CSC 212: Data Structures and Abstractions

Binary search trees (part 1)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

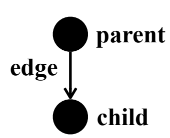
Spring 2025

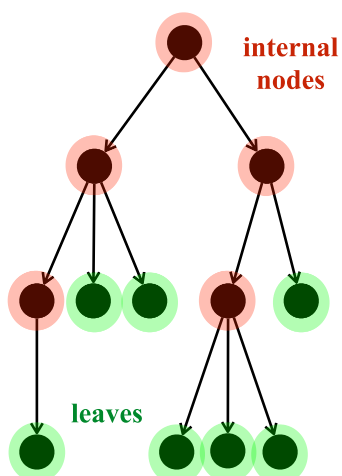
THINK BIG WE DO℠

Trees

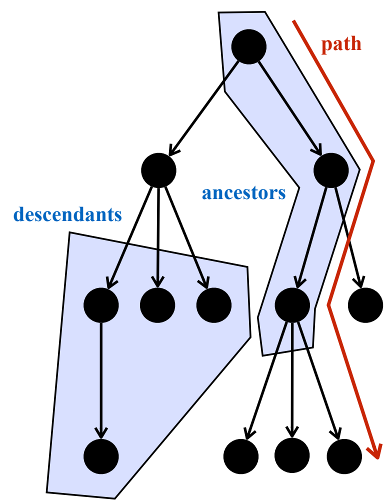
Trees

- Definition
  - data structure that consists of **nodes** connected by **edges**
    - <u>hierarchical structure</u>, 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

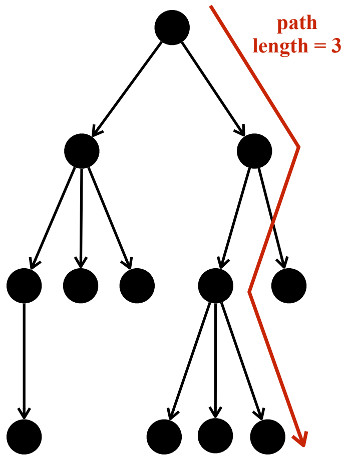


A **path** from node v_0 to v_n is a sequence of nodes $v_0, v_1, \ldots, v_n$, where there is a (directed) edge from one node to the next

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*

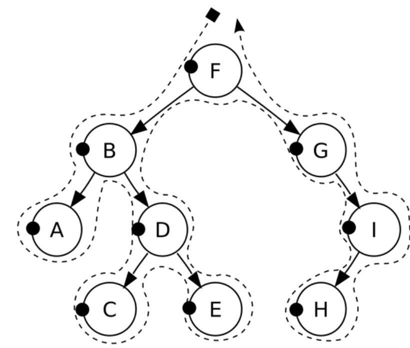
The **height** of a node *v* is the length of the path from *v* to its deepest descendant

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

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

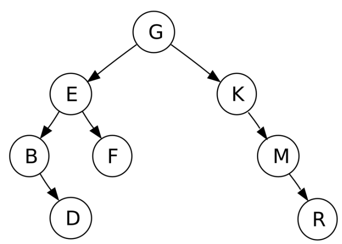
Traversals

- Definition
  - a **traversal** 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



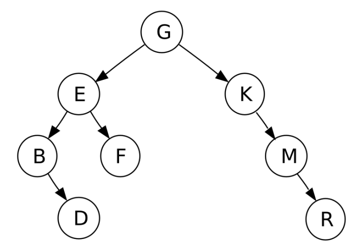
Pre-order traversal

```
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)
}
```



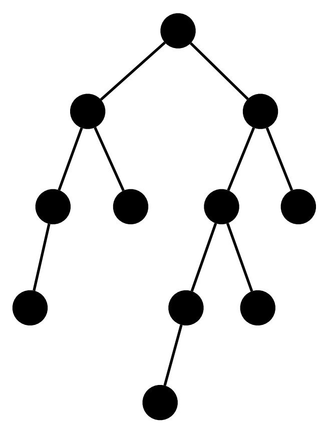
Binary trees

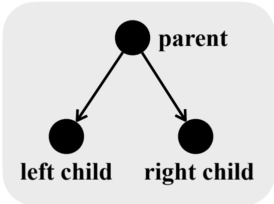
k-ary trees

- k-ary tree
  - every node has between 0 and k children
- Full k-ary tree
  - every node has exactly 0 or k children
- Complete k-ary tree
  - every level is entirely filled
  - 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

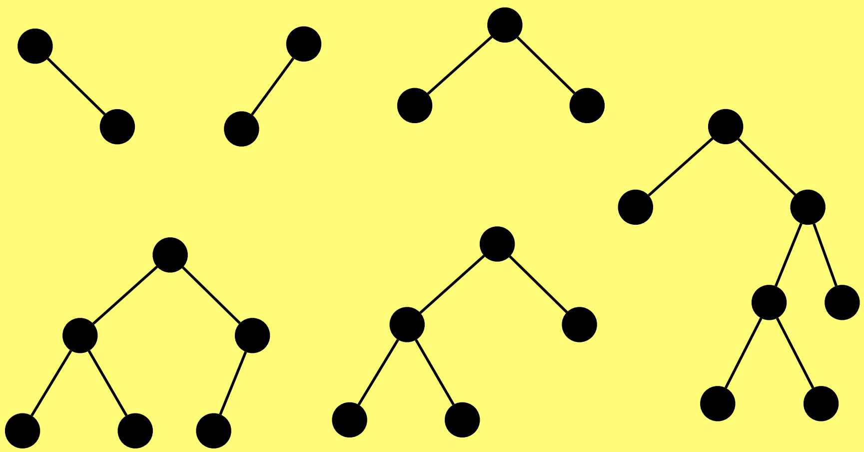
- Definition
  - a special case of a k-ary tree, where $k = 2$





Practice

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

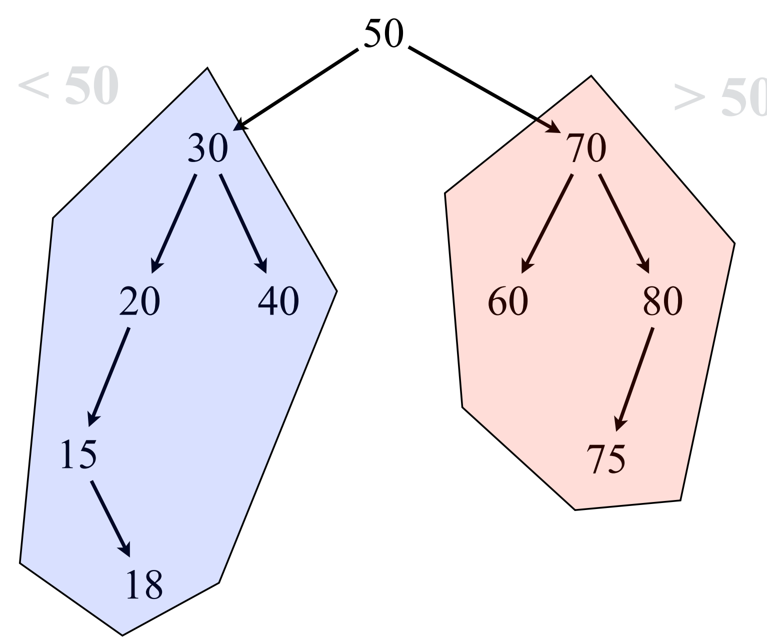


Binary search trees

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, $key(y) > key(x)$ **

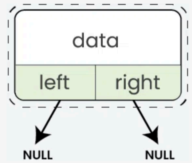
`(**) assume that the keys of a BST are pairwise distinct`



Representing a node

```
template <typename T>
struct BSTNode {
    T key;
    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

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();
};
```