Why Do We Need Trees?

- Lists, Stacks, and Queues are linear relationships.
- Information often contains hierarchical relationships:
  - File directories or folders on your computer
  - Moves in a game
  - Employee hierarchies in organizations
- Can build a tree to support fast searching

Tree Jargon

- root
- nodes and edges
- leaves
- parent, children, siblings
- ancestors, descendants
- subtrees
- path, path length
- height, depth

More Tree Jargon

- Length of a path = number of edges
- Depth of a node N = length of path from root to N
- Height of node N = length of longest path from N to a leaf
- Depth of tree = depth of deepest node
- Height of tree = height of root

Definition and Tree Trivia

- A tree is a set of nodes
- that is an empty set of nodes, or
- has one node called the root from which zero or more trees (subtrees) descend
- A tree with N nodes always has N-1 edges
- Two nodes in a tree have at most one path between them
### Paths
- Can a non-zero path from node N reach node N again?
  - No. Trees can never have cycles (loops)
- Does depth of nodes in a non-zero path increase or decrease?
  - Depth always increases in a non-zero path

### Implementation of Trees
- One possible pointer-based Implementation
  - tree nodes with value and a pointer to each child
  - but how many pointers should we allocate space for?
- A more flexible pointer-based implementation
  - 1st Child / Next Sibling List Representation
  - Each node has 2 pointers: one to its first child and one to next sibling
  - Can handle arbitrary number of children

### Arbitrary Branching
```
  A
 /\  
B   C
 \_/ 
  E F
```

### Application: Arithmetic Expression Trees
Example Arithmetic Expression:
```
A + (B * (C / D))
```
How would you express this as a tree?

### Application: Arithmetic Expression Trees
Example Arithmetic Expression:
```
A + (B * (C / D))
```

### Traversing Trees
- Preorder: Node, then Children recursively
  - $A + B * C / D$
- Inorder: Left child recursively, Node, Right child recursively (Binary Trees)
  - $A = B * C / D$
- Postorder: Children recursively, then Node
  - $A B C D / * +$
Binary Trees

• Every node has at most two children
  › Most popular tree in computer science
  › Easy to implement, fast in operation
• Given N nodes, what is the minimum depth of a binary tree?
  › At depth d, you can have \( N = 2^d \) to \( 2^{d+1} - 1 \) nodes
  \[ 2^d \leq N \leq 2^{d+1} - 1 \implies d = \lfloor \log_2 N \rfloor \]

Minimum depth vs node count

• At depth d, you can have \( N = 2^d \) to \( 2^{d+1} - 1 \) nodes
• minimum depth d is \( \Theta(\log N) \)

\[ t(n) = \Theta(f(n)) \text{ means } t(n) = O(f(n)) \text{ and } f(n) = O(T(n)) \]

\[ d = 2 \implies N = 2^d \text{ to } 2^{d+1} - 1 \text{ (i.e., 4 to 7 nodes)} \]

Maximum depth vs node count

• What is the maximum depth of a binary tree?
  › Degenerate case: Tree is a linked list!
  › Maximum depth = \( N-1 \)
• Goal: Would like to keep depth at around \( \log N \) to get better performance than linked list for operations like Find

A degenerate tree

A linked list with high overhead and few redeeming characteristics

Binary Search Trees

• Binary search trees are binary trees in which
  › all values in the node’s left subtree are less than node value
  › all values in the node’s right subtree are greater than node value
• Operations:
  › Find, FindMin, FindMax, Insert, Delete

Operations on Binary Search Trees

• How would you implement these?
  › Recursive definition of binary search trees allows recursive routines
  › Call by reference helps too
• FindMin
• FindMax
• Find
• Insert
• Delete
**Binary Search Tree**

![Binary Search Tree Diagram]

**Find**

\[
\text{Find}(T: \text{tree pointer}, x: \text{element}): \text{tree pointer} \{
\text{case} \{
T = \text{null} : \text{return null;}
T.\text{data} = x : \text{return } T;
T.\text{data} > x : \text{return Find}(T.\text{left}, x);
T.\text{data} < x : \text{return Find}(T.\text{right}, x)
\}
\}
\]

**FindMin**

- Class Participation
- Design recursive FindMin operation that returns the smallest element in a binary search tree.

  \[
  \text{FindMin}(T: \text{tree pointer}): \text{tree pointer} \{ 
  \text{// precondition: } T \text{ is not null} \}
  \]

**Insert Operation**

- Insert(T: tree, X: element):
  - Do a “Find” operation for X
  - If X is found update duplicates counter
  - Else, “Find” stops at a NULL pointer
  - Insert Node with X there

- Example: Insert 95

**Insert 95**

![Insert 95 Diagram]

**Insert Done Very Elegantly**

Insert(T: reference tree pointer, x: element): integer:

1. if T = null then
   1.1 T := new tree; T.data := x; return 1
2. case:
   2.1 T.data = x : return 0;
   2.2 T.data > x : return Insert(T.left, x);
   2.3 T.data < x : return Insert(T.right, x);

3. Advantage of reference parameter is that the call has the original pointer not a copy.
Call by Value vs Call by Reference

- Call by value
  - Copy of parameter is used
  - F(p) used inside call of F

- Call by reference
  - Actual parameter is used

Delete Operation

- Delete is a bit trickier... Why?
- Suppose you want to delete 10?
- Strategy:
  - Find 10
  - Delete the node containing 10
- Problem: When you delete a node, what do you replace it by?

Delete Operation

- Problem: When you delete a node, what do you replace it by?
- Solution:
  - If it has no children, by NULL
  - If it has 1 child, by that child
  - If it has 2 children, by the node with the smallest value in its right subtree (the successor of the node)

Delete “5” - No children

Find 5 node

Then Free the 5 node and NULL the pointer to it

Delete “24” - One child

Find 24 node

Then Free the 24 node and replace the pointer to it with a pointer to its child

Delete “10” - Two children

Find 10, Copy the smallest value in right subtree into the node

Then recursively delete node with smallest value in right subtree
Note: it does not have two children
Delete “11” - One child

Remember 11 node

Then Free
the 11 node and
replace the
pointer to it with
a pointer to its
child

FindMin Solution

FindMin(T : tree pointer) : tree pointer {
  // precondition: T is not null
  if T.left = null return T
  else return FindMin(T.left)
}