A tree is a “natural” way to represent hierarchical structure and organization.

A lot of problems in computer systems can be solved by breaking it down into smaller pieces and arranging the pieces in some form of hierarchical structure.

For example: binary search
**Node Path**

A path: the set of nodes visited to get from a node higher up on the tree to a node lower down, not including the originating, higher up node.

There is a unique path from one node to another, e.g.,

- path from root to \( y \) is \( \{ l, b, y \} \)
- the path length of root to \( y \) is 3 (hops)

Path length may be 0, e.g., \( l \) going to itself is a path.

**Ancestors and Descendants**

Ancestor: \( l \) and \( b \) are ancestors of \( y \): there is a path from \( l \) to \( y \) and \( b \) to \( y \)

- each node is its own ancestor
- node \( i \) is a proper ancestor of node \( j \) if the path length from \( i \) to \( j \) is not 0

Descendant: \( b \) and \( y \) are descendants of \( l \): there is a path from \( l \) to \( b \) and \( l \) to \( y \)

- node \( j \) is a proper descendant of node \( i \) if the path length from \( i \) to \( j \) is not 0

**Depth and Height**

The depth of node \( i \) is the length of the path from the root node to \( i \)

- \( \text{depth}(R) = 0 \), \( \text{depth}(x) = 3 \)

All nodes on a level of the tree have the same depth

- the root is at level 0

The depth of a tree is the maximum depths of all nodes, \( T \) is of depth 3

The height of node \( i \) is the longest path from \( i \) to a leaf node

- \( \text{height}(x) = \text{height}(d) = 0 \),
- \( \text{height}(b) = \text{height}(r) = 1 \),
- \( \text{height}(l) = 2 \), \( \text{height}(R) = 3 \)

**Binary Tree Characteristics**

Every node in a binary tree has 0, 1, or 2 children

Every node in a proper binary tree has 0 or 2 children

Every level in a perfect binary tree is fully populated

Every level except the lowest in a complete binary tree is fully populated; the lowest level is populated left to right
**Binary Tree Representation**

A binary tree can be represented as a linked structure:

```c
struct Node {
    Item item;
    Node *left, *right;
};
```

Efficient for moving down a tree from parent to child.

How to move up the tree?

How to remove a node from, and add a node to, a binary tree?

---

**Tree Traversal**

The expression \(a/b + (c - d)e\) has been encoded as \(T\).

How would you traverse \(T\) to re-create (print out) the expression?

- to ensure correct evaluation precedence, enclose the printout of each subtree in parentheses, e.g., \(((a)/(b)) + (((c) - (d)) \times (e)))\)

Write a pseudo-code recursive function to print \(T\):

```c
void rtraverse(Node *root)
{
    if (node->right) {
        stack.push(rparen);
        stack.push(node->right);
        stack.push(lparen);
        node->right = NULL;
    }
    push(node);
    if (node->left) {
        stack.push(rparen);
        stack.push(node->left);
        stack.push(lparen);
        node->left = NULL;
    }
    do {
        if (!node->right &&
            !node->left) {
            print(node);)
        } else {
            print(lparen);
            do {
                if (node->right) {
                    stack.push(rparen);
                    stack.push(node->right);
                    stack.push(lparen);
                    node->right = NULL;
                }
                push(node);
                if (node->left) {
                    stack.push(rparen);
                    stack.push(node->left);
                    stack.push(lparen);
                    node->left = NULL;
                }
            } while (node = pop());
        } else if (node->right) {
            stack.push(rparen);
            stack.push(node->right);
            stack.push(lparen);
            node->right = NULL;
        }
    }
    print(rparen);
}
```
Tree Traversal

Aside from in-order depth-first traversal, we could also traverse the tree depth-first pre-order or post-order:

- **in-order**: visit $T_h$, visit node, visit $T_r$
- **pre-order**: visit node, visit $T_h$, visit $T_r$
- **post-order**: visit $T_r$, visit $T_h$, visit node

Which traversal order will give you Reverse Polish Notation (RPN)?

$ab/cd\cdot e^+\cdot$

And the Polish Notation?

$+/ab\cdot cde$.

Breadth-first traversal visits the tree level by level.

How would you implement breadth-first traversal?

Tree Sizes

A tree may be empty.

**External node**: an empty node with no children.

**Internal node**: a node with children.

**Leaf node**: an internal node whose children are all external nodes.

[Sometimes, outside this course, internal node simply means non-leaf node.]

In general, how many external nodes does an $N$-ary tree with $n$ internal nodes have?

An $N$-ary tree: a tree with degree $N$ (each node can have a maximum of $N$ children).

Tree Sizes

How many external nodes does an $N$-ary tree with $n$ internal nodes have?

<table>
<thead>
<tr>
<th>$n$</th>
<th>binary</th>
<th>tertiary</th>
<th>4-ary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Every new internal node replenishes one external node and brings with it $N-1$ new external nodes.

For $n$ internal nodes, we have $1+n(N-1)$ external nodes.

For binary tree, $n$ internal nodes means $n+1$ external nodes $\Rightarrow$ maximum $\text{ceil}(n/2)$ leaf nodes.

How many internal nodes does an $N$-ary tree with $m$ external nodes have?

Study Questions

1. How many links are there in an $N$-ary tree with $n$ internal nodes?
2. What is the maximum height of a binary tree of $n$ internal nodes?
3. How many internal nodes does it take to fully populate level $l$ of a binary tree?
4. What do you call a tree of $l$ levels that are fully populated?
5. Identify any proper, perfect, and complete binary tree in the figure:

![Binary Tree Diagram]

6. How many internal nodes are there in a perfect binary tree of height $h$ ($h+1$ levels)?
7. How many levels of a binary tree are needed to hold $n$ internal nodes?
8. What is the minimum height of a binary tree of $n$ internal nodes?
9. Is the height of the root node of a subtree the same as the depth of the subtree?
Binary Search Trees (BST)

A BST is a binary tree that
• has a key associated with each of
  its internal node, and that
• the key in any node is \( > \) the keys
  in all nodes in its left subtree and
• \( < \) the keys in all nodes in its right
  subtree,
• where \( '<' \) and \( '>' \) can be user
  defined

Implements sorted dictionary
with \( O(\log N) \) complexity for
both insert and search

Binary Search Trees Representation

A binary search tree can be
represented as a linked structure:

```c
struct Node {
    Item item;
    Node *left, *right;
};
typedef Node *Link;
```
Efficient for moving down a tree
to search for an item

How to remove a node from, and
add a node to, a binary search tree?

BST Search: Recursive

```c
Item BST::
rsearch(Node *root, Key &searchkey)
{

}
BST::rsearch() called with pointer to root and key
```

BST Search: Iterative

```c
Item BST::
isearch(Node *root, Key &searchkey)
{

}
BST::isearch() with minimal change to
BST::rsearch()
We will refer to both as BST::search()
```
BST Insert

1st (Bad) Attempt

- If new item has a key smaller than root's, recursive call on left subtree
- Else recursive call on right subtree
- Insert new item as leaf node

Example: insert(root, Item('b'));

```
void BST::
    insert(Node *root, Item newitem)
    {
        if (root == NULL) {
            new Node(newitem);
            // how to update parent
            // to point to this new child?
            return;
        }

        if (newitem.key < root->item.key) {
            insert(root->left, newitem);
        } else if (newitem.key > root->item.key) {
            insert(root->right, newitem);
        }
    }
```

What to do with duplicates?

```
typedef Node *Link;

void BST::
    insert(Link &root, Item newitem)
    {
        if (root == NULL) {
            root = new Node(newitem);
            return;
        }

        if (newitem.key < root->item.key) {
            insert(root->left, newitem);
        } else if (newitem.key > root->item.key) {
            insert(root->right, newitem);
        }
    }
```

BST Insert

```
BST::insert() called with double pointer to root and item to be inserted
if at leaf, and only at leaf, insert
(note the nifty use of reference args!)
if new item has a key smaller than that of root's, recursive call on left subtree
else recursive call on right subtree
```

BST Removal

After the removal of a node, the tree must remain a BST

1. Find the node to be removed
2. If node is a leaf node, remove, done
3. If node has a single child, replace node to be removed with child, done
4. If node has 2 children, find the smallest element in right child, called the in-order successor (find_ios())
5. Swap with in-order successor, repeat Steps 2 and 3 (Instead of in-order successor, Steps 4 and 5 can also use in-order predecessor, the largest element in the left child)

```
void BST::
    remove(Link &root, Key &searchkey)
    {
        if (root == NULL) return; // item not found
        key = root->item.key; // look for item
        if (searchkey < key) {
            remove(root->left, searchkey);
        } else if (searchkey > key) {
            remove(root->right, searchkey);
        } else if (searchkey == key) {
            if (isleaf(root)) {
                // e.g., rm f, m, or y
                delete root; root = NULL;
            } else { // what to do? see next page
```
BST Removal

```c
else {
    if (root->right == NULL) { // rm
        Node *temp = root;
        root = root->left;
        delete temp; return;
    }
    if (root->left == NULL) { // rm
        Node *temp = root;
        root = root->right;
        delete temp; return;
    }
    Link *ios = find_ios(root->right); // rm
    Node *temp = *ios;
    root->item = temp->item;
    *ios = temp->right; // null ok
    delete temp;
    // or swap root and *ios instead of copying item
}
}
```
**BST Sort**

```cpp
void BST::printsorted(Link root)
{
    if (root == NULL) return;
    printsorted(root->left);
    print(root);
    printsorted(root->right);
}
```

What kind of tree traversal does `BST::printsorted()` perform?

---

**BST Rank Search**

Given a BST, where is the smallest item?

Where is the largest item?

Would the node containing the largest/smallest item always be a leaf node?

How would you find the k-th largest item? E.g., find 2\textsuperscript{nd}, 3\textsuperscript{rd}, and 6\textsuperscript{th} largest items

---

**BST Add Count**

A BST node with count:

```cpp
struct Node {
    Item item;
    int count;
    Node *left, *right;
};
typedef Node *Link;
```

Let `count` counts the number of a node's descendants (nodes at and below the current node in tree)

---

**BST Insert with Count**

How would you modify `BST::insert()` to keep track of the count?

```cpp
void BST::insert(Link &root, Item newitem)
{
    if (root == NULL) {
        root = new Node(newitem);
        return;
    }
    if (newitem.key < root->item.key)
        insert(root->left, newitem);
    else insert(root->right, newitem);
}
```
BST Rank Search: Idea

A: If there are more than \( k \) items in the right subtree, the \( k \)-th largest item must be in the right subtree.

B: Else, there are \( m \) (< \( k \)) items in the right subtree, if the \( k \)-th item is not in the root node, find the \((k-m-1)\)-th largest item in the left subtree.

**BST Rank Search**

Node.rightcount() returns right->count if right is non-null, else returns 0

Link BST:
findkth(Link root, int rank)
{
    if (root == NULL) return root;
    if (root->rightcount() - rank >= 0)
        return findkth(root->right, rank);
    else
    {
        rank -= (root->rightcount() + 1);
        if (rank == 0) return root;
        else return findkth(root->left, rank);
    }
}

T: Find 2\textsuperscript{nd}, 3\textsuperscript{rd}, and 6\textsuperscript{th} largest items

Find 2\textsuperscript{nd} smallest item?