ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 8

Trees, Binary Trees

Dayu Wang

- **Overview of Sequential (Linear) Data Structures**

Data Structure	Advantages	Disadvantages	Application
Vector	Fast access by index	Slow search Slow insertion Slow deletion	General purpose
Ordered Vector	Fast access by index Fast search	Slow insertion Slow deletion	General purpose
Linked List	Fast insertion Fast deletion	Slow search	Database General purpose
Stack	Last-in-first-out	Slow access otherwise	Expressions
Sequential Queue	First-in-first-out	Slow access otherwise	Scheduling
Deque	Operations on both ends	Slow access otherwise	General purpose

→ **They suffer from slow search or slow insertion/deletion.**

- **It would be nice to have a data structure…**

→ **that combines the quick addition/deletion of linked lists and the quick search of ordered vectors.**

- **Trees**

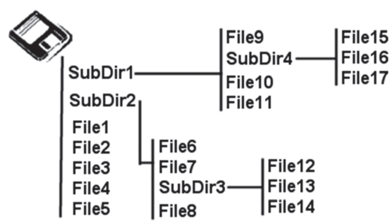
→ Trees are non-linear abstract data structures.

→ Trees support both quick insertion/deletion and quick search.

→ Trees model hierarchical structures.

Family tree

File tree



- **Tree Terminology**

→ A tree consists of nodes and edges.

Nodes represent entities like people, car parts, airline reservations.

The lines (edges) between the nodes represent that they are related.

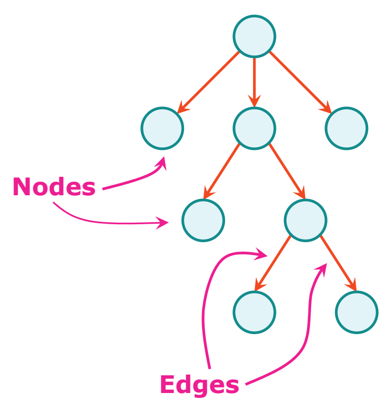
→ A tree is an instance of a general data structure graph.

A graph consists of vertices and edges.

→ Difference between tree and graph

A tree does not have any ~~loop~~ (cycle).

A graph may have loops (cycles).



- **Tree Terminology**

→ The node at the top is called the root of the tree, or root.

A is the root.

→ The successor of a node is called the children of the node.

B C D are A's children.

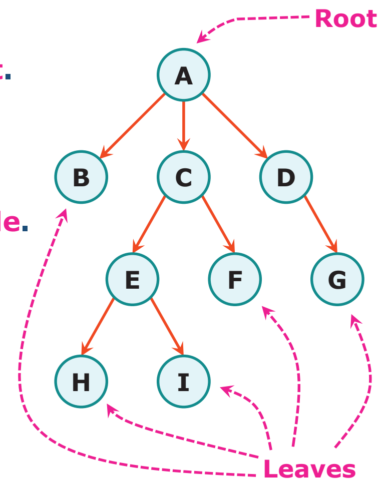
→ The predecessor of a node is called its parent.

C is E and F's parent.

→ Nodes that have the same parent are called siblings.

B and D are siblings; F and G are not ~~siblings~~.

→ A node that has no children is called a leaf node (or leaf). Nodes that have children are called internal nodes.

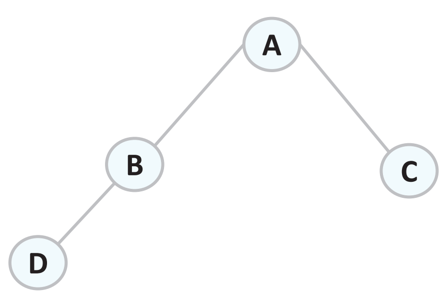


- **Tree Terminology**

→ If node X is a parent of node Y, and Y is a parent of Z, then node X is an ancestor of node Y and Z (e.g., A is an ancestor of B,C, and D).

→ If node Z is a child of node Y, and node Y is a child of node X, then node Z is a descendant of node Y and X (e.g., D is a descendant of A and B).

→ The root node is an ancestor of all other nodes, and all other nodes are descendants of the root node.

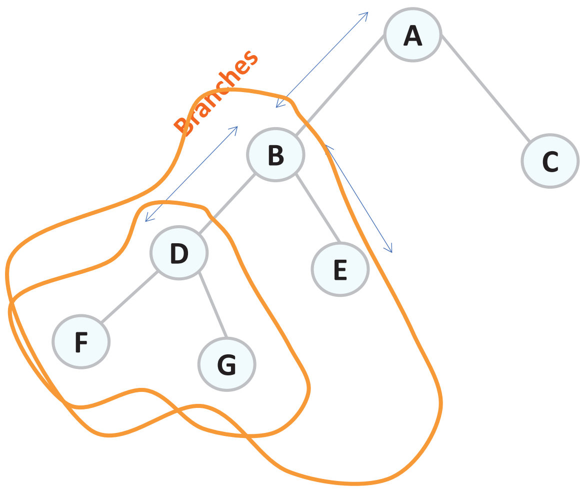


- **Tree Terminology**

  → **Branches are the lines connecting a parent to its children.**

  → **A subtree of a node is a tree whose root is a child of that node.**

  - **[Example 1] Left subtree of node B**
  - **[Example 2] Left subtree of node A**



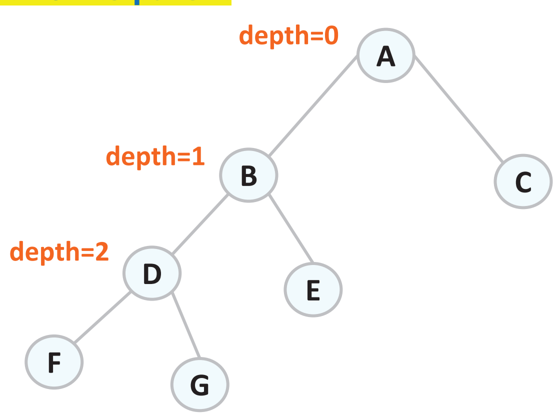
- **Tree Terminology**

  → **The depth of a node is a measure of its distance from the root.**

  The distance between two nodes in a tree is the count of edges (not ~~count of nodes~~) in the path between the two nodes.

  → **A "recursive" algorithm to find the depth of a node N:**

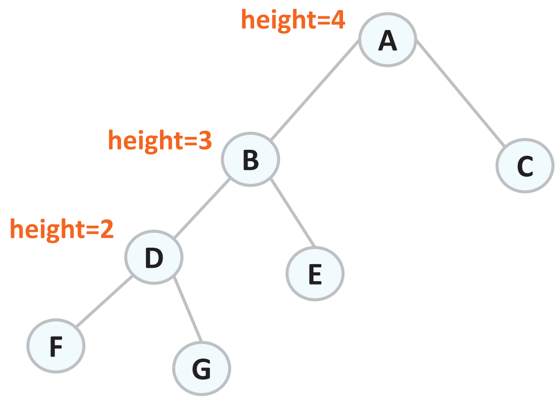
  1) **If N is the root of the tree, then return 0.**
  2) **Otherwise, return 1 + depth of its parent.**



- **Tree Terminology**

→ The **height of a node** is the number of the nodes in the longest path from that node to a leaf node.

→ The **height of a tree** is the same as the height of the root.



- **Binary Trees**

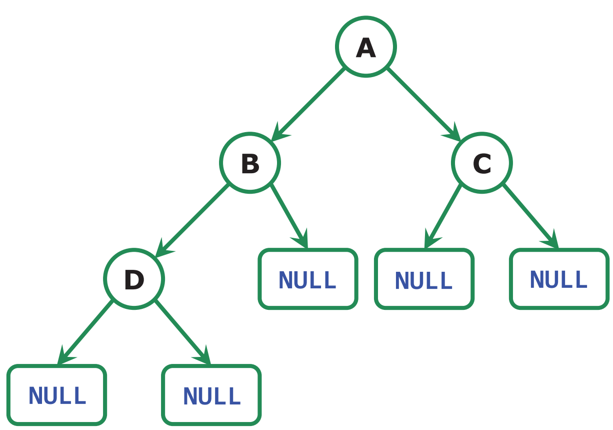
→ In a **binary tree**, each node has at most two branches to subtrees.

- Left branches are called **left subtrees**.
- Right branches are called **right subtrees**.

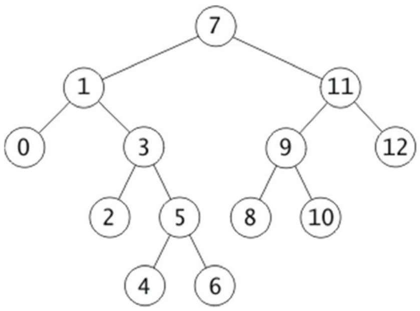
→ Node B has an empty right subtree.

→ Nodes D and C have empty subtrees (leaf nodes).

Empty trees are represented by `NULL` pointers.

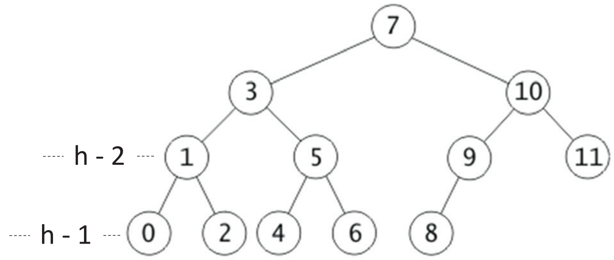


• **Fullness and Completeness**



Full Tree

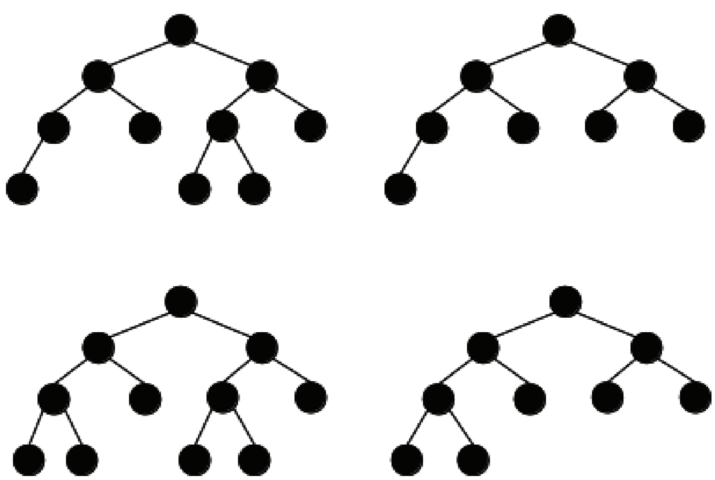
Every node has 0 or 2 non-`NULL` children.



Complete Tree

A complete tree of height h is filled up to depth h − 2, and, at depth h − 1, any unfilled nodes are on the right.

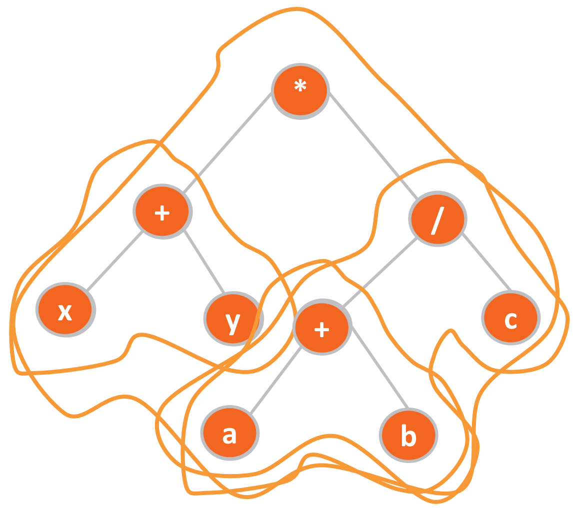
• **Fullness and Completeness**



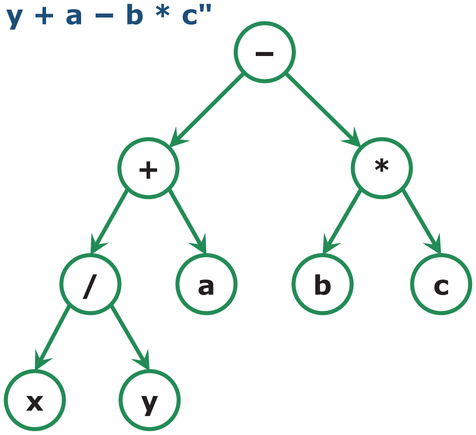
Resource: http://gsourcecode.files.wordpress.com/2012/02/complete-full-trees1.png

- **Expression Trees**
  - → Each node is an operator or an operand.
    - The operands are leaf nodes.
    - The operators are internal nodes.
  - → What is the expression that this tree represents?

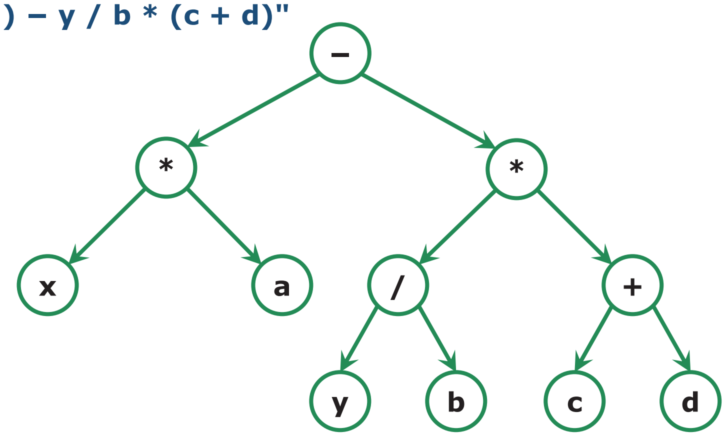
((x + y) * ((a + b) / c))



- **Exercise**
  - → Draw binary expression trees for the following expressions:
    1) "x / y + a − b * c"
    2) "(x * a) − y / b * (c + d)"
  - → Procedure
    1) Find out which operator will be evaluated last. That operator will be the root.
    2) Apply the same logic to left and right subtree.
  - → "x / y + a − b * c"

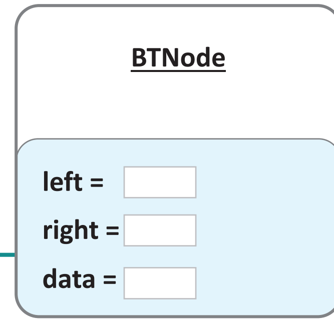


- → "(x * a) − y / b * (c + d)"



- **Implementing Binary Trees**

A binary tree node

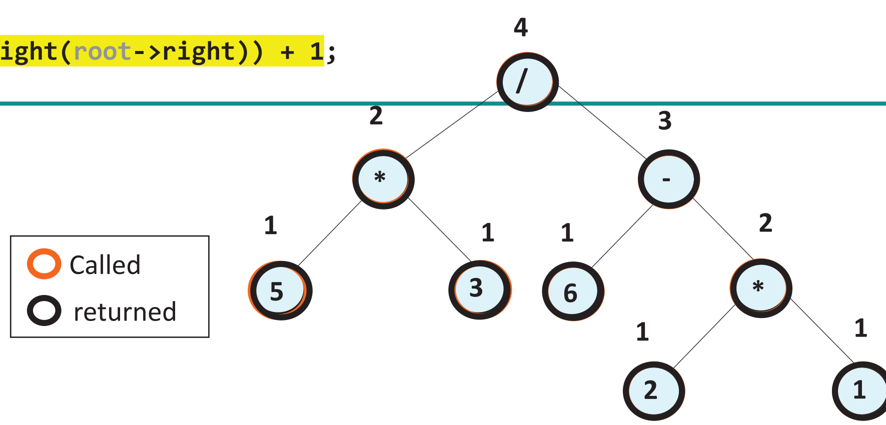



```
/** A binary tree node */
template<class T>
struct BTNode {
    // Data fields
    T data;  // Stores some data in the node.
    BTNode<T> *left, *right;  // Stores pointers to the left and right child.

    // Constructor
    BTNode(const T&, BTNode<T>* = NULL, BTNode<T>* = NULL);

    // Destructor
    virtual ~BTNode();  // Avoid warning messages.

    // Class-member function
    virtual string to_string() const;  // Returns a string containing the data stored in the node.
};
```


- **Exercise**

→ **Write a function that calculates the height of the a binary tree.**

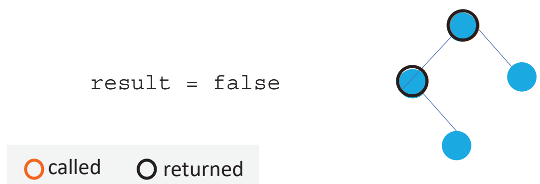
```
/** Calculates the height of a binary tree.
    @param root: a pointer to the root node of the binary tree
    @return: calculated height of the binary tree
*/
template<class T>
unsigned int height(const BTNode<T>* root) {
    // Base case
    if (!root) { return 0; }
    // Recurrence relation
    return max(height(root->left), height(root->right)) + 1;
}
```



- **Exercise**

→ **Please write a function that tests whether a binary tree is full.**

```
/** Tests whether a binary tree is full.
    @param root: a pointer to the root node of the binary tree
    @return: {true} if the binary tree is full; {false} otherwise
*/
template<class T>
bool is_full(const BTNode<T>* root) {
    // Base case
    if (!root) { return true; }

    if ((bool)root->left + (bool)root->right == 1) { return false; }

    // Recurrence relation
    return is_full(root->left) && is_full(root->right);
}  // Time complexity: O(n)
```



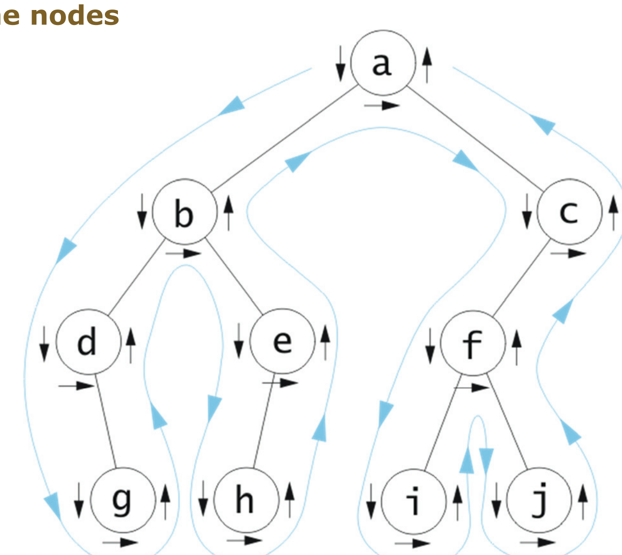
- **Binary Tree Traversals**

→ **Often we want to determine the nodes of a tree and their relationship.**
We can do this by walking through the tree and visiting the nodes.
Visiting the nodes: processing the information in the nodes

→ **This process is called traversal.**

→ **We will discuss three kinds of traversal algorithms:**

1) **Preorder traversal**
2) **Inorder traversal**
3) **Postorder traversal**



- **Preorder Traversal**

→ **Algorithm**

If the tree is empty, return.

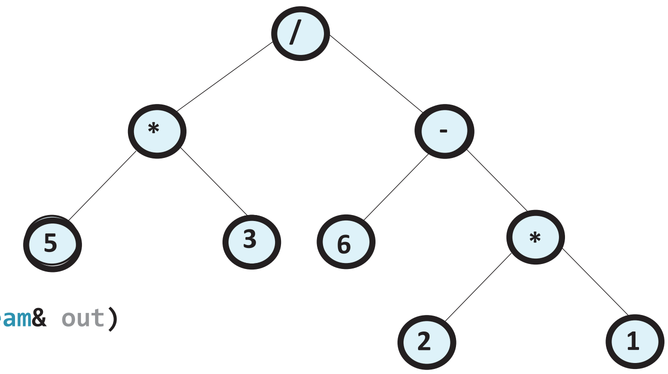
Else

Visit the root.

Preorder traverse the left subtree (recursive).

Preorder traverse the right subtree (recursive).

→ **Preorder traversing an expression tree will generate a prefix expression.**




```
// Preorder traversal
template<class T>
void preorder_traversal(const BTNode<T>* root, ostream& out)
{
    if (root) {
        out << root << ' ';
        preorder_traversal(root->left, out);
        preorder_traversal(root->right, out);
    }
}  // Time complexity: O(n)
```

Prefix Expression

/ * 5 3 - 6 * 2 1

- **Inorder Traversal**

→ **Algorithm**

If the tree is empty, return.

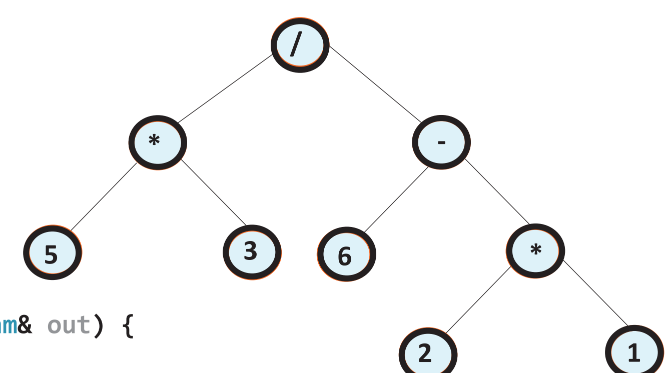
Else

Inorder traverse the left subtree (recursive).

Visit the root.

Inorder traverse the right subtree (recursive).

→ **Inorder traversing an expression tree will generate an infix expression.**




```
// Inorder traversal
template<class T>
void inorder_traversal(const BTNode<T>* root, ostream& out) {
    if (root) {
        out << " ( ";
        inorder_traversal(root->left, out);
        out << ' ' << root << ' ';
        inorder_traversal(root->right, out);
        out << " ) ";
    }
}  // Time complexity: O(n)
```

Called

Visited

Infix Expression

((5 * 3) / (6 - (2 * 1)))

- **Postorder Traversal**

→ **Algorithm**

If the tree is empty, return.

Else

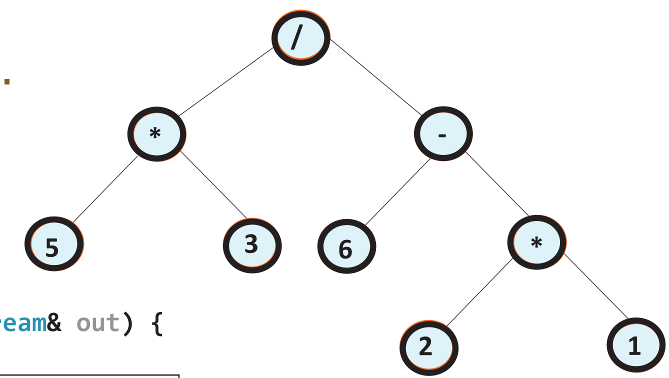
Postorder traverse the left subtree (recursive).

Postorder traverse the right subtree (recursive).

Visit the root.

→ **Postorder traversing an expression tree will generate a postfix expression.**

```
// Postorder traversal
template<class T>
void postorder_traversal(const BTNode<T>* root, ostream& out) {
    if (root) {
        postorder_traversal(root->left, out);
        postorder_traversal(root->right, out);
        out << root << ' ';
    }
}  // Time complexity: O(n)
```



Postfix Expression

5 3 * 6 2 1 * - /

- **Three Binary Tree Traversal Algorithms**

Preorder	Inorder	Postorder
If the tree is empty Return Else Visit the root Traverse the left subtree Traverse the right subtree	If the tree is empty Return Else Traverse the left subtree Visit the root Traverse the right subtree	If the tree is empty Return Else Traverse the left subtree Traverse the right subtree Visit the root