# CSC 212: Data Structures and Abstractions

## Binary search trees


Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island


Spring 2025

THINK BIG WE DO℠

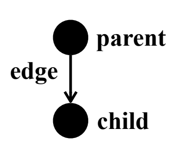
---

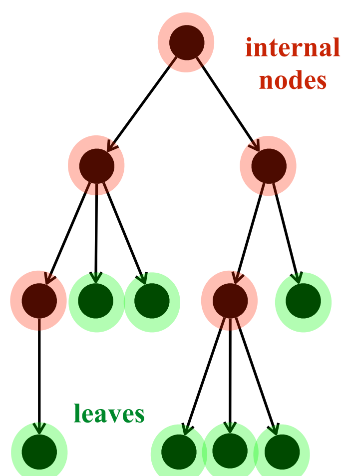
# Trees

---

# Trees

- Definition
  - data structure that consists of **nodes** connected by **edges**
    - hierarchical structure, with a single **root** node
    - each node can have zero or more **children**
- Terminology
  - each node is either a **leaf** or an **internal node**
    - leaves are nodes with no children, while internal nodes are nodes with one or more children
  - nodes with the same **parent** are **siblings**

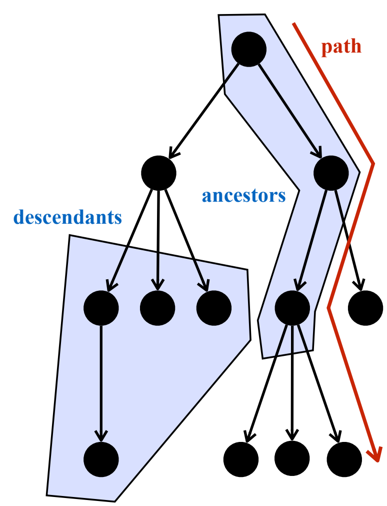





---

# Paths

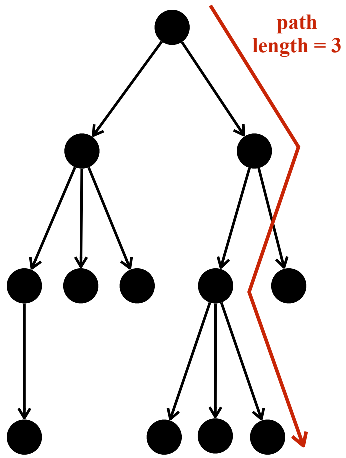



A **path** from node $v_0$ to $v_n$ is a sequence of nodes $v_0, v_1, \ldots, v_n$, where there is a (directed) edge from one node to the next

The **descendants** of a node $v$ are all nodes reached by a path from node $v$ to the leaf nodes

The **ancestors** of a node $v$ are all nodes found on the path from the root to node $v$

# Depth and height




The length of a **path** is the number of edges in the path

The **depth** (level) of a node *v* is the length of the path from the root node to *v*

The **height** of a node *v* is the length of the path from *v* to its deepest descendant

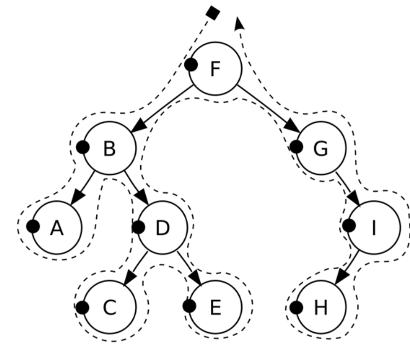
The **depth of the tree** is the depth of deepest node

The **height of the tree** is the height of the root



# Traversals

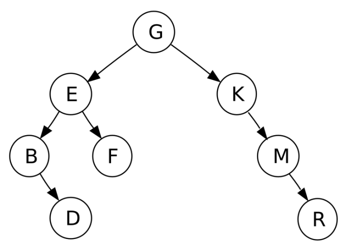
- Definition
  - a **<u>traversal</u>** is a way of visiting all the nodes in a tree
- Types of traversals:
  - **pre-order traversal:** visit the root node first, then recursively visit all subtrees
  - **post-order traversal:** recursively visit all subtrees first, then visit the root node






# Pre-order traversal

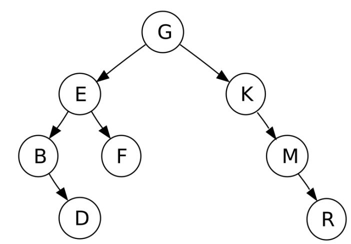
```
algorithm preorder(p) {
    visit(p)
    for each child c of p {
        preorder(c)
    }
}
```






# Post-order traversal

```
algorithm postorder(p) {
    for each child c of p {
        postorder(c)
    }
    visit(p)
}
```



# Binary trees

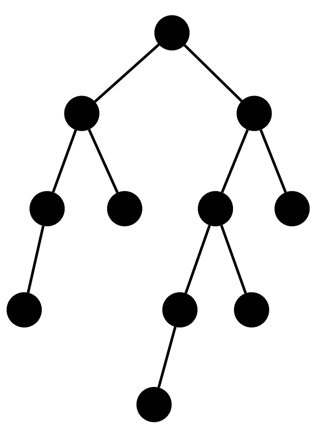
# k-ary trees

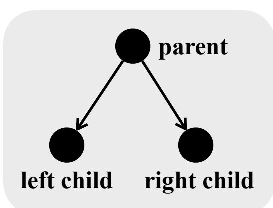
- k-ary tree
  - ✓ every node has <u>between 0 and k</u> children
- Full k-ary tree
  - ✓ every node has <u>exactly 0 or k</u> children
- Complete k-ary tree
  - ✓ every level is <u>entirely filled</u>
  - ✓ except possibly the deepest, where all nodes are as far left as possible
- Perfect k-ary tree
  - ✓ every leaf has the same depth and the tree is full



# Binary trees

- Definition
  - ✓ a special case of a k-ary tree, where $k = 2$

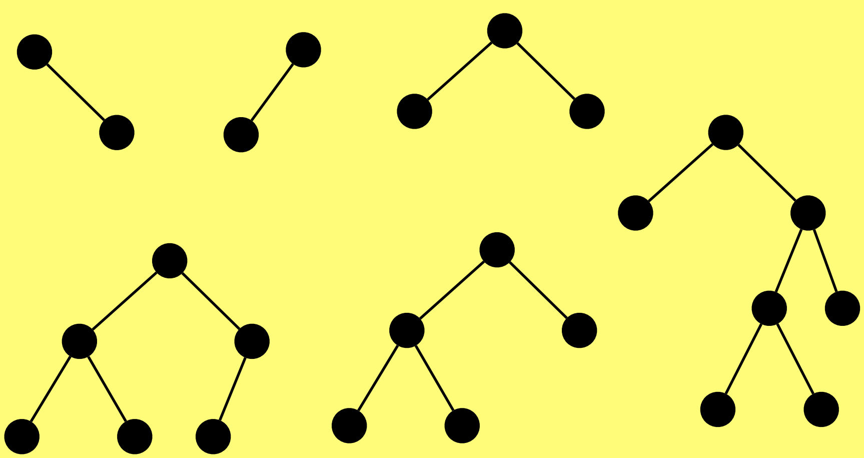







# Practice

- Mark the following binary trees (k=2) as full/complete/perfect



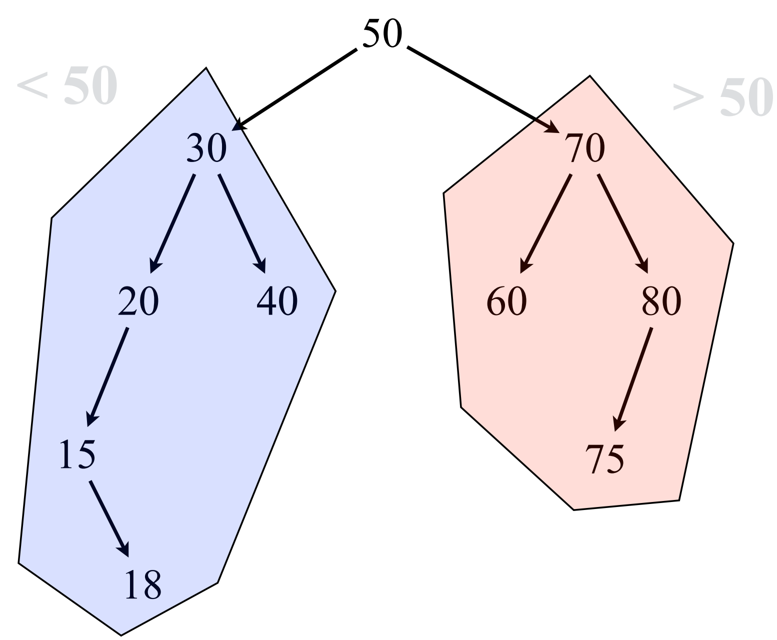
# Binary search trees

## Binary search tree

- A binary search tree (BST) is **a binary tree**
- A BST has **symmetric order**
  - ✓ each node $x$ in a BST has a key denoted by $key(x)$
  - ✓ for all nodes $y$ in the left subtree of $x$, $key(y) < key(x)$ **
  - ✓ for all nodes $y$ in the right subtree of $x$, $key(y) > key(x)$ **

`(**) assume that the keys of a BST are pairwise distinct`



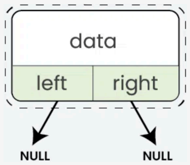





## Representing a node

```
template <typename T>
struct BSTNode {
    T data;
    BSTNode<T> *left, *right;

    BSTNode(const T& value) {
        data = value;
        left = right = nullptr;
    }
};
```




The implementation of a **binary tree node** requires a structure that can accommodate connections to two child nodes

https://www.geeksforgeeks.org/binary-tree-representation/

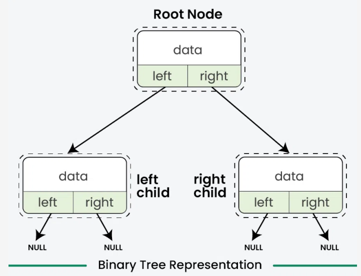
# Representing a binary search tree

```
template <typename T>
class BST {
    private:
        struct Node {
            T data;
            Node *left, *right;
            Node(const T& value) {
                data = value;
                left = right = nullptr;
            }
        };

        Node *root;
        size_t size;

    public:
        BST() : root(nullptr), size(0) {}
        ~BST() { clear(); }
        size_t getSize() const { return size; }
        bool empty() const { return size == 0; }

        void insert(const T& value);
        void remove(const T& value);
        bool contains(const T& value) const;
        void clear();
};
```




https://www.geeksforgeeks.org/binary-tree-representation/



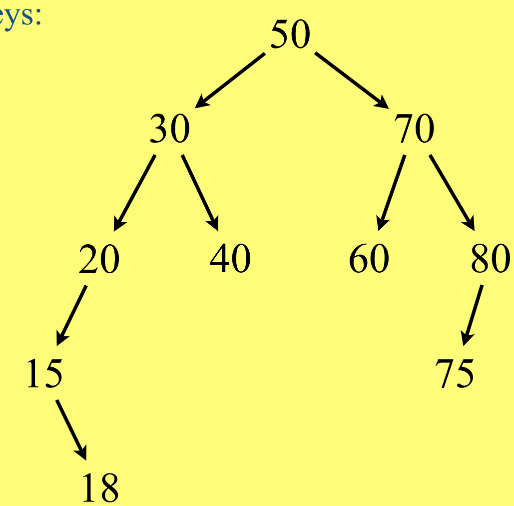
# Insert

- Algorithm
  - ✓ if tree is empty, create a new node as root, done
  - ✓ starting node p at the root, repeat:
    - compare the key to insert with the key of p, if equal, done
    - if the new key is less than the key of p, set p to the left subtree
      - if p is empty, create new node here, done
      - else continue
    - if the new key is greater than the key of p, set p to the right subtree
      - if p is empty, create new node here, done
      - else continue
- Time complexity
  - ✓ $O(h)$, where $h$ is the height of the tree



# Practice

- Insert the following keys:
  - ✓ 65, 27, 90, 11, 51






# Remove

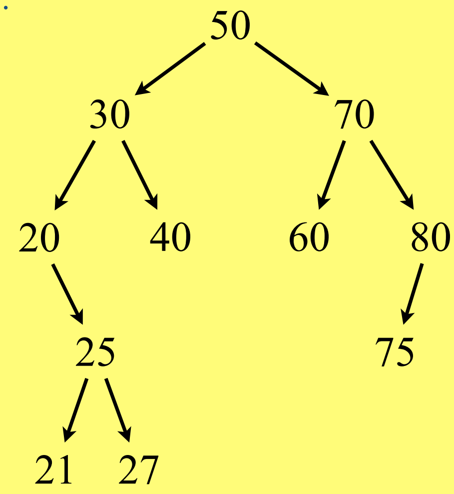
- Approach
  - ✓ find node to be removed, then apply one of the cases below
  - ✓ **case 1**: node is a leaf
    - trivial, delete node and set parent's pointer to nullptr
  - ✓ **case 2**: node has 1 child
    - set parent's pointer to the only child and delete node
  - ✓ **case 3**: node has 2 children
    - find successor (smallest node in the right subtree)
    - copy successor's data to node
    - delete successor

`can also use predecessor`

- Time complexity
  - ✓ $O(h)$, where $h$ is the height of the tree

# Practice

- Remove the following keys:
  - ✓ 27, 40, 80, 20, 30, 50





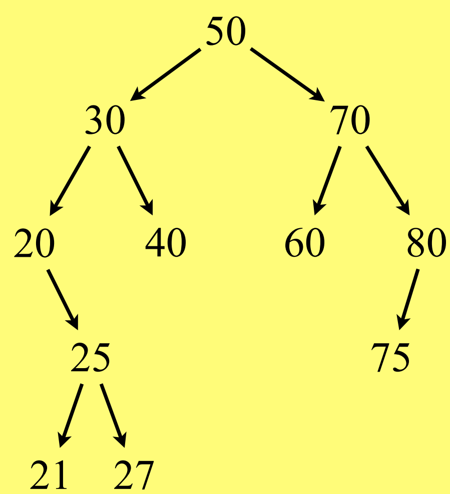

# Contains

- Algorithm
  - ✓ start at root node
  - ✓ if the search key matches the current node's key then found
  - ✓ if search key is greater than current node's key
    - search on right child
  - ✓ if search key is less than current node's
    - search on left child
  - ✓ stop when current node is nullptr (not found)
- Time complexity
  - ✓ $O(h)$, where $h$ is the height of the tree



# Practice

- Search the following keys:
  - ✓ 25, 77, 18, 40, 75






# Analysis

# Practice

- Starting from an empty BST, insert the following keys in the order given
  - ✓ 20 10 30 5 15 25 35
  - ✓ 10 20 5 15 30 35 25
  - ✓ 5 10 15 20 25 30 35
  - ✓ How is the order of insertion related to the shape of the tree?
  - ✓ How is the height of the tree related to the number of nodes?



# Practice

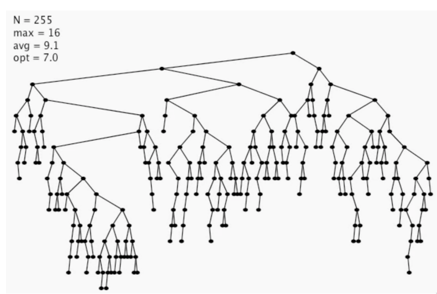
- Complete the following table with rates of growth
  - ✓ as a function of the number of nodes

| Operation | Best case | Average case | Worst case |
|---|---|---|---|
| Insert | | | |
| Remove | | | |
| Search | | | |



# Average case

- Proposition
  - ✓ if $n$ distinct keys are randomly inserted into a BST, the expected number of compares is $\sim c \log n$
    - can be formally justified through probabilistic analysis (not covered in this class)




- Implications
  - ✓ even without explicit balancing mechanisms, randomly built BSTs provide reasonably efficient operations