**Why Trees?**

- Sorted list arrays → fast searching, but:
  - Slow inserts
  - Slow deletes

- Binary search trees → fast for search and update
  - Variants of BSTs are very important in database applications
  - BSTs often used within other algorithms

---

**Binary Trees**

**Terminology**

- node
- edge
- root node
- internal nodes
- external nodes / leaves

---

**Binary Trees**

**Recursive Definition**

- A binary tree is (empty) or a root node with a binary tree as left subtree and a binary tree as right subtree

---

**Binary Trees**

**Terminology**

- $\mu$ is the parent of $v$
- $v$ is the left child of $\mu$
- $\mu$ is an ancestor of $u$ and $v$
- $\mu$ and $v$ are descendants of $p$

---

**Binary Trees**

**Items are stored in the internal nodes**

The tree is sorted if, for every node, items in its left subtree are smaller than the item in the node, and items in the right subtree are larger.
Binary Trees

More terminology:
- The size of the tree is the number of nodes.
- A tree is full if every internal node has two children.
- A path is a sequence of edges between two nodes.
- The length of the path is the number of edges in the path.
- For every node μ, the length of the unique path between μ and the root is the depth of μ.
- The height of the tree is the maximum depth of any node.
- The path length is the sum of depths over all nodes.

Binary Search Trees

A binary search tree (BST) is a sorted binary tree.

We assume each node is a structure holding
- Data
- A pointer to the (root of the) left subtree
- A pointer to the (root of the) right subtree

In a Java-like language, this would look like

```java
class Node {
    int data;
    Node left;
    Node right;
    Node() {
    }
}
```

Searching

```java
Node search(Node tree, Item x) {
    if (tree == null) return null;
    else if (x < tree.info) return search(tree.left, x);
    else if (x > tree.info) return search(tree.right, x);
    else return tree;
}
```

Example: search(tree, 'C')

```

Inserting

```java
Node insert(Node tree, Item x) {
    if (tree == null) return new Node(x);
    if (x < tree.info) tree.left = insert(tree.left, x);
    else tree.right = insert(tree.right, x);
    return tree;
}
```

Example: insert(tree, 'D')

```

Deleting

- Follow path to node v containing item
- Multiple cases:
  - v has no internal node children: remove v
  - v has one internal child: make that child the parent of v and remove v
  - v has two internal children: find the rightmost internal node in the left subtree of v, remove it, and substitute it for v (see illustration)

Example: delete(tree, 'X')

Final Words

- Basic operations (search, insert, delete) have same complexity = O(height of tree)
- If tree is balanced, height = O(log n)
- Worst case, tree is a linked list, height = O(n)
- Trees can become unbalanced through sequences of inserts and deletes
- Next time: ways to maintain balance (efficiently)