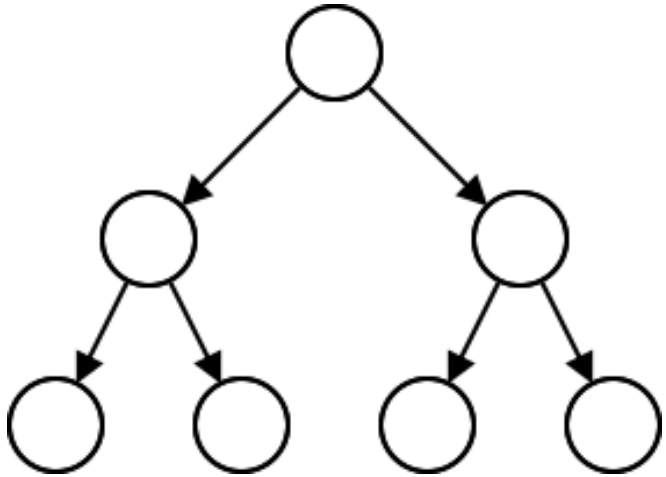
Binary Trees

L.EIC

Algorithms and Data Structures

2026/2027



P Ribeiro, AP Tomas

Non-linear Structures

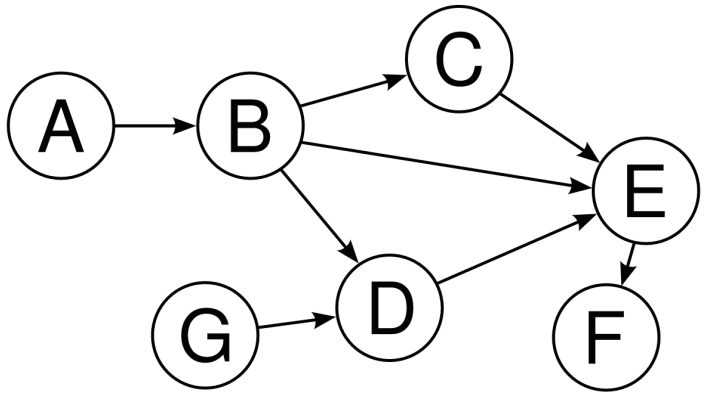
Arrays and lists are examples of **linear** data structures.

Each element has:

- a single predecessor (except for the first element of the list);
- a single successor (except for the last element of the list).

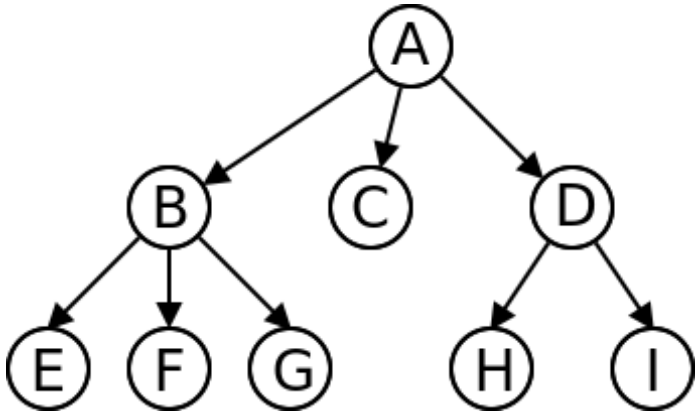
Are there other types of structures?

- A **graph** is a **non-linear** data structure, as each of its elements, called nodes, can have more than one predecessor or more than one successor.



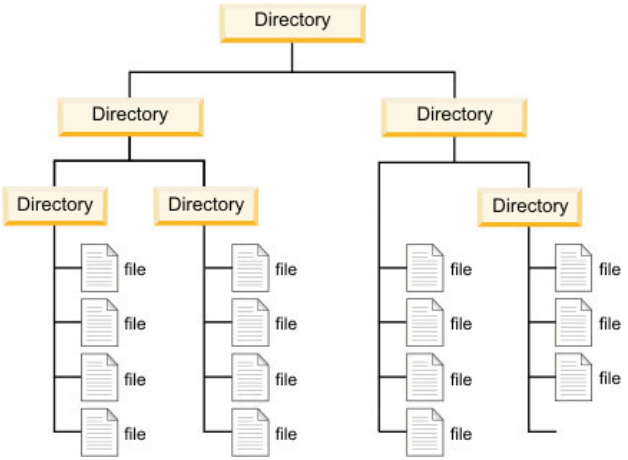
Trees

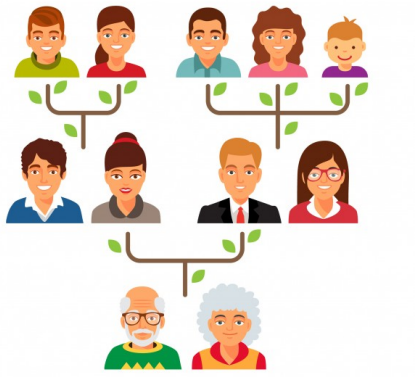
- A **rooted tree** (here simply called a **tree**) is a specific type of graph.
  (in the Graphs chapter we will also see trees without a root)
- Each element, called a **node**, has zero or more successors, but only one predecessor (except for the first node, known as the **root** of the tree, which has none).
- Every node can be reached from the root by following successors
  (this guarantees there are no cycles)
- An example of a tree:

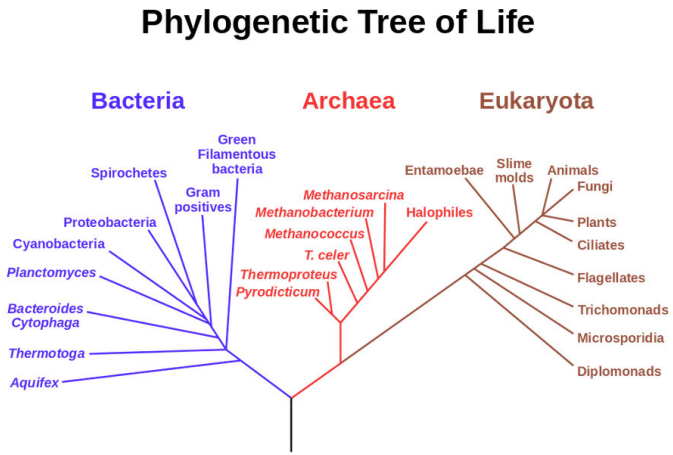


Trees - Examples

- Trees are particularly well-suited for representing information organized in **hierarchies**.
- Some examples:
  - The directory structure (or folders) of a file system
  - A family tree
  - A tree of life

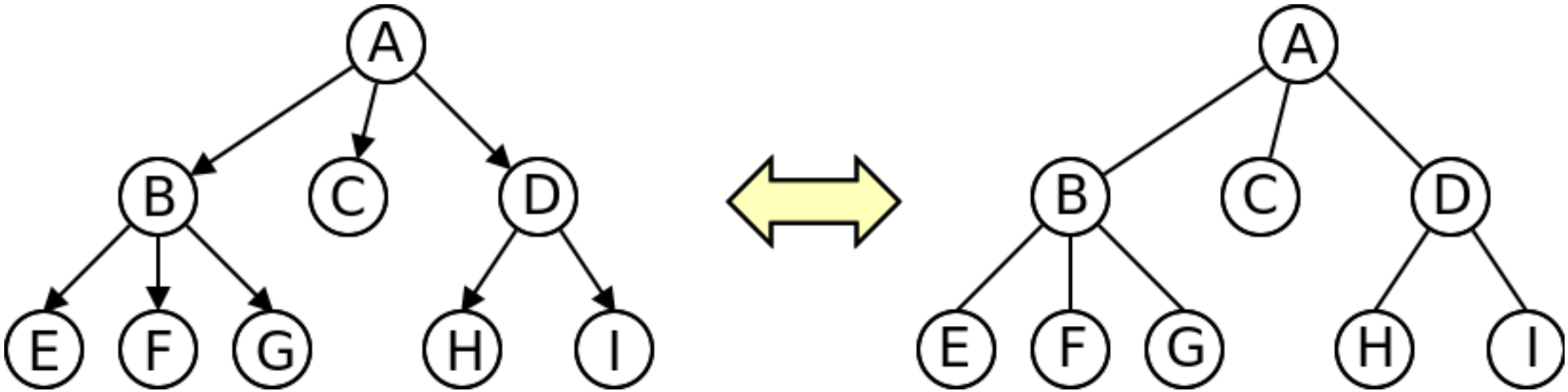




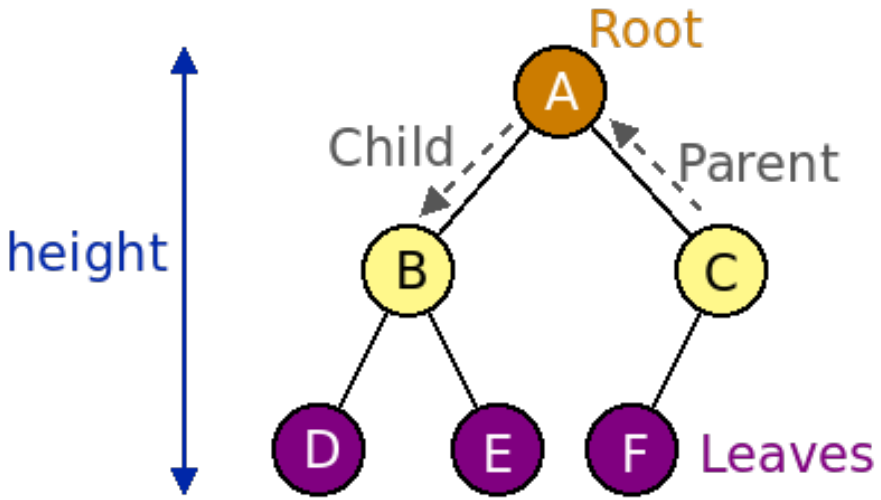


Trees - Visualization

- Often, arrows are omitted in the edges (or connections) since it is clear from the diagram which nodes descend from which:

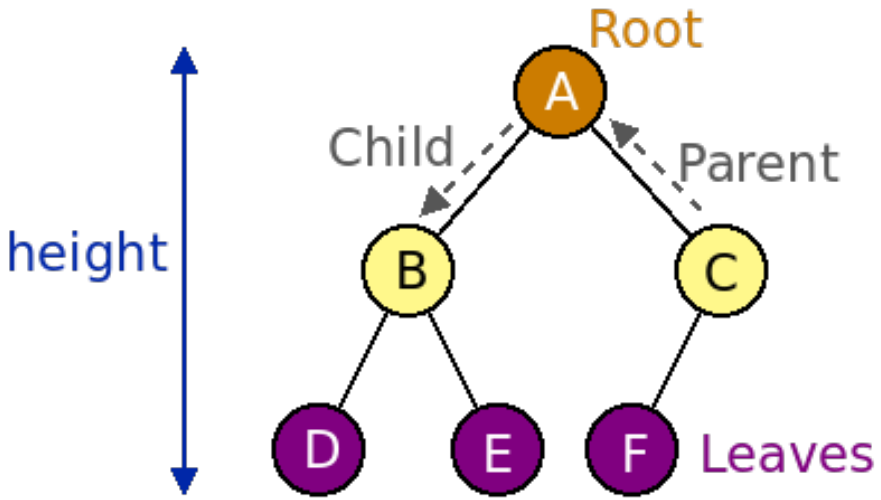


Trees - Terminology



- The unique predecessor of a node is called its **parent**.
  - *Example: The parent of B is A; the parent of C is also A*
- The successors of a node are its **children**.
  - *Example: The children of A are B and C*
- The **degree** of a node is the number of its children.
  - *Example: A has 2 children, C has 1 child*
- A **leaf** is a node without children, i.e., with degree 0.
  - *Example: D, E, and F are leaf nodes*
- The **root** is the only node without a parent.
- The **subtree** of a node is the node plus all its **descendants**.
  - *Example: the subtree of B is {B,D,E}*

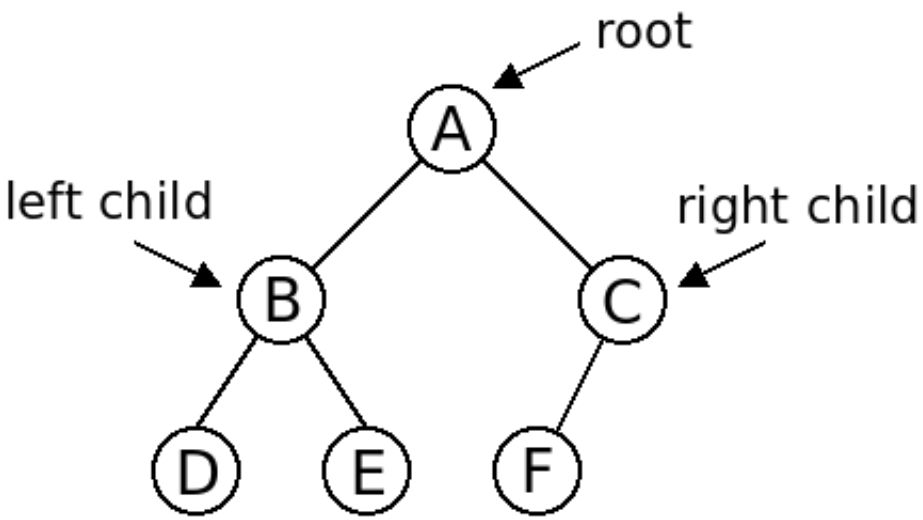
Trees - Terminology



- The edges that connect the nodes are called **branches**.
- A **path** is the sequence of branches between two nodes.
  - *Example: A-B-D is the path between A and D*
- The **length** of a path is the number of branches it contains.
  - *Example: A-B-D has a length of 2*
- The **depth** of a node is the length of the path from the root to that node (the depth of the root is zero).
  - *Example: B has depth 1, D has depth 2*
- The **height** of a tree is the maximum depth of any node in the tree.
  - *Example: The tree in the diagram has a height of 2*

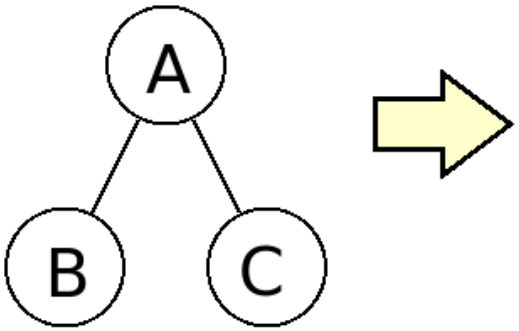
Binary Trees

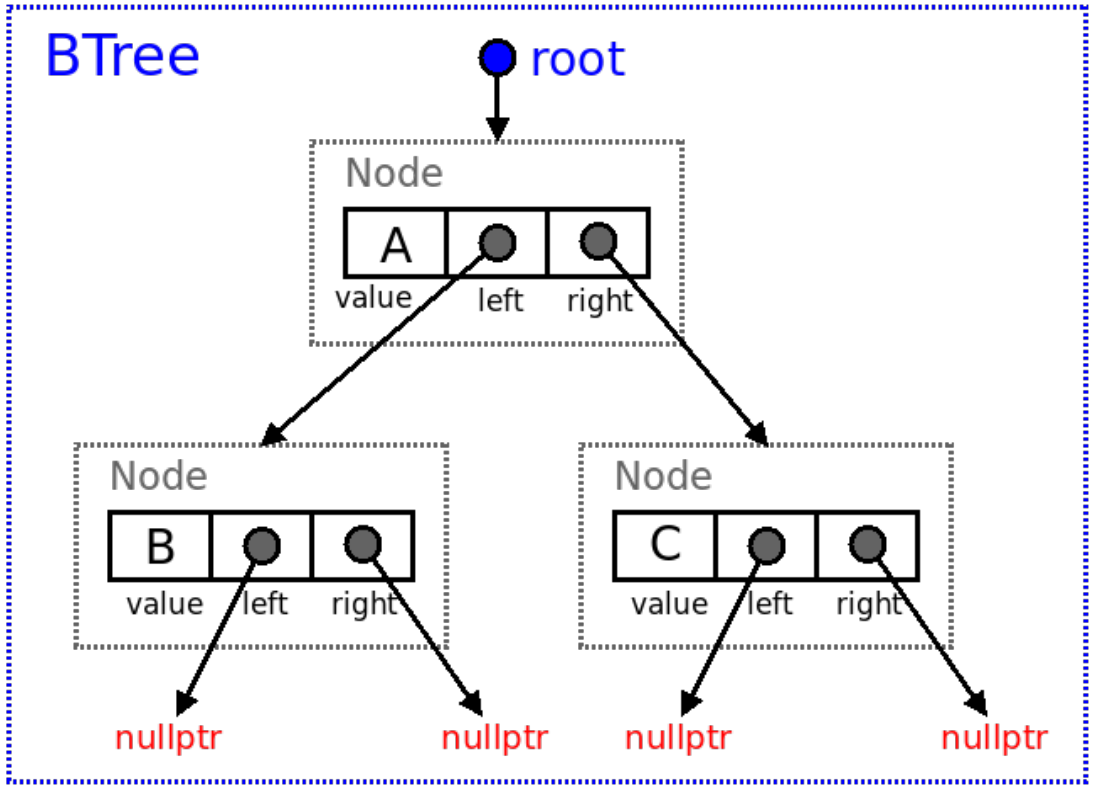
- The **arity** of a tree is the maximum degree of a node.
- A **binary tree** is a tree with arity **at most** 2, meaning each node has at most two children, called the **left** child and the **right** child.



Implementation

- Just as a linked list has a reference to its first node, a tree has a reference to its **root node**
- Each node should contain the **value** and a reference to the **left** and **right** child
- Leaves should contain pointers to nullptr





Binary Trees - Implementation

- Let's put this to practice with a **generic** binary tree:
  values can be integers, strings, or any other type of object
- For the purposes of this class we will be using a **lightweight** class,
  with only the essentials *(focus more on the algorithms)*

```
template <class T> class BTree {
private:
  struct Node {              // A tree node is just a simple private struct
    T value;
    Node *left, *right;
  };

  Node *root;                // the root is just a pointer to a node

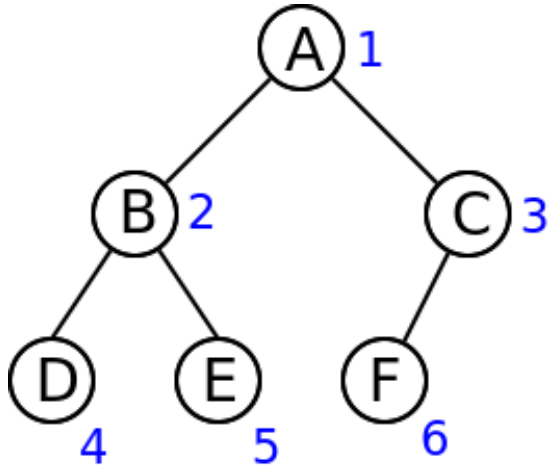
public:
  BTree() {root = nullptr;} // the constructor starts with an empty tree
};
```

Binary Trees - A Recursive Definition

- A **binary tree** is either:
  - **empty** *(in code: nullptr)*, or
  - a **root** node, plus a **left** binary tree and a **right** binary tree
- Left and right are *positions*: a root with only a left child is a different tree from a root with only a right child.
- The definition uses itself: the left and right **subtrees** are binary trees too!
- This is why most tree methods are naturally **recursive**:
  - **Base case:** the empty tree (*nullptr*)
  - **Recursive case:** handle the root, and call the method on the left and right subtrees

Binary Trees - Number of Nodes

- Let's now create some **methods** to add to the `BTree<T>` class.
- The first method's goal is to **count the number of nodes** in a tree. For example, the following binary tree has 6 nodes:



- Let's create a **recursive method**
  - **Base case:** when the tree is empty... it has 0 nodes!
  - **Recursive case:** the number of nodes in a non-empty tree is equal to 1 + nr of nodes in the left subtree + nr of nodes in the right subtree.
    - ★ *Fig.: num_nodes = 1 + num_nodes({B,D,E}) + num_nodes({C,F})*

Binary Trees - Number of Nodes

- We need to start counting... from the root!
- We want to have a `numberNodes()` method in the `BTree<T>` class.
  - *Example: if t is a tree, we want to be able to call t.numberNodes()*
- We'll use a **helper method** that is recursive.
  - *Methods that do not change the tree are* const*, and helpers should ideally be* private *(users would only see `numberNodes()`)*

```
// Main method - returns the total number of nodes in the tree
int numberNodes() const {
  return numberNodes(root);
}

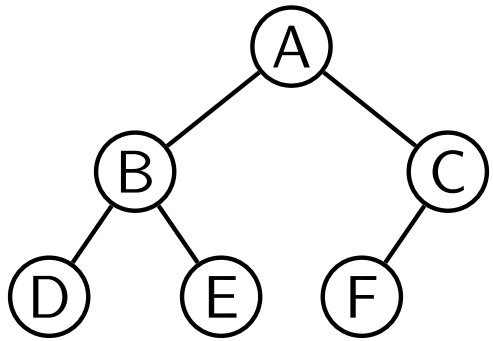
// Helper method (recursive)
int numberNodes(Node *n) const {
  if (n == nullptr) return 0;
  return 1 + numberNodes(n->left) + numberNodes(n->right);
}
```

- This **pattern** *(main method that calls a recursive helper method from the root)* can be used for many kinds of tasks.
- Let's look at a few more examples...

Binary Trees - Live Coding



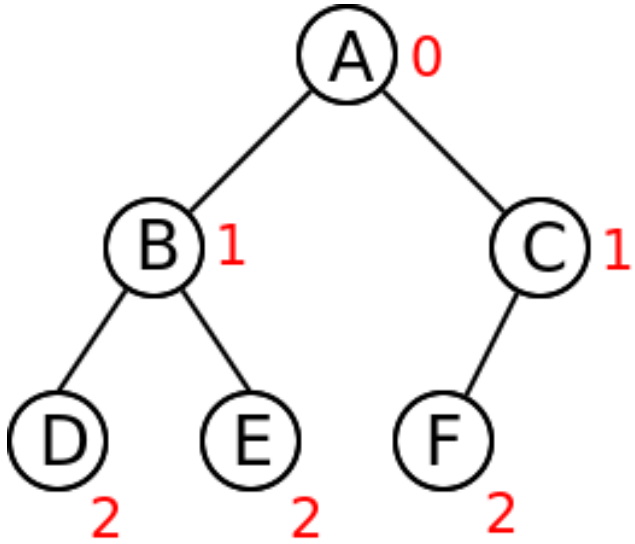
- We'll implement **two methods** together: `height()` and `contains()`
  (in the practical class you'll write many more)
- Starting point: a file with what we already have:
  - the `BTree<T>` class with `Node`, `root` and `numberNodes()`
  - a `read()` method, so we don't build the tree by hand
    (we'll see how it works later today; for now, it just gives us our usual tree:)



Binary Trees - Tree Height

LIVE

- **Problem:** calculate the **height of a tree** (maximum depth of a node).
- Each node's depth is shown in red:



- For this tree, `height()` should return **2**.

Binary Trees - Tree Height

- A **recursive method** very similar to the previous one:
  - **Recursive case:** the height of a tree is 1 plus the maximum between the heights of the left and right subtrees.
    - ★ *Fig.: height = 1 + max(height({B,D,E}), height({C,F}))*
- What should be the **base case**? Two options:
  - We can stop at a leaf: height 0 *(but one-child nodes need extra checks)*
  - If we stop at a *nullptr* tree, the height should be... **-1**
    - ★ *E.g.: height of a 1-node tree = 1 + max(-1, -1) = 0*

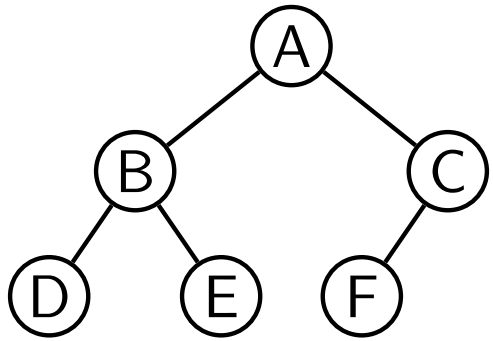
```
// Returns the height of the tree
int height() const {
  return height(root);
}

// Helper method (recursive)
int height(Node *n) const {
  if (n == nullptr) return -1;
  return 1 + std::max(height(n->left), height(n->right));
}
```

Binary Trees - Searching for an Element

LIVE

- **Problem:** check if an element is **contained** in a tree.



- For this tree:
  - `contains('E')` should return **true**
  - `contains('G')` should return **false**

Binary Trees - Searching for an Element

- A **recursive method** very similar to the previous ones:
  - **Base case 1:** if the current node is *nullptr*, it doesn't contain the value, so we return *false*.
  - **Base case 2:** if the value is in the current node, we return *true*.
  - **Recursive case:** otherwise, we check the left subtree **OR** the right subtree.

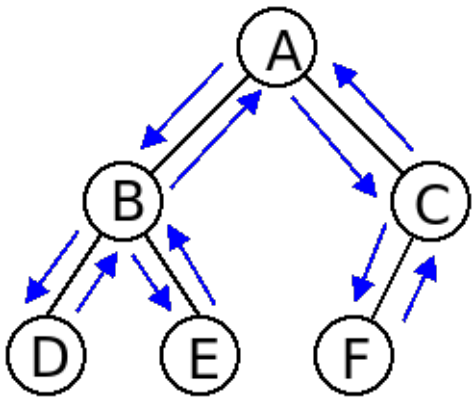
```
// Returns true if val is in the tree, or false otherwise
bool contains(const T & val) const {
  return contains(root, val);
}

// Helper method (recursive)
bool contains(Node *n, const T & val) const {
  if (n == nullptr) return false;
  if (n->value == val) return true;
  return contains(n->left, val) || contains(n->right, val);
}
```

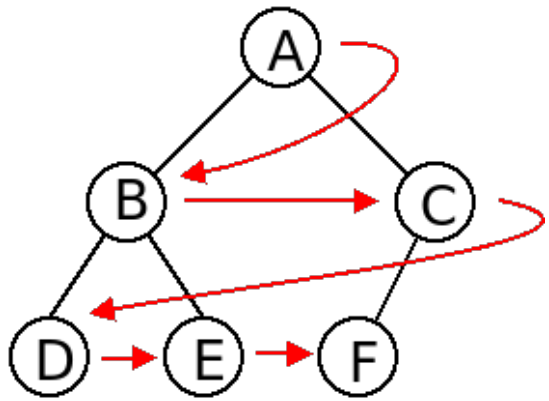
- || is **short-circuit**: if found on the left, the right is **not** searched.

Binary Trees - Writing the Nodes of a Tree

- How can we **write the content (nodes)** of a tree?
- We need to **traverse all nodes**. But in what **order**?
- Let's distinguish between two different orders:



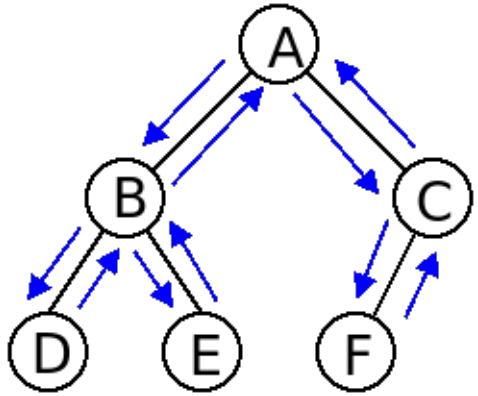
Depth-First Search



Breadth-First Search

- **Depth-First Search** (DFS): visit all nodes in the subtree of one child before visiting the subtree of the other child.
- **Breadth-First Search** (BFS): visit nodes in increasing depth order.

Binary Trees - Depth-First Search



Depth-First Search

- If we write a node the first time we pass through it, we get the following for the figure: **A B D E C F**
- This corresponds to doing the following:
  1. Write the root
  2. Write the entire left subtree
  3. Write the entire right subtree
- This can be directly converted into a **recursive method**!

Binary Trees - Depth-First Search

- Translating what was described on the previous slide into code:

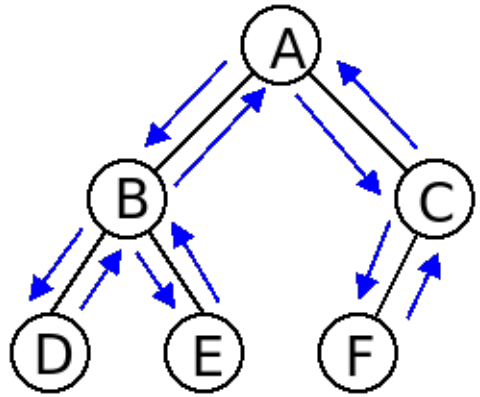
```
// Write all nodes in PreOrder
void printPreOrder() const {
  std::cout << "PreOrder:";
  printPreOrder(root);
  std::cout << std::endl;
}

// Helper method (recursive)
void printPreOrder(Node *n) const {
  if (n == nullptr) return;
  std::cout << " " << n->value;
  printPreOrder(n->left);
  printPreOrder(n->right);
}
```

- For the previous tree, this would output *"PreOrder: A B D E C F"*.
- We call this order **PreOrder**, because we write the root before the two subtrees.

Binary Trees - Depth-First Search

- Besides **PreOrder**, we can also consider two other depth-first orders:
  - **InOrder:** root written *between* the two subtrees.
  - **PostOrder:** root written *after* the two subtrees.



Depth-First Search

- For the tree in the figure:
  - **PreOrder:** A B D E C F (1st time we pass a node)
  - **InOrder:** D B E A F C (2nd time: coming back from the left)
  - **PostOrder:** D E B F C A (last time: leaving the node)

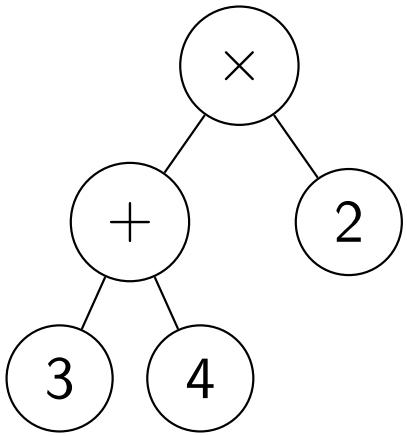
Binary Trees - Depth-First Search

- Implementing *InOrder* and *PostOrder*:
  - the main methods are the same as in PreOrder (only the label changes);
  - in the helper methods, only the position of the print moves.

```
// InOrder:  left, root, right
void printInOrder(Node *n) const {
  if (n == nullptr) return;
  printInOrder(n->left);
  std::cout << " " << n->value;
  printInOrder(n->right);
}
```

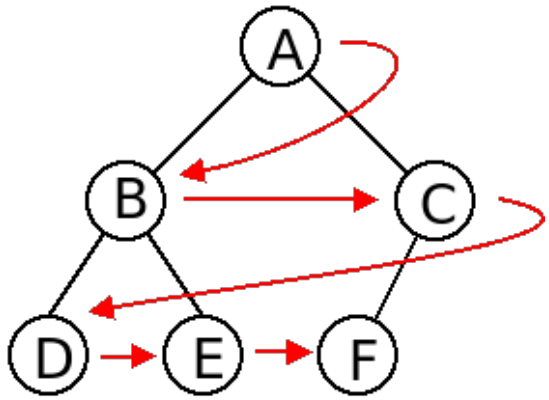
```
// PostOrder:  left, right, root
void printPostOrder(Node *n) const {
  if (n == nullptr) return;
  printPostOrder(n->left);
  printPostOrder(n->right);
  std::cout << " " << n->value;
}
```

Binary Trees - Why Three Orders?



- An **expression tree**: operators in internal nodes, numbers in the leaves. This one represents $(3 + 4) \times 2$.
- **PreOrder:** $\times \; + \; 3 \; 4 \; 2$
  prefix notation, like function calls: `mul(add(3,4),2)`
- **InOrder:** $3 + 4 \times 2$
  the usual *infix* notation... but it needs parentheses to be correct!
- **PostOrder:** $3 \; 4 \; + \; 2 \; \times$
  postfix notation: evaluated with a **stack** (no parentheses needed)

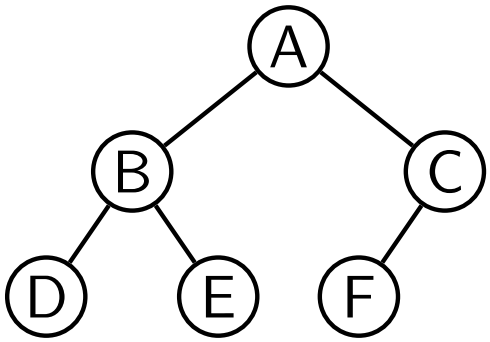
Binary Trees - Breadth-First Search



Breadth-First Search

- To traverse in breadth-first order, we'll use a **queue** ADT.
  1. Initialize a queue Q by adding the root.
  2. While Q is not empty:
  3. Dequeue the first element, *cur*, from the queue.
  4. Write *cur*.
  5. Add the children of *cur* to the end of the queue.

Binary Trees - Breadth-First Search



step	print	Q (front on the left)
start		A
1	A	B C
2	B	C D E
3	C	D E F
4	D	E F
5	E	F
6	F	(empty)

- Each step: dequeue, print, enqueue the children.
- Output: `A B C D E F` (level by level).

Binary Trees - Breadth-First Search in Code

- Implementing in code:

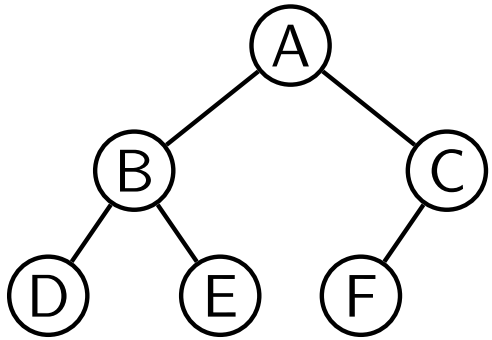
```
// Write all nodes in BFS order
void printBFS() const {
  std::cout << "BFS:";
  std::queue<Node *> q;
  q.push(root);
  while (!q.empty()) {
    Node *cur = q.front();
    q.pop();
    if (cur != nullptr) {
      std::cout << " " << cur->value;
      q.push(cur->left);
      q.push(cur->right);
    }
  }
  std::cout << std::endl;
}
```

- In this version, we allow *null* nodes to enter the queue, but then ignore them. Alternatively, we could only add non-null nodes.

Binary Trees - BFS vs DFS

FOLLOW

- If instead of a **queue** Q (*FIFO*) we used a **stack** S (*LIFO*), we would be performing a DFS instead of BFS!



step	print	S (top on the right)
start		A
1	A	C B
2	B	C E D
3	D	C E
4	E	C
5	C	F
6	F	(empty)

- Each step: pop, print, push the children (**right first**).
- Output: `A B D E C F` (same as PreOrder).

Binary Trees - Depth-First Search in Code

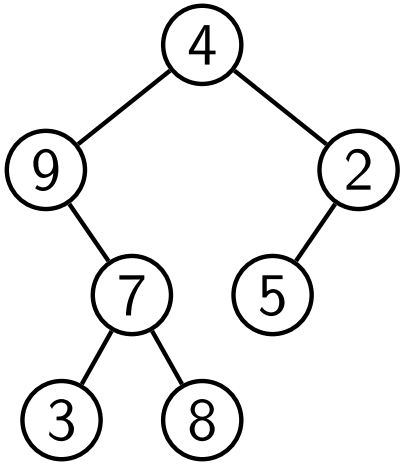
- Implementing in code:

```
// Write all nodes in DFS order
void printDFS() const {
  std::cout << "DFS:";
  std::stack<Node *> s;
  s.push(root);
  while (!s.empty()) {
    Node *cur = s.top();
    s.pop();
    if (cur != nullptr) {
      std::cout << " " << cur->value;
      s.push(cur->right);   // right first, so that left is on top
      s.push(cur->left);
    }
  }
  std::cout << std::endl;
}
```

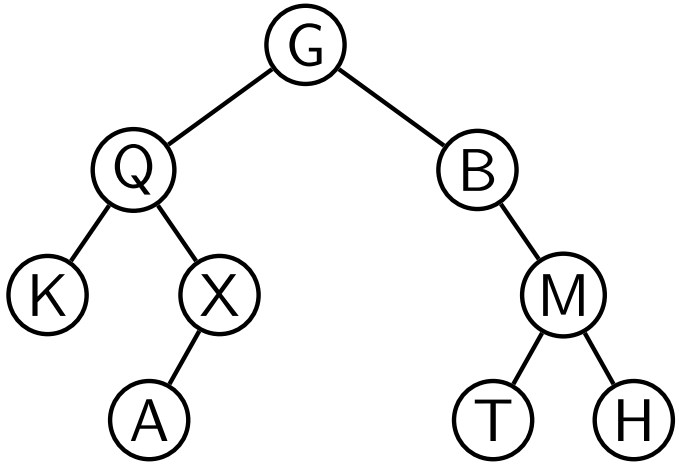
- Pushing the right child first makes the output match **PreOrder**.
- The recursive version does exactly this *implicitly*: it uses the **call stack**!

Binary Trees - Your Turn!

- Write the **PreOrder**, **InOrder**, **PostOrder** and **BFS** of each tree:



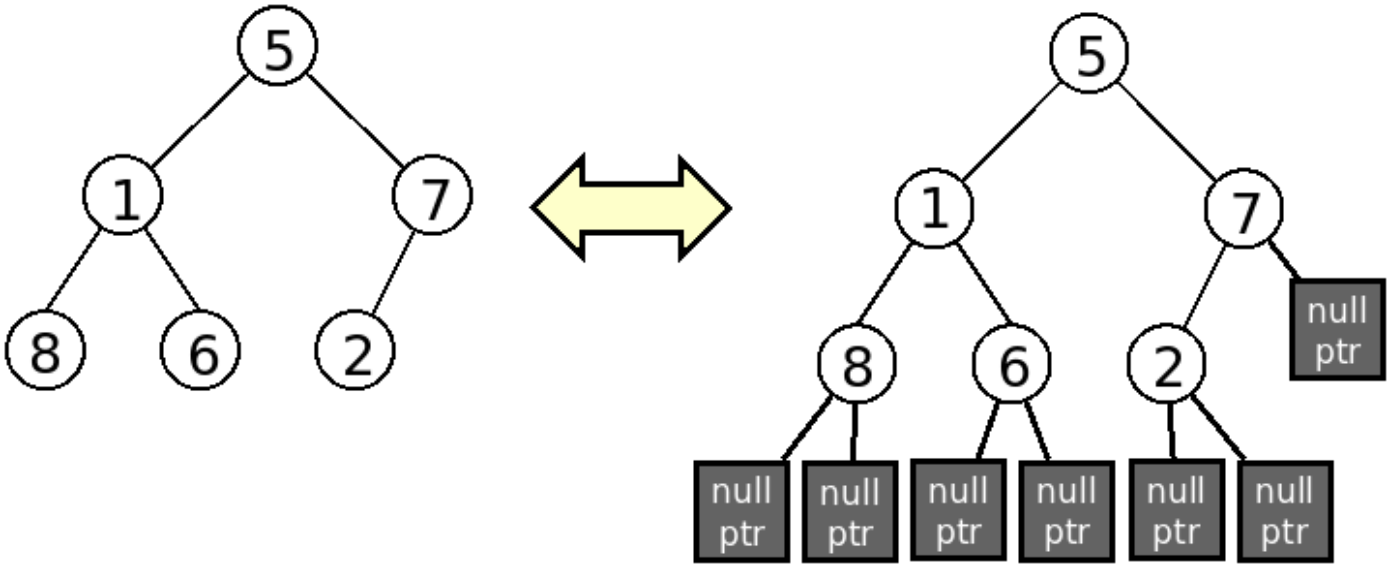
Pre: 4 9 7 3 8 2 5
In: 9 3 7 8 4 5 2
Post: 3 8 7 9 5 2 4
BFS: 4 9 2 7 5 3 8



Pre: G Q K X A B M T H
In: K Q A X G B T M H
Post: K A X Q T H M B G
BFS: G Q B K X M A T H

Binary Trees - Reading a Tree in PreOrder

- How can we **read a tree**?
- One approach is to use **PreOrder**, explicitly representing *nullptrs*.
- Note that the following two representations refer to the same tree:

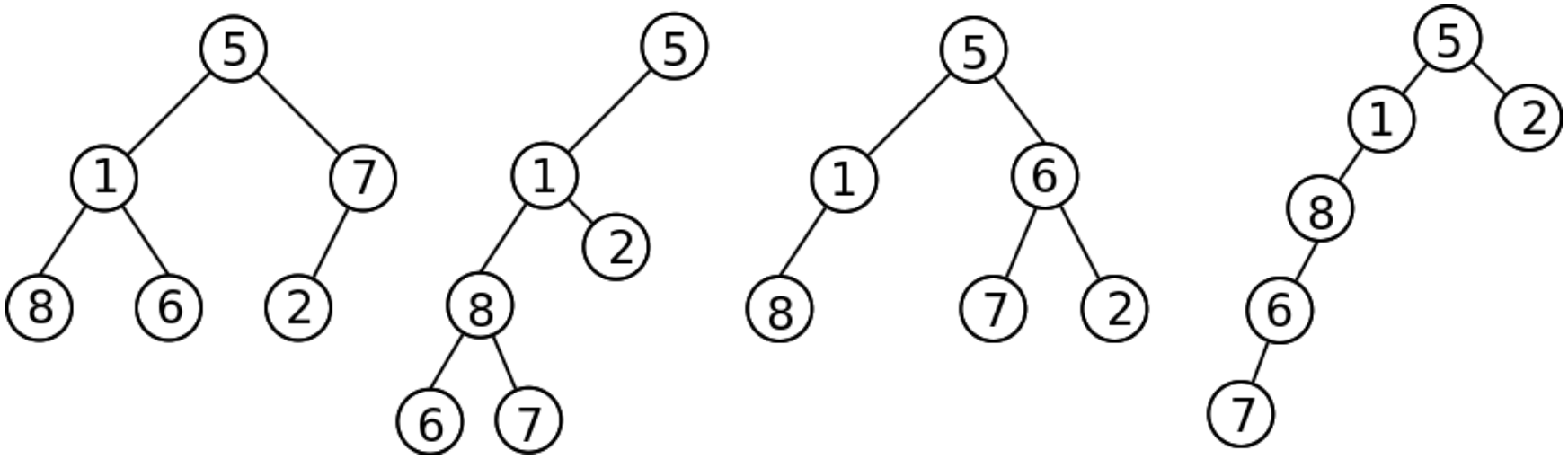


- If we represent *nullptr* as **N**, then the tree in *PreOrder* would be represented as:

```
5 1 8 N N 6 N N 7 2 N N N
```

Binary Trees - Reading a Tree in PreOrder

- Note the necessity of including *nullptrs*.
- Example: without *nullptrs*, the following preorder representation could refer to any of the four trees (among others): `5 1 8 6 7 2`



- With *nullptrs*, these four trees become distinguishable:
  - 1st Tree: `5 1 8 N N 6 N N 7 2 N N N`
  - 2nd Tree: `5 1 8 6 N N 7 N N 2 N N N`
  - 3rd Tree: `5 1 8 N N N 6 7 N N 2 N N`
  - 4th Tree: `5 1 8 6 7 N N N N N 2 N N`

Binary Trees - Why PreOrder?

- Could we use **InOrder** with **N** instead?
- **No!** `N 1 N 2 N` is the InOrder of two different trees:



- In PreOrder these are `1 N 2 N N` and `2 1 N N N`: different!
- PreOrder starts with the **root**, so the reader always knows which node comes next and where it goes (this is what makes `read()` recursive and simple).

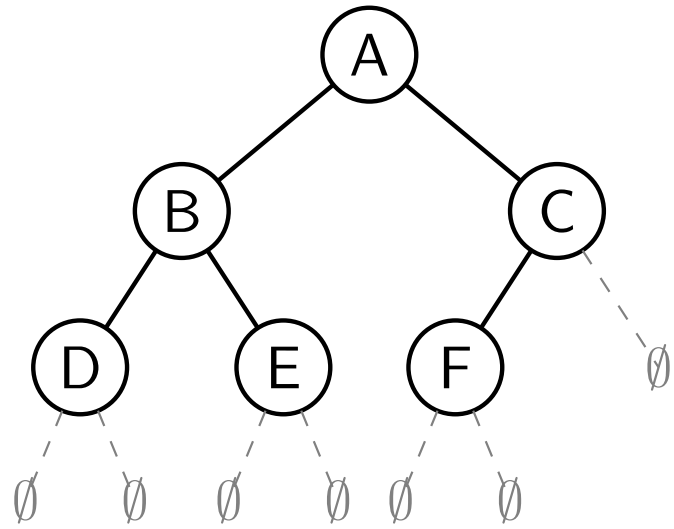
(writing a tree to a file and reading it back is called **serialization**)

Binary Trees - Reading a Tree in PreOrder

FOLLOW

A	B	D	N	N	E	N	N	C	F	N	N	N

```
void read(std::string null) {
  root = readNode(null);
}
Node *readNode(std::string null) {
  std::string buffer;
  std::cin >> buffer;
  if (buffer == null) return nullptr;
  Node *n = new Node;
  std::istringstream ss(buffer);
  ss >> n->value;
  n->left = readNode(null);
  n->right = readNode(null);
  return n;
}
```



(`null`: string used for *nullptr*, e.g. `read("N")`; assumes well-formed input)

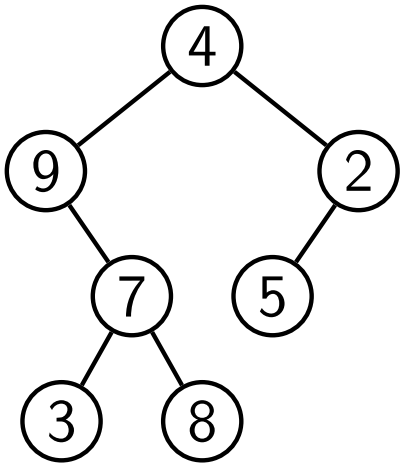
- Each call reads **one token**: a value creates a node and then reads its left and right subtrees; **N** returns `nullptr`.

Binary Trees - Serialization

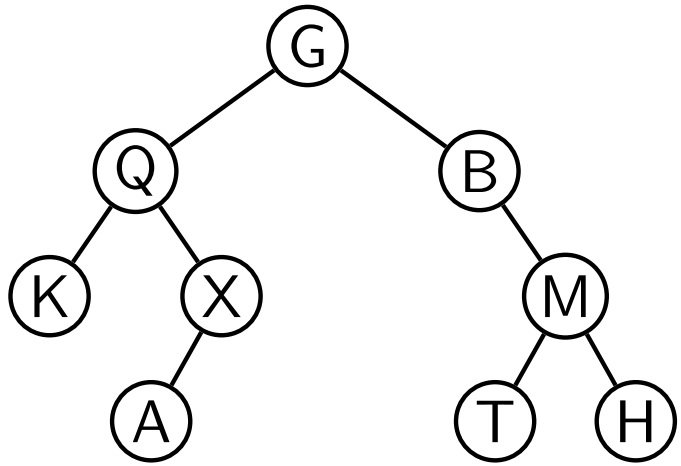
LIVE

LIVE CODING

- Write each tree from **Your Turn!** in **PreOrder with N**:



Pre: 4 9 7 3 8 2 5

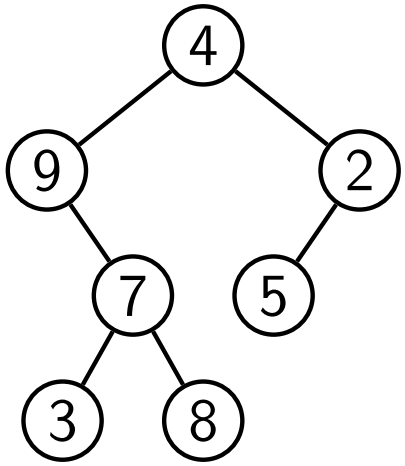


Pre: G Q K X A B M T H

(hint: a tree with n nodes has $n+1$ N's)

Binary Trees - Serialization (answers)

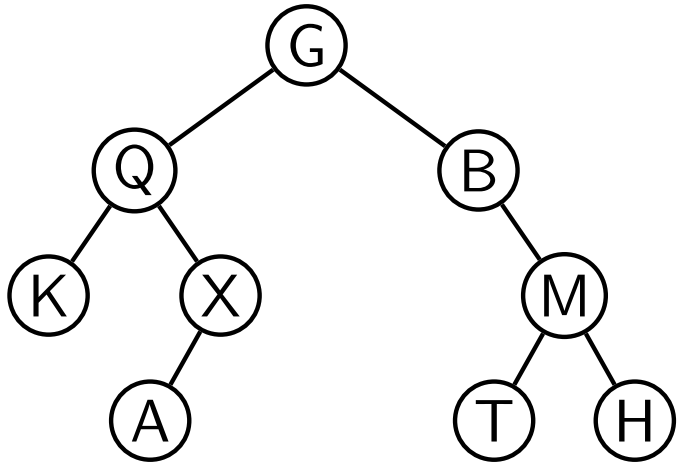
- Each tree in **PreOrder with N**:



Pre: 4 9 7 3 8 2 5

Serialized:

4 9 N 7 3 N N 8 N N 2 5 N N N



Pre: G Q K X A B M T H

Serialized:

G Q K N N X A N N N B N M T N N H N N

Binary Trees - Destructor

- C++ does **not** do automatic **garbage collection**, so we need to explicitly **delete** what we created with **new** to clean the memory
  (would not matter "a lot" on programs that terminate right after using one tree, but it is a good practice and will avoid memory leaks when you create many trees and/or nodes and then stop using them)

```
~BTree() {
  destroy(root);
}
void destroy(Node *n) {
  if (n == nullptr) return;
  destroy(n->left);
  destroy(n->right);
  delete n;
}
```

- `destroy` is a **PostOrder** traversal: a node is deleted only *after* its children (otherwise we would lose `n->left` and `n->right`).
- Note: if you want to check for memory leaks you could use valgrind (not available on Windows)

Binary Trees - Careful with Copies!

- Our class only stores a **pointer** to the root, so a default copy shares the same nodes:

```
void show(BTree<int> t) { ... }          // a copy:  shares the nodes!
```

- When the copy is destroyed, it deletes the nodes of the original too $\Rightarrow$ **double delete** (crash)
- Always pass trees by **reference** (**const** if the function does not change the tree, so only const methods can be called):

```
void show(const BTree<int> & t) { ... }  // no copy:  the original tree
```

- (A complete class would also define a copy constructor that copies all the nodes, e.g., with a preorder traversal... a good exercise!)

Binary Trees - Forbidding Copies (C++11)

- If we don't write them, C++ **automatically creates** a copy constructor and a copy assignment operator: both just copy the `root` pointer.
- Since C++11, `= delete` says that a function **must not exist**, so the **compiler** catches copies for us:

```
template <class T> class BTree {
public:
  BTree(const BTree &) = delete;                 // no copy constructor
  BTree & operator=(const BTree &) = delete;     // no copy assignment
```

- Any code that copies a tree is now a **compilation error** instead of a crash:

```
BTree<int> c = a;   // error:  use of deleted function (copy constructor)
b = a;              // error:  use of deleted function (copy assignment)
show(a);            // error, if show() receives the tree by value
v.push_back(a);     // error:  push_back would copy a
```

- *Rule of three*: if a class needs a **destructor**, a **copy constructor** or a **copy assignment operator**, it usually needs all three.

Binary Trees - Testing Everything We've Done

- Testing everything that has been implemented (saved as `test.cpp`):

```
#include "binaryTree.h"

int main() {

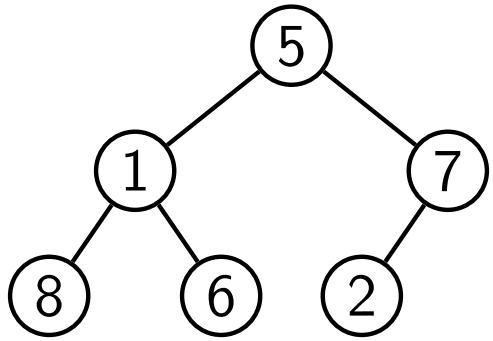
  BTree<int> t; // Create an empty tree of integers
  t.read("N");  // Read contents from stdin, using "N" as nullptr

  // Call some of the methods the class provides
  std::cout << "numberNodes = " << t.numberNodes() << std::endl;
  std::cout << "height = " << t.height() << std::endl;
  std::cout << "contains(2) = " << t.contains(2) << std::endl;
  std::cout << "contains(3) = " << t.contains(3) << std::endl;

  // Print nodes in several possible orders
  t.printPreOrder();
  t.printInOrder();
  t.printPostOrder();
  t.printBFS();
  t.printDFS();

  return 0;
}
```

Binary Trees - Testing Everything We've Done



- The tree in the figure, saved in a file called `input.txt`:

```
5 1 8 N N 6 N N 7 2 N N N
```

- `g++ -Wall test.cpp` and then `./a.out < input.txt` gives:

```
numberNodes = 6
height = 2
contains(2) = 1
contains(3) = 0
PreOrder: 5 1 8 6 7 2
InOrder: 8 1 6 5 2 7
PostOrder: 8 6 1 2 7 5
BFS: 5 1 7 8 6 2
DFS: 5 1 8 6 7 2
```

Binary Trees - Method Complexity

- What is the **time complexity** of the methods we've implemented?

numberNodes()	printPreOrder()	printBFS()
height()	printInOrder()	printDFS()
contains()	printPostOrder()	read(std::string null)

- All of these methods traverse each node in the tree exactly once (for *contains()*, this is in the worst case; for the other methods, it is always so), performing a constant number of operations per node.
- Therefore, all these methods have linear complexity, $\mathcal{O}(n)$, where n is the number of nodes in the tree.

  Memory: recursive methods use $\mathcal{O}(h)$ extra space for the call stack (h = height), which can reach $\mathcal{O}(n)$.

- Is it possible to improve this complexity for the `contains` method? **Binary Search Trees!** Search becomes $\mathcal{O}(h)$, where h is the height:
  - best case: $h \approx \log_2 n$ *(a balanced tree)*
  - worst case: $h = n - 1$ *(a tree that is just a chain)*

  $\Rightarrow$ keeping trees *balanced* will be the key (next class!)