Trees

CSE 326
Data Structures
Unit 4

Reading: Chapter 4.1-4.3

Why Do We Need Trees?

- Lists, Stacks, and Queues are linear relationships
- Information often contains hierarchical relationships
  - › File directories or folders
  - › Moves in a game
  - › Hierarchies in organizations

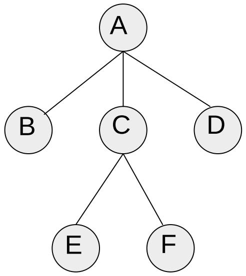
Tree Jargon

- root
- nodes and edges
- leaves

- parent, children, siblings
- ancestors, descendants

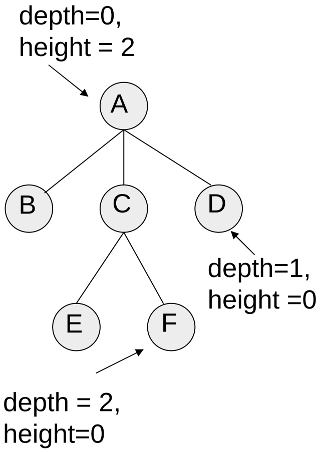
- subtrees

- 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



depth=0,
height = 2

depth=1,
height =0

depth = 2,
height=0

Definition and Tree Trivia

- A tree is a set of nodes,i.e., either
  - it's an empty set of nodes, or
  - it has one node called the root from which zero or more trees (subtrees) descend
- Two nodes in a tree have at most one path between them
- Can a non-zero path from node N reach node N again?

No. Trees can never have cycles (loops)

Paths

- A tree with N nodes always has N-1 edges (prove it by induction)

Base Case: N=1

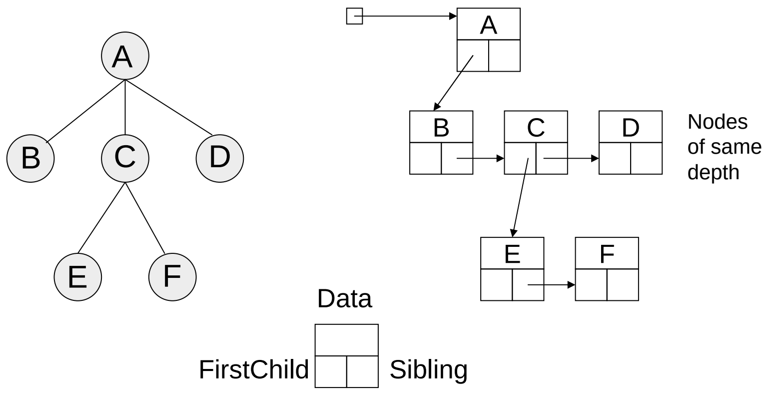
Inductive Hypothesis: Suppose that a tree with N=k nodes always has k-1 edges.

Induction: Suppose N=k+1…

Implementation of Trees

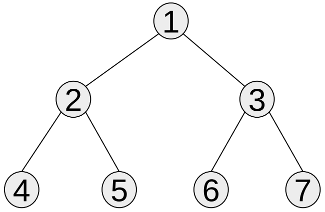
- One possible pointer-based Implementation
  - tree nodes with value and a pointer to each child
  - but how many pointers should we allocate space for?
- A more flexible pointer-based implementation
  - 1st Child / Next Sibling List Representation
  - Each node has 2 pointers: one to its first child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

Arbitrary Branching



Binary Trees

- Every node has at most two children
  - › Most popular tree in computer science
- Given N nodes, what is the minimum depth of a binary tree?



Binary Trees

- › At depth 0 (the root) there is one node.
- › At depth 1, there are two nodes.
- › At depth k, there are 2^k nodes
- › At depth d (tree depth), there might be 1 to 2^d nodes.

N is the total so

$1+2+\ldots+2^{(d-1)}+1 \le N \le 1+2+\ldots+2^{(d-1)}+2^d$

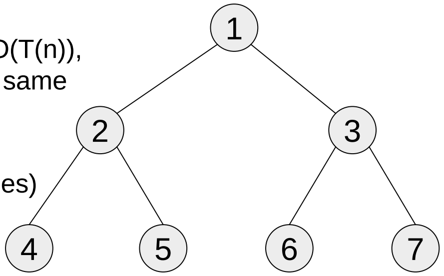
⇨ $2^d \le N \le 2^{d+1}-1$ implies $d_{min} = \lfloor \log_2 N \rfloor$

Minimum depth vs node count

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

$T(n) = \Theta(f(n))$ means
$T(n) = O(f(n))$ and $f(n) = O(T(n))$,
i.e. T(n) and f(n) have the same growth rate

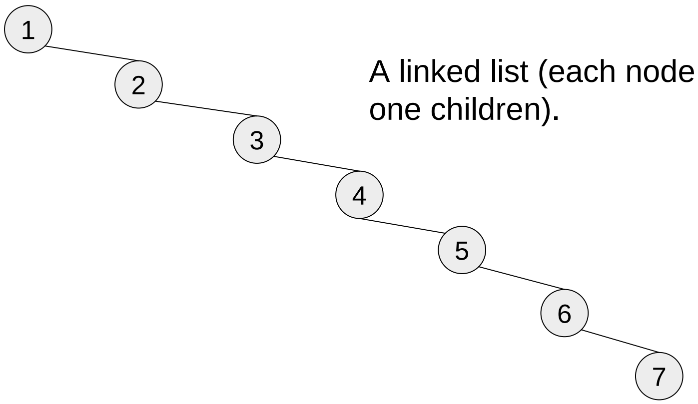
d=2
$N=2^2$ to 2^3-1 (i.e, 4 to 7 nodes)



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

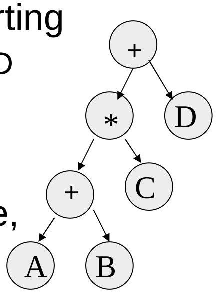


A linked list (each node has one children).

Traversing Binary Trees

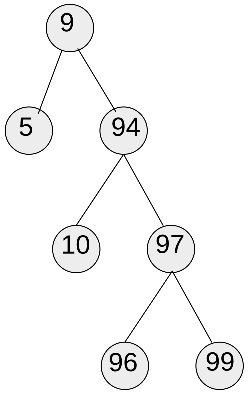
- The definitions of the traversals are recursive definitions. For example:
  - › Visit the root
  - › Visit the left subtree (i.e., visit the tree whose root is the left child) and do this recursively
  - › Visit the right subtree (i.e., visit the tree whose root is the right child) and do this recursively
- Traversal definitions can be extended to general (non-binary) trees

Traversing Binary Trees



- Preorder: Node, then Children (starting with the left) recursively + * + A B C D
- Inorder: Left child recursively, Node, Right child recursively A + B * C + D
- Postorder: Children recursively, then Node A B + C * D +

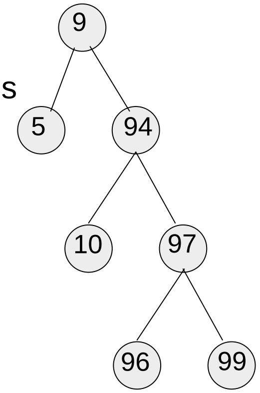
Binary Search Trees



- 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

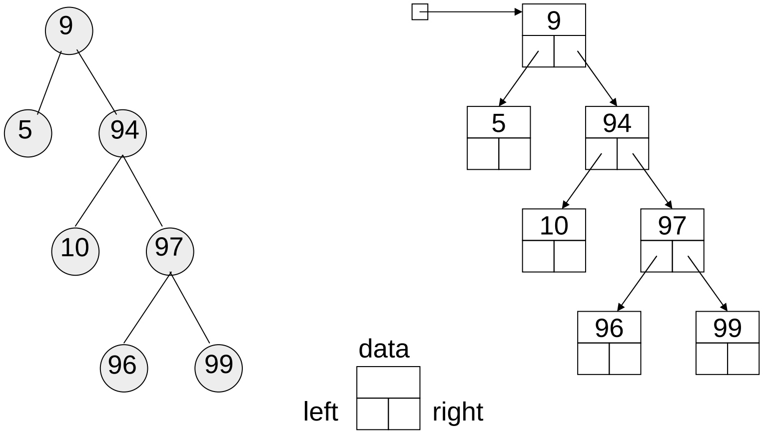
What happens when we traverse the tree in inorder?

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 SearchTree



Find

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

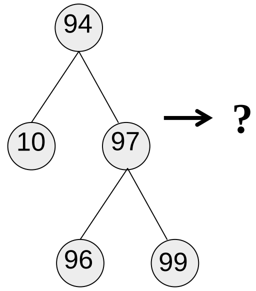
FindMin

- Design recursive FindMin operation that returns the smallest element in a binary search tree.
  - `FindMin(T : tree pointer) : tree pointer {`

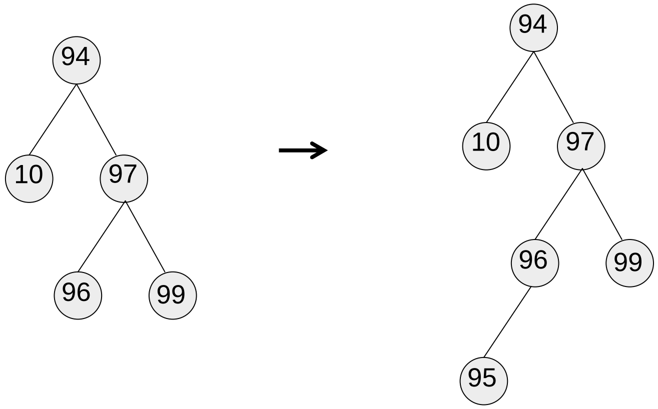
```
FindMin(T : tree pointer) : tree pointer {
// precondition: T is not null //
???
}
```

Insert Operation

- **`Insert(T: tree, X: element)`**
  - Do a “Find” operation for X
  - If X is found à update (no need to insert)
  - Else, “Find” stops at a NULL pointer
  - Insert Node with X there
- Example: Insert 95



Insert 95



Insert Done with call-by-reference

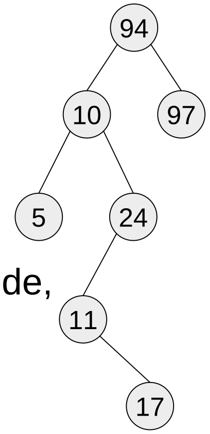
```
Insert(T : reference tree pointer, x : element) : integer {
if T = null then
  T := new tree; T.data := x; return 1;//the links to
                                       //children are null
case
  T.data = x : return 0;
  T.data > x : return Insert(T.left, x);
  T.data < x : return Insert(T.right, x);
endcase
}
```

This is where call by reference makes a difference.

Advantage of reference parameter is that the call has the original pointer not a copy.

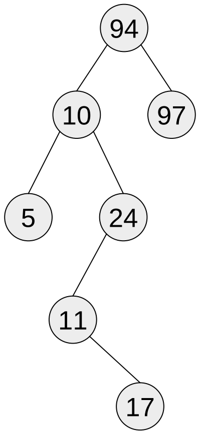
Delete Operation

- 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?



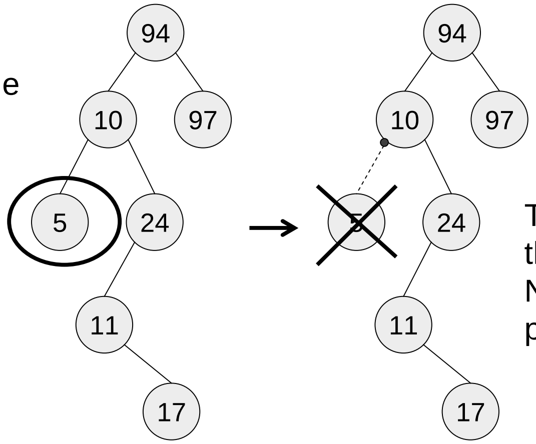
Delete Operation

- 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

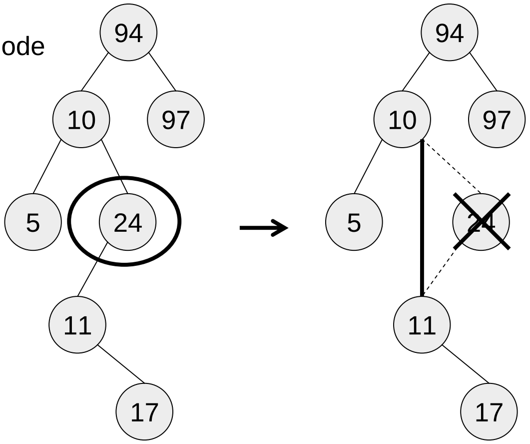
Find 5 node



Then Free
the 5 node and
NULL the
pointer to it

Delete “24” - One child

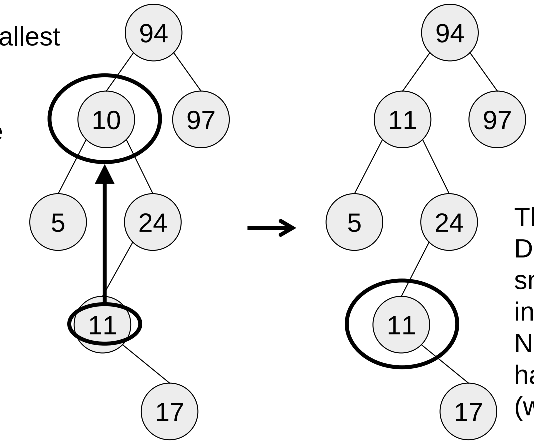
Find 24 node



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

Delete “10” - two children

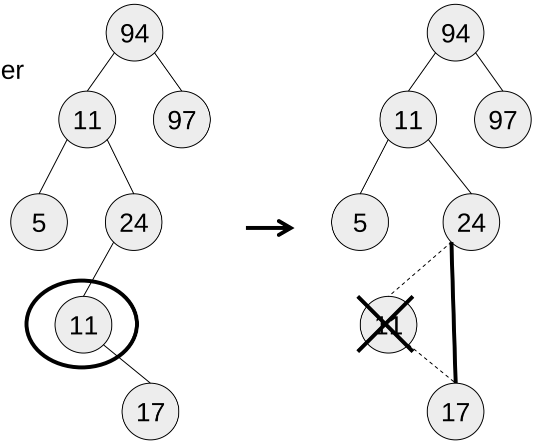
Find 10,
Copy the smallest
value in
right subtree
into the node



Then (recursively)
Delete node with
smallest value
in right subtree
Note: it cannot
have two children
(why?)

Then 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)
}
```