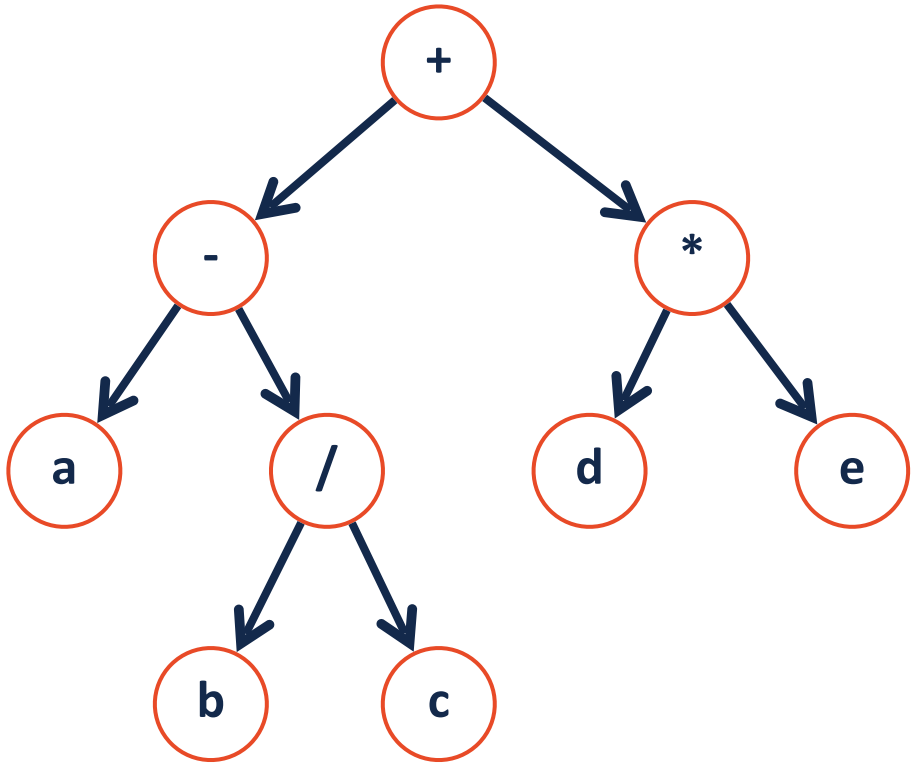


Breadth First Search

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?