Trees

CSE 326
Data Structures
Lecture 6

Readings and References

- Reading
  - › Chapter 4.1-4.3,

Why Do We Need Trees?

- Lists, Stacks, and Queues are linear data structures
- Information often contains hierarchical relationships
  - › File directories or folders on your computer
  - › Moves in a game
  - › Employee hierarchies in organizations
- Trees support fast searching

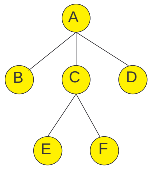
Tree Jargon

- root
- nodes and edges
- leaves

- parent, children, siblings
- ancestors, descendants

- subtrees

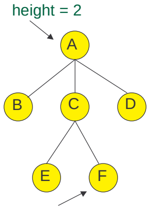
- path, path length
- height, depth



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
- **Height of tree** = height of root

depth=0,
height = 2



depth = 2,
height=0

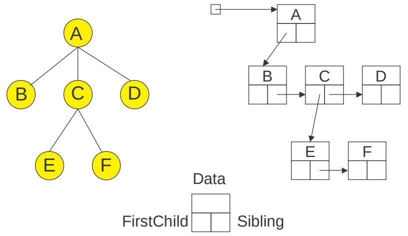
Definition and Tree Trivia

- 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

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



Application: Arithmetic Expression Trees

Example Arithmetic Expression:

A + (B * (C / D))

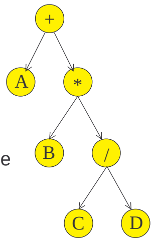
How would you express this as a tree?

Application: Arithmetic Expression Trees

Example Arithmetic Expression:

A + (B * (C / D))

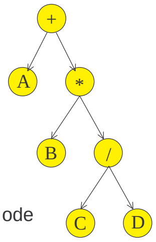
Tree for the above expression:



- Used in most compilers
- No parenthesis need – use tree structure
- Can speed up calculations e.g. replace / node with C/D if C and D are known
- Calculate by traversing tree (how?)

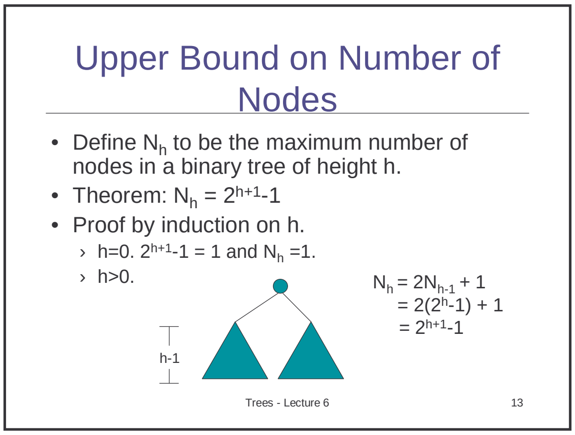
Traversing Trees

- 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

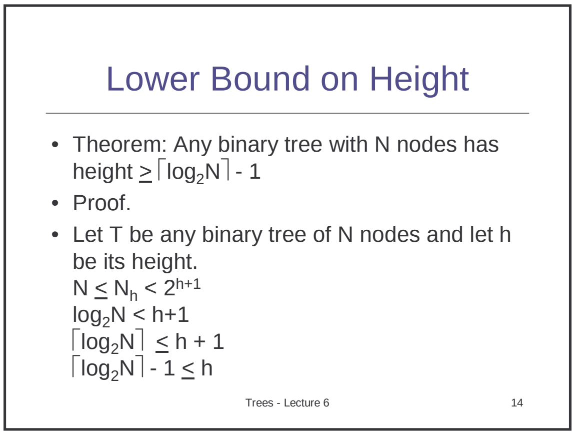
- Every node has at most two children
  - › Most popular tree in computer science
  - › Easy to implement, fast in operation
- Given N nodes, what is the minimum height of a binary tree?
  - › A height h tree has at most 2^{h+1}-1 nodes
  - › Hence, a binary tree with N node has height > $\log_2 N$ -1



Upper Bound on Number of Nodes

- Define N_h to be the maximum number of nodes in a binary tree of height h.
- Theorem: $N_h = 2^{h+1}-1$
- Proof by induction on h.
  - › h=0. $2^{h+1}-1 = 1$ and $N_h = 1$.
  - › h>0.

$$\begin{aligned} N_h &= 2N_{h-1} + 1 \\ &= 2(2^h-1) + 1 \\ &= 2^{h+1}-1 \end{aligned}$$



Lower Bound on Height

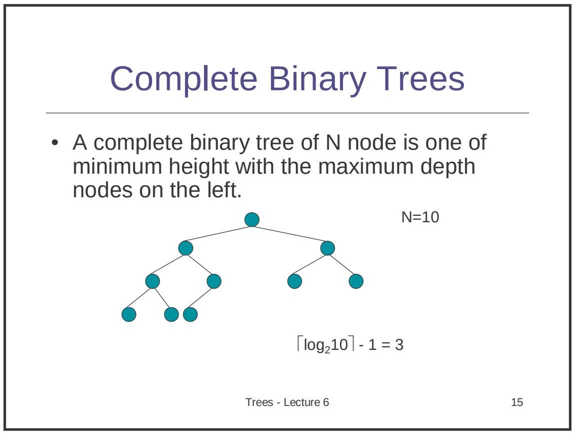
- Theorem: Any binary tree with N nodes has height $\geq \lceil \log_2 N \rceil - 1$
- Proof.
- Let T be any binary tree of N nodes and let h be its height.

  $N \leq N_h < 2^{h+1}$

  $\log_2 N < h+1$

  $\lceil \log_2 N \rceil \leq h + 1$

  $\lceil \log_2 N \rceil - 1 \leq h$

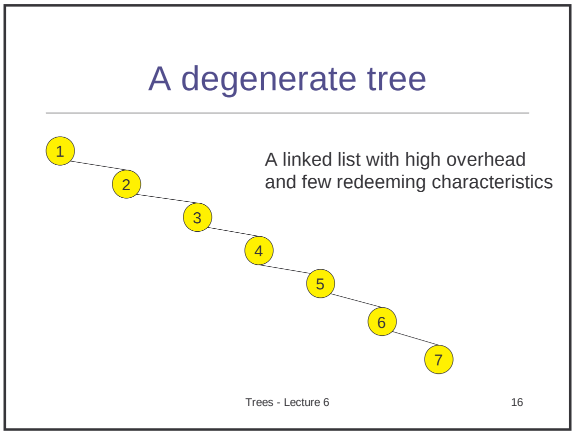


Complete Binary Trees

- A complete binary tree of N node is one of minimum height with the maximum depth nodes on the left.

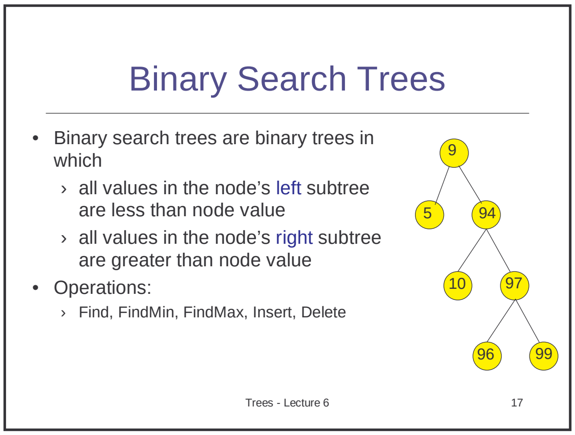
N=10

$\lceil \log_2 10 \rceil - 1 = 3$



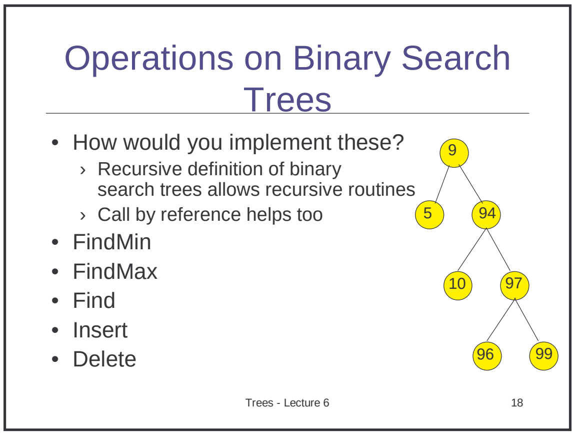
A degenerate tree

A linked list with high overhead and few redeeming characteristics



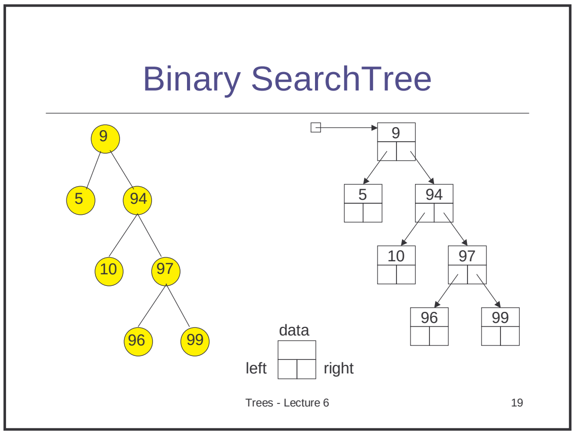
Binary Search Trees

- 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

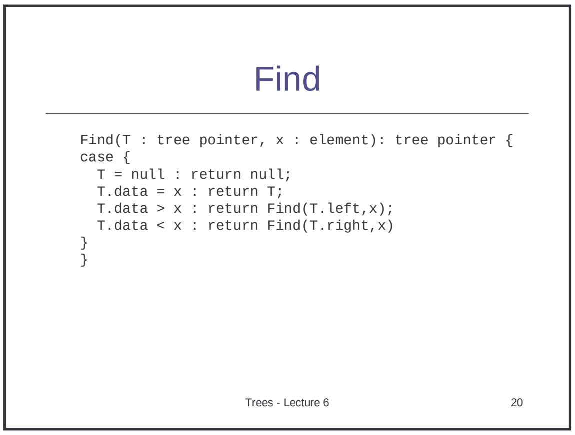


Operations on Binary Search Trees

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

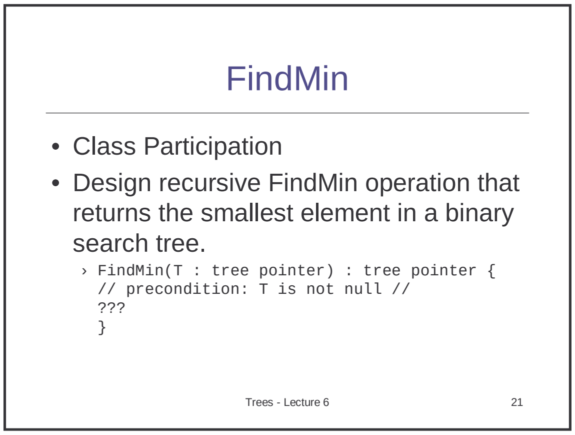


Binary SearchTree



Find

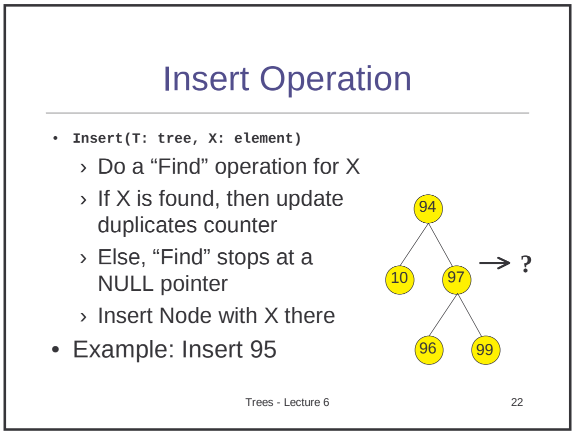
```
Find(T : tree pointer, x : element): tree pointer {
case {
  T = null : return null;
  T.data = x : return T;
  T.data > x : return Find(T.left,x);
  T.data < x : return Find(T.right,x)
}
}
```



FindMin

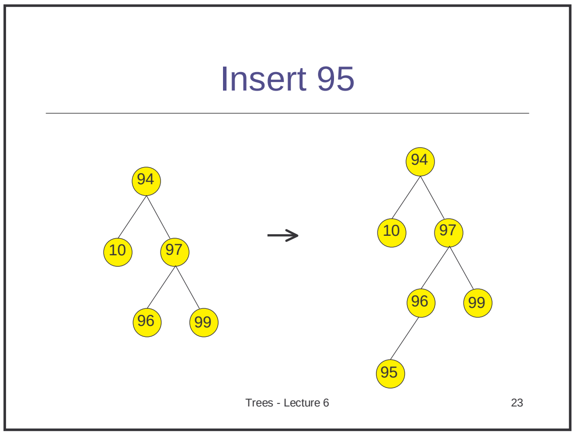
- Class Participation
- Design recursive FindMin operation that returns the smallest element in a binary search tree.
  - › FindMin(T : tree pointer) : tree pointer {

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

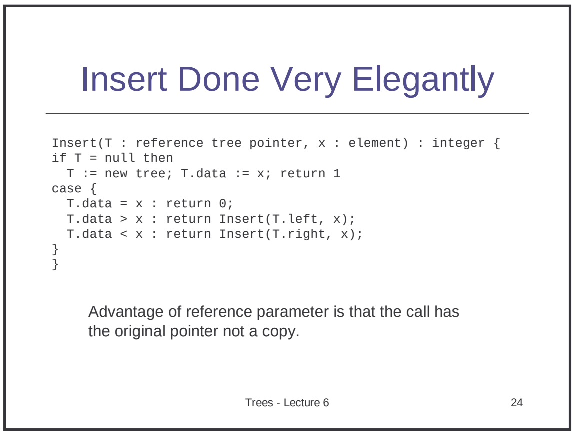


Insert Operation

- **Insert(T: tree, X: element)**
  - Do a “Find” operation for X
  - If X is found, then update duplicates counter
  - Else, “Find” stops at a NULL pointer
  - Insert Node with X there
- Example: Insert 95



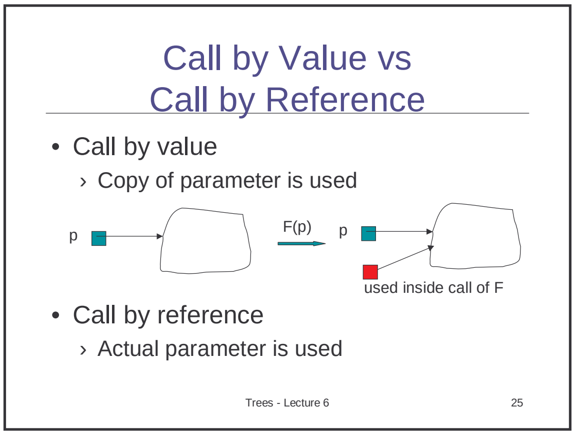
Insert 95



Insert Done Very Elegantly

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

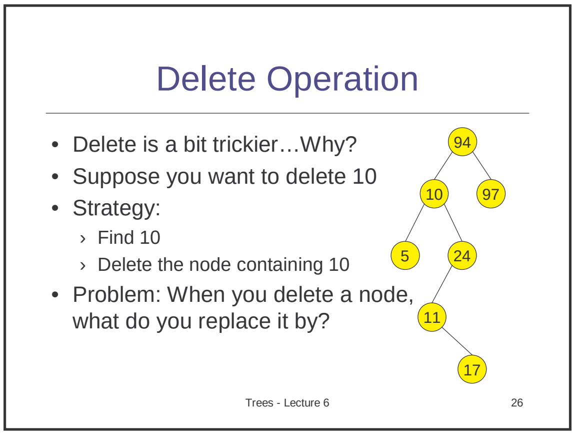
Advantage of reference parameter is that the call has the original pointer not a copy.



Call by Value vs Call by Reference

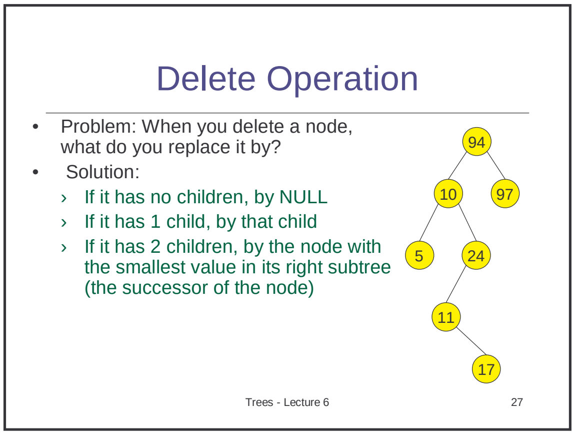
- Call by value
  - › Copy of parameter is used

- Call by reference
  - › Actual parameter is used



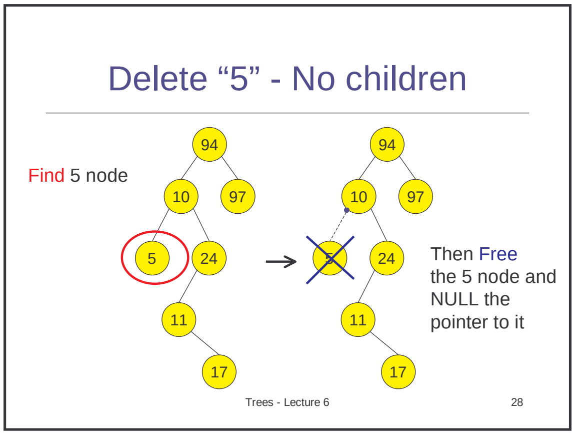
Delete Operation

- 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

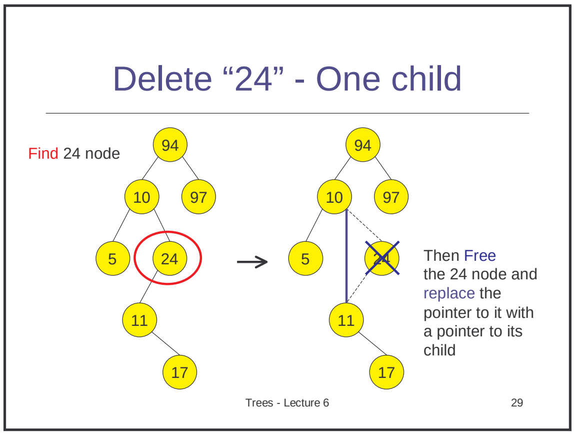
- Problem: When you delete a node, what do you replace it by?
- Solution:
  - › If it has no children, by NULL
  - › If it has 1 child, by that child
  - › If it has 2 children, by the node with the smallest value in its right subtree (the successor of the node)



Delete “5” - No children

Find 5 node

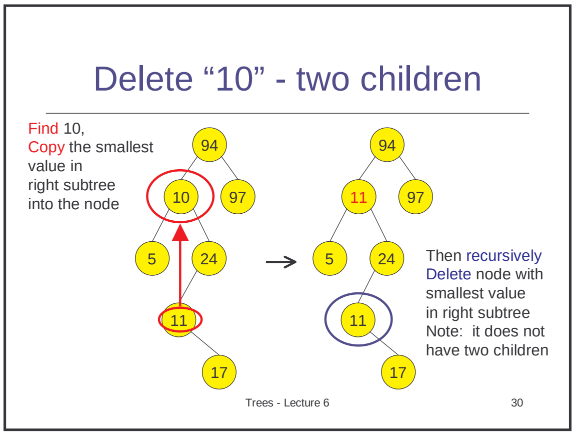
Then Free the 5 node and NULL the pointer to it



Delete “24” - One child

Find 24 node

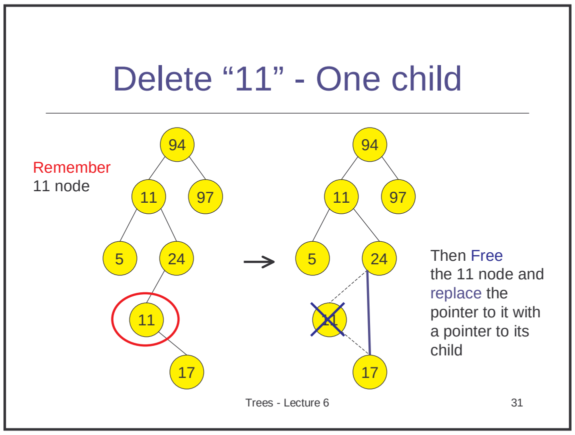
Then Free the 24 node and replace the pointer to it with a pointer to its child



Delete “10” - two children

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

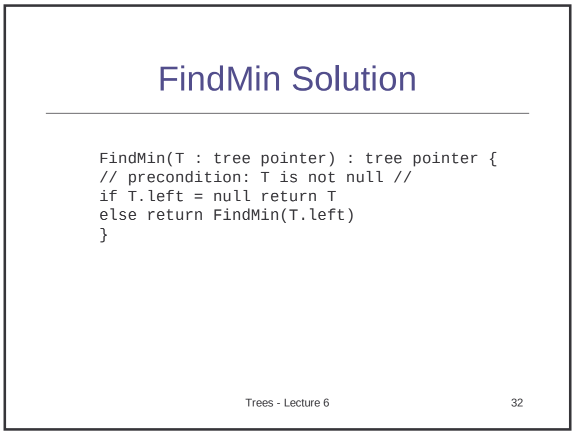
Then recursively Delete node with smallest value in right subtree Note: it does not have two children



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)
}
```