CSC 212: Data Structures and Abstractions

Binary search trees (part 1)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

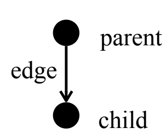
Fall 2025

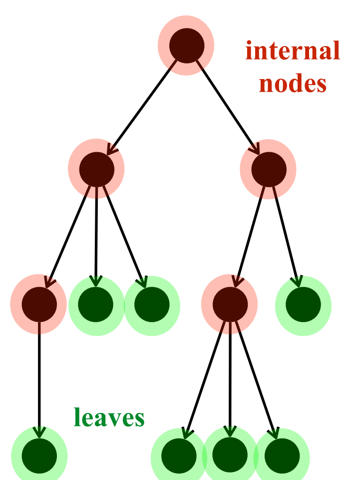
THINK BIG WE DO℠

Trees

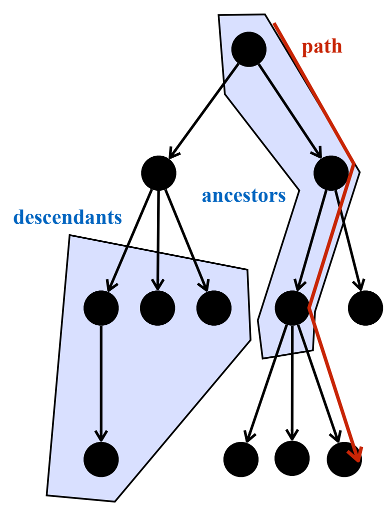
Trees

- Definition
  - data structure consisting of **nodes** connected by **edges** forming a hierarchical structure with the following properties:
    - there exists a unique node called the **root** with no parent
    - every node except the root has exactly one parent
    - there is exactly one path from the root to any node
- Terminology
  - root, parent/child, leaf/external node, internal node, siblings





Paths

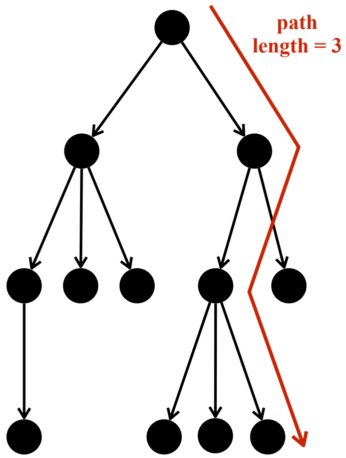


A **path** from node v_0 to v_n is a sequence of nodes $v_0, v_1, \ldots, v_n$, where there is a (directed) edge from one node to the next

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

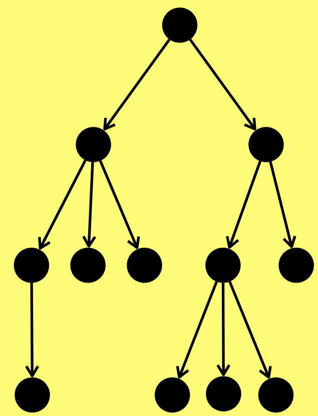
The **height** of a node v is the length of the path from v to its deepest descendant

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

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

Practice

- Label all nodes with height and depth
  - ✓ indicate the height and the depth of the tree



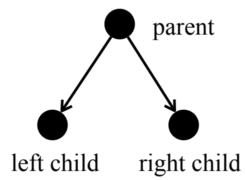
Binary trees

k-ary trees

- **k-ary tree**
  - ✓ a tree in which every internal node has at most k children
- **Full k-ary tree**
  - ✓ a tree in which every internal node has exactly k children
- **Complete k-ary tree**
  - ✓ given h representing the height of the tree, all levels 0 through $h-1$ are completely filled, and level h has all nodes as far left as possible
- **Perfect k-ary tree**
  - ✓ all internal nodes have exactly k children and all leaves are at the same depth

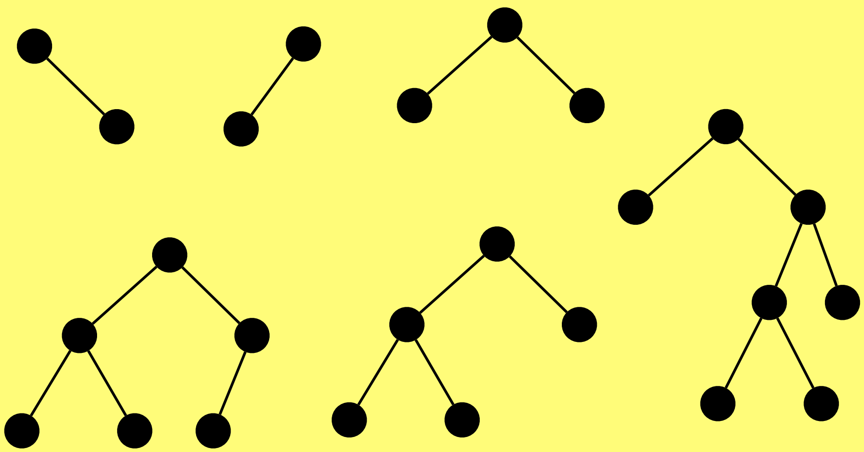
Binary trees

- Definition
  - a special case of a k-ary tree, where $k = 2$
- Properties
  - every node has at most 2 children
  - children are distinguished as left child and right child
  - the subtrees rooted at these children are called the left subtree and right subtree



Practice

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

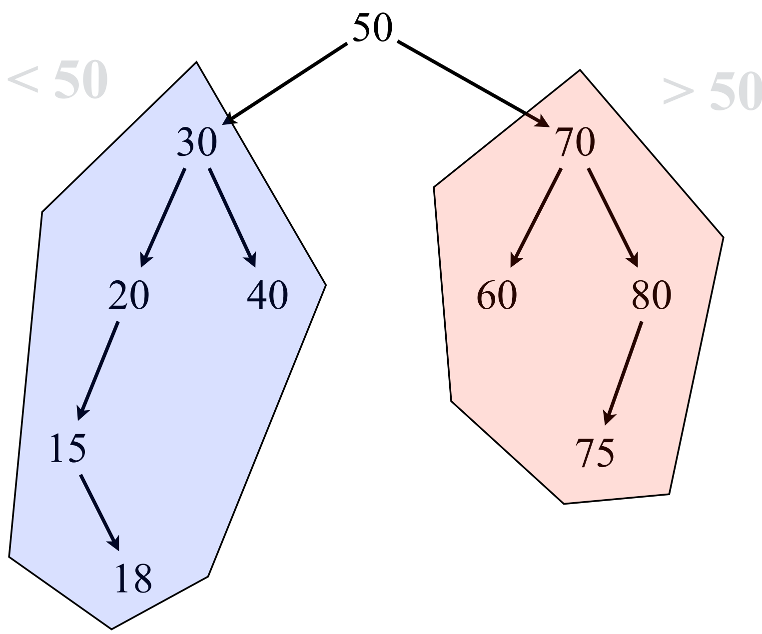


Binary search trees

Binary search tree

- A binary search tree (BST) is a binary tree that satisfies the **BST property** (also called symmetric order)
- **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, $key(y) > key(x)$ **
  - this property must hold for every node in the tree

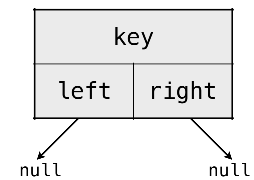
`(**) assume that the keys of a BST are pairwise distinct`



Representing a node

```
template <typename T>
class Node {
    private:
        T key;
        Node<T> *left, *right;
        friend class BST<T>;

    public:
        Node(const T& value) {
            key = value;
            left = right = nullptr;
        }
};
```



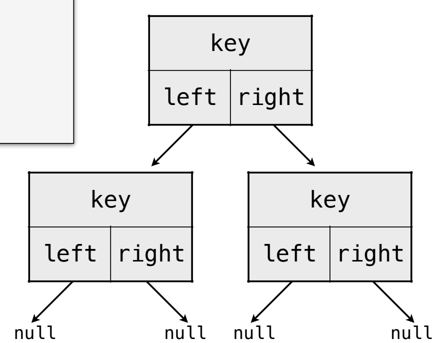
The implementation of a **BST node** requires a structure that can accommodate connections to two child nodes

Representing a binary search tree

```
template <typename T>
class BST {
    private:
        Node<T> *root;
        size_t size;
        bool search_helper(Node<T>* p,const T& target);
        Node<T>* insert_helper(Node<T>* p, const T& value);
        Node<T>* remove_helper(Node<T>* p, const T& value);

    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 search(const T& value) const;
        void clear();
};
```



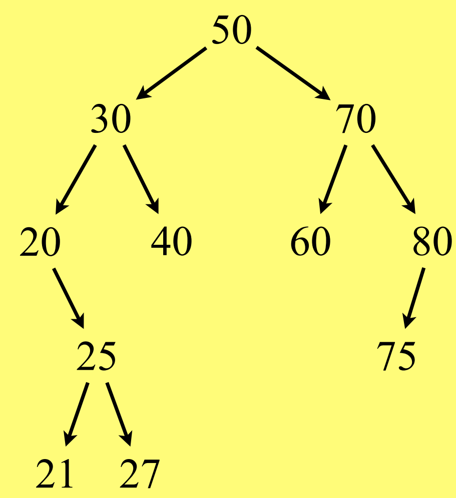
Operations: search and insert

Search

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

Practice

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



Recursive search

```
template <typename T>
bool BST<T>::search_helper(Node<T>* p, const T& target) {
    if ( !p ) return false;

    if (target < p->key) return search(p->left, target);
    else if (target > p->key) return search(p->right, target);
    else return true;
}
```

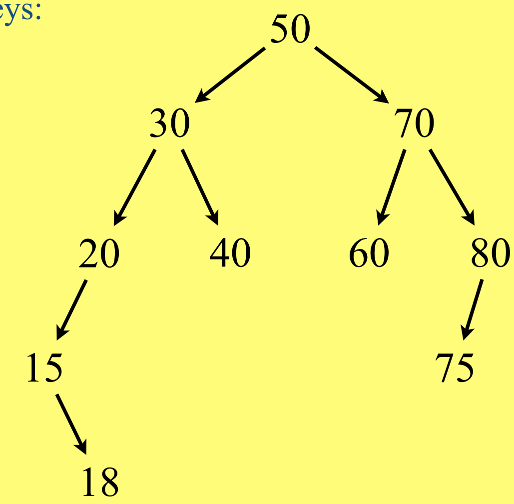
```
template <typename T>
bool BST<T>::search(const T& value) const {
    return search_helper(root, value);
}
```


Insert

- Algorithm
  - ✓ if tree is empty, create a new node as the root and done
  - ✓ otherwise, start at the root node and repeat:
    - compare the key to insert with the current node's key
      - if equal, the key already exists — done
      - if the new key is less than the current node's key
        - if left child is empty, create new node as left child — done
        - otherwise, move to the left child and continue
      - if the new key is greater than the current node's key
        - if right child is empty, create new node as right child — done
        - otherwise, move to the right child and continue
- Time complexity
  - ✓ $O(h)$, where h is the height of the tree

Practice

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



Recursive insert

```
template <typename T>
Node<T>* BST<T>::insert_helper(Node<T>* p, const T& value) {
    if ( !p ) return new Node<T>(value);
    if (value < p->key) p->left = insert_helper(p->left, value);
    else if (value > p->key) p->right = insert_helper(p->right, value);
    return p;
}
```

```
template <typename T>
void BST<T>::insert(const T& value) {
    root = insert_helper(root, value);
    ++size;
}
```


Repeated keys

- Assumption
  - ✓ the BST contains unique keys, no duplicate keys are allowed in the standard implementation

- Handling duplicate keys
  - ✓ **strategy 1**: ignore duplicates
    - if key is in the tree, do nothing and return
  - ✓ **strategy 2**: frequency counter
    - add a counter/frequency field to each node
    - increment the counter when a duplicate key is inserted
  - ✓ **strategy 3**: update associated value
    - if the BST is used as a map/dictionary (key-value pairs)
    - replace the old value with the new value for the duplicate key