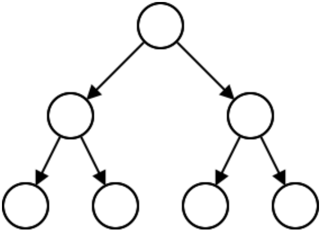
# Binary Trees

L.EIC

Algorithms and Data Structures

2025/2026



**P Ribeiro**, AP Tomas

# Non-linear Structures

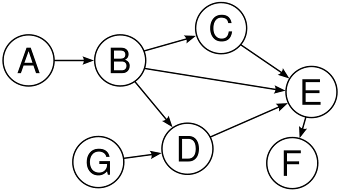
Arrays and lists are examples of **linear** data structures.

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).

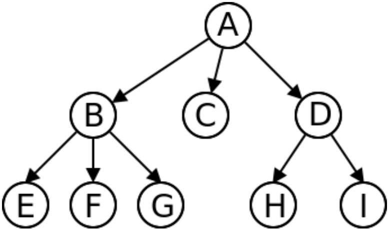
Are there other types of structures?

- 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.



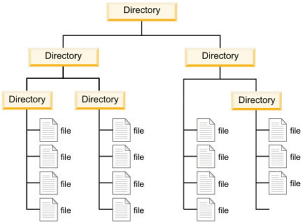
# Trees

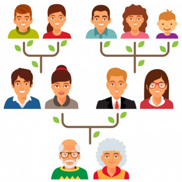
- A **tree** is a specific type of graph.
- Each element, called a **node**, has zero or more successors, but only one predecessor (except for the first node, known as the **root** of the tree).
- An example of a tree:

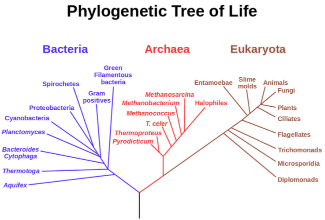


# Trees - Examples

- 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

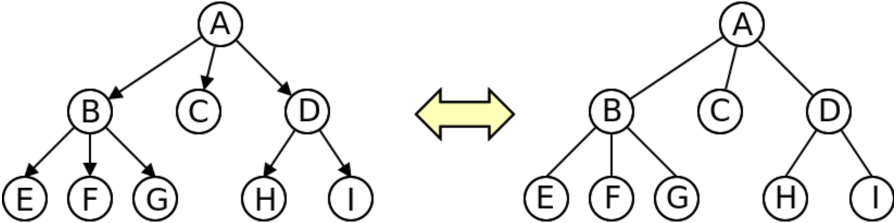


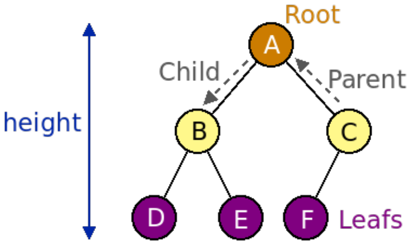


# Trees - Visualization

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

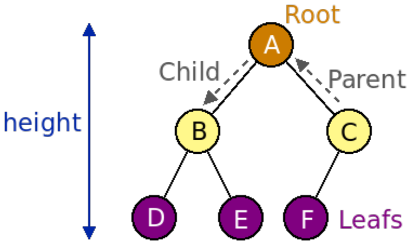


# Trees - Terminology




- 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.
- A **subtree** is a connected subset of nodes in the tree.
  - *Example: {B,D,E} is a subtree*

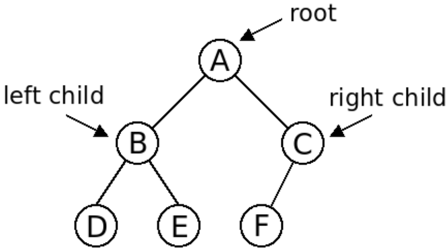
# Trees - Terminology




- 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*

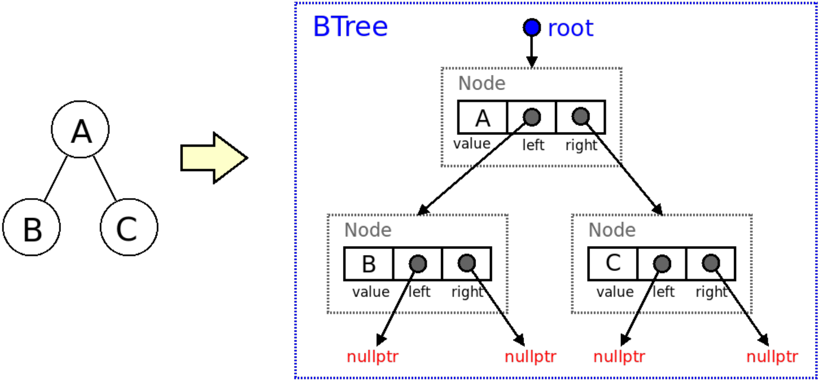
# Binary Trees

- The **arity** of a tree is the maximum degree of a node.
- A **binary tree** is a tree with arity 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
- Leafs should contain pointers to nullptr



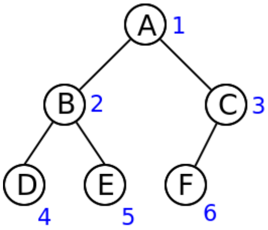
# Binary Trees - Implementation

- 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 **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 - 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

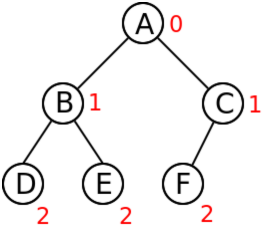
- We need to start counting... from the root!
- We want to have a `numberNodes()` method in the `BTree<T>` class.
  - *Example: if t is a tree, we want to be able to call t.numberNodes()*
- We'll use as **helper method** that is recursive.

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

// Helper method (recursive)
int numberNodes(Node *n) {
  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 - Tree Height

- We'll calculate the **height of a tree** (maximum depth of a node). For example, the tree in the figure has height 2 (with each node's depth in red).




- We'll create a **recursive method** very similar to the previous one:
  - **Recursive case:** the height of a tree is equal to 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 node: it has height zero (0).
  - If we stop at a *nullptr* tree, the height should be... **-1**
    - ★ *E.g.: height(1 node tree) = 1 + max(nullptr, nullptr) = 1 + max(-1, -1) = 0*

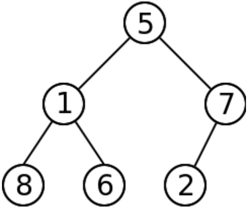
# Binary Trees - Tree Height

- Implementing it, with the base case of the recursive helper method being the *nullptr* tree (similar to the node-counting method):

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

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

# Binary Trees - Searching for an Element

- Now let's see a method to **check if an element is contained in a tree**. For example, the tree in the following figure contains the number 2, but does not contain the number 3:




- We'll create a **recursive method** very similar to the previous ones:
  - **Base case 1:** if the current node is *nullptr*, it doesn't contain the value we're looking for, so we return *false*.
  - **Base case 2:** if the value we're looking for is in the current node, we return *true*.
  - **Recursive case:** if it's not in the root, then we check if it's in the left subtree **OR** the in the right subtree.

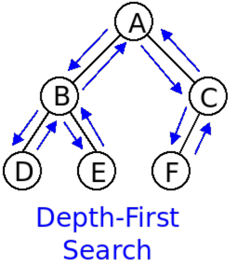
# Binary Trees - Searching for an Element

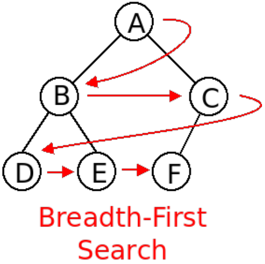
- Implementing it:

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

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

# Binary Trees - Writing the Nodes of a Tree

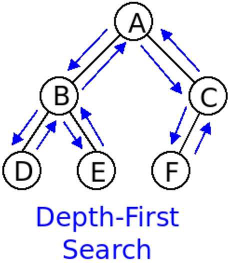
- How can we **write the content (nodes)** of a tree?
- We need to **traverse all nodes**. But in what **order**?
- Let's distinguish between two different orders:




- **Depth-First Search** (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




- 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

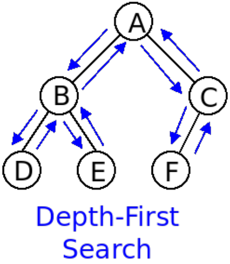
- Translating what was described on the previous slide into code:

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

// Helper method (recursive)
void printPreOrder(Node *n) {
  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.




- For the tree in the figure:
  - **PreOrder:** A B D E C F
  - **InOrder:** D B E A F C
  - **PostOrder:** D E B F C A

- Implementing *InOrder*:

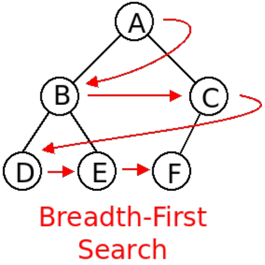
```cpp
// Write all nodes in InOrder
void printInOrder() {
  std::cout << "InOrder:";
  printInOrder(root);
  std::cout << std::endl;
}

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

# Binary Trees - Depth-First Search

- Implementing *PostOrder*:

```
// Write all nodes in PostOrder
void printPostOrder() {
  std::cout << "PostOrder:";
  printPostOrder(root);
  std::cout << std::endl;
}

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

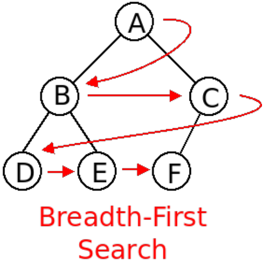
# Binary Trees - Breadth-First Search




- 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

- Let's see an example:




1. Initially, we have $Q = \{A\}$
2. We dequeue and print **A**, add children $B$ and $C$: $Q = \{B, C\}$
3. We dequeue and print **B**, add children $D$ and $E$: $Q = \{C, D, E\}$
4. We dequeue and print **C**, add child $F$: $Q = \{D, E, F\}$
5. We dequeue and print **D**, no children: $Q = \{E, F\}$
6. We dequeue and print **E**, no children: $Q = \{F\}$
7. We dequeue and print **F**, no children: $Q = \{\}$

# Binary Trees - Breadth-First Search in Code

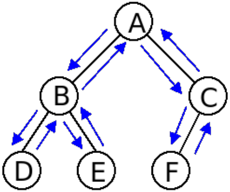
- Implementing in code:

```
// Write all nodes in BFS order
void printBFS() {
  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

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




Depth-First Search

1. Initially, we have $S = \{A\}$
2. *Pop* and print **A**, push children $B$ and $C$: $S = \{B, C\}$
3. *Pop* and print **C**, push child $F$: $S = \{B, F\}$
4. *Pop* and print **F**, no children: $S = \{B\}$
5. *Pop* and print **B**, push children $D$ and $E$: $S = \{D, E\}$
6. *Pop* and print **E**, no children: $S = \{D\}$
7. *Pop* and print **D**, no children: $S = \{\}$

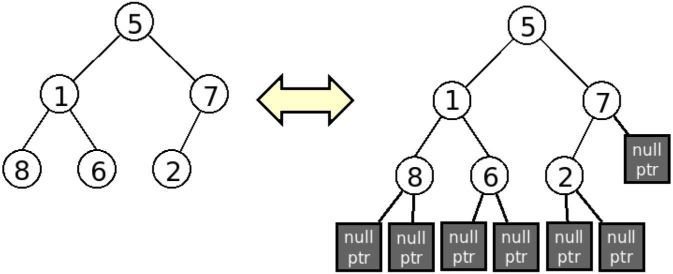
# Binary Trees - Depth-First Search in Code

- Implementing in code:

```
// Write all nodes in DFS order
void printDFS() {
  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->left);
      s.push(cur->right);
    }
  }

  std::cout << std::endl;
}
```

# 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:

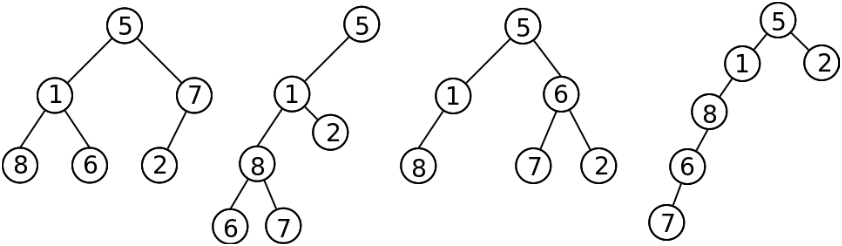



- If we represent *nullptr* as **N**, then the tree in *PreOrder* would be represented as:

  `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 inorder 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 - Reading a Tree in PreOrder

- Implementing a generic preorder read of a tree:

  (with the string *null* representing a nullptr - for the previous usage example we should call `read("N")`)

```
// Read a tree in preorder from stdin
void read(std::string null) {
  root = readNode(null);
}

// Helper method (recursive)
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;
}
```

# Binary Trees - Destructor

- C++ does **not** do automatic **garbage collection**, so we need to explicitely **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;
}
```

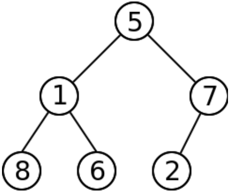
- Note: if you want to check for memory leaks you could use valgrind (not available on Windows)

# Binary Trees - Testing Everything We've Done

- Testing everything that has been implemented:

```
#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 << "depth = " << t.depth() << 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



- Running the program (assuming executable name `a.out`) with input from 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
```

- `./a.out < input.txt` would give the following result:

```
numberNodes = 6
depth = 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 7 2 1 6 8
```

# Binary Trees - Method Complexity

- What is the **time complexity** of the methods we've implemented?
  - numberNodes()
  - depth()
  - contains()
  - printPreOrder()
  - printInOrder()
  - printPostOrder()
  - printBFS()
  - printDFS()
  - readTree(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.
- Is it possible to improve this complexity for the `contains` method? **Binary Search Trees!**