# Trees: Definitions

- A **tree** is a dynamic data structure in which data are stored in **nodes**.
- Each node has a number of pointers to other nodes.
- If a node A points to a node B, then B is a **child** of A and A is a **parent** of B.
- One of the nodes in the tree is the **root**: no node in the tree points to it.
- A node with no children (i.e. null pointer) is called a **leaf node**.
- The number of children of each node can be fixed or variable.
- A tree is usually drawn "upside down" with the root at the top and the leaves at the bottom.

# Trees: Properties

- There is a unique path from the root to every node in the tree.
- There are pointers from parent to child, but not in the reverse direction.
- The children of the root node can be thought of as the roots of smaller **subtrees**. That is, the data structure is recursive.
- A tree in which every node has one child is the same as a singly linked list.

# Binary Trees

- A **binary tree** is a tree in which every node has at most two children: **left** and **right**.
- If there is no left or right child, the corresponding pointer is null.
- A node is defined as

```
class Node {
public:
  int data;
  Node *left, *right;
  Node(int d, Node *l, Node *r)
    : data{d}, left{l}, right{r} {}
};
```

- A pointer to the root node is used to access the tree.

# Inserting Nodes

- To add the root node: `root = new Node(data, nullptr, nullptr);`
- To add a left child to a node pointed to by `p` (assuming that there was no left child before):

```
p->left = new Node(data, nullptr, nullptr);
```

- Inserting a right child is similar.

# Removing Nodes

- Removing a leaf node is easy, as long as we have a pointer `p` to its parent.
- For example, to remove the left child (a leaf) of `p`:

```
delete p->left;
p->left = nullptr;
```

- If we do not have a pointer to the parent, it is hard (how do we find the parent?).
- If we delete a non-leaf node, how do we link the subtrees?

# Traversing Trees

- We can do this recursively:
  - If the pointer is null, do nothing (empty tree); otherwise
  - recursively traverse left subtree
  - examine item in node
  - recursively traverse right subtree
- This is called **inorder** traversal: the elements are traversed from left to right.
- **Preorder** traversal: examine the node first, and then visit the children.
- **Postorder** traversal: visit the children first, then examine the node.

## Example: Printing Elements in Order

```
void print(Node *root)
{
  if (root) {  // only do something if nonempty
    print(root->left);
    cout << root->data << endl;
    print(root->right);
  }
}
```

# Example: Height of a Tree

```
int height(Node *root)
{
  if (!root)
    return 0;   // empty tree
  else
    return 1 + max(height(root->left), height(root->right));
}
```

# Deleting All Nodes

It is important to delete the subtrees before deleting the root (postorder).

```
void deleteTree(Node *&root)
{
  if (root) {
    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
    root = nullptr;
  }
}
```

## Binary Search Trees

- A **binary search tree** is a binary tree in which the data in each node is greater than or equal to **every** node in the left subtree and less than or equal to every node in the right subtree.
- To look for an item, look at the data at the root. If it is not there, repeat the search with either the left or the right subtree.
- To insert an item, follow a path to a leaf node and insert as either a left or a right child.

# Searching in a Binary Search Tree

```
Node *find(Node *root, int data)
{
  if (!root) return nullptr;       // not found
  if (root->data == data)
    return root;
  else if (root->data > data)
    return find(root->left, data);
  else
    return find(root->right, data);
}
```

# Inserting a Node

```
void insert(Node *&root, int data)
{
  if (!root) {
    root = new Node(data, nullptr, nullptr);
  } else if (root->data >= data) {
    insert(root->left, data);
  } else {
    insert(root->right, data);
  }
}
```

# Deleting a Node (Sketch)

- We wish to delete a node pointed to by `p`.
- Deleting a leaf node is the same as before.
- Otherwise, look at the leftmost leaf of the right subtree, call it `N`. i.e. go to `p->right` and follow the left children for as long as possible.
- `N` is the element that comes after `p`.
- So we copy the value in `L` to `p`, and recursively delete the node `N` until it is a leaf (which is easy to delete).

# Efficiency

- The amount of work to find, insert, or delete a node in the tree is proportional to the height of the tree.
- For a "bushy" tree, we have:
  - nodes = 1: height = 1
  - nodes = 3: height = 2
  - nodes = 7: height = 3
  - nodes = 15: height = 4
  - ...
  - nodes = 1048575: height = 20
- If there are $n$ elements in the tree, each operation takes approximately $\log_2 n$ steps.
- Doubling the size of the tree requires just a little bit more work.

# Efficiency

- But if the tree is not "bushy", then the height can be very bad.
- For example, if we insert the elements from smallest to largest, the tree becomes a linked list.
- In that case, the height is $n$.
- A number of variations on binary search trees allow "rebalancing" whenever the heights of the two subtrees are very different. This ensures that the operations are fast.
- The STL containers `map` and `set` are implemented with a balanced binary search tree.
- In a map, each data element is a key-value pair and the comparison operator is defined to compare only the key.

# Other Uses of Trees

Trees are used in many applications in computer science.

- Expression trees represent arithmetic expressions for evaluation: nodes contain operators (binary) and children contain the operands. Use postorder traversal to evaluate.
- Parse tree: represent the source code of a program by its logical units. May have more than two children per node.
- Image compression with quadtrees.
- and a lot more.