

Trees

CSE 373
Data Structures

Readings

- Reading
 - › Goodrich and Tamassia, Chapter 6.

3/28/04

Trees

2

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
- Can build a tree to support fast searching

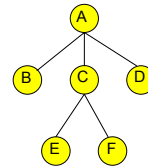
3/28/04

Trees

3

Tree Jargon

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



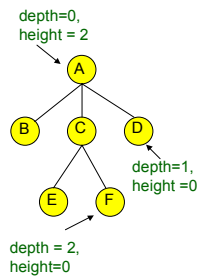
3/28/04

Trees

4

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



3/28/04

Trees

5

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)

3/28/04

Trees

6

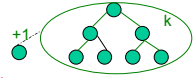
Paths

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

Base Case: N=1 one node, zero edges

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

Induction: Suppose N=k+1...
 The k+1st node must connect to the rest by 1 or more edges.
 If more, we get a cycle. So it connects by just 1 more edge



3/28/04

Trees

7

Implementation of Trees

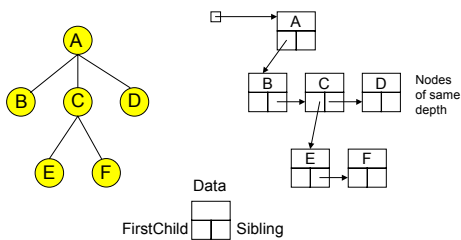
- One possible pointer-based Implementation
 - tree nodes with value and a pointer to each child
 - but how many pointers should we allocate space for?
- A more flexible pointer-based implementation
 - 1st Child / Next Sibling List Representation
 - Each node has 2 pointers: one to its first child and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

3/28/04

Trees

8

Arbitrary Branching



3/28/04

Trees

9

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? (This means all levels but the last are full!)
 - At depth d, you can have $N = 2^d$ to $N = 2^{d+1}-1$ nodes

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

3/28/04

Trees

10

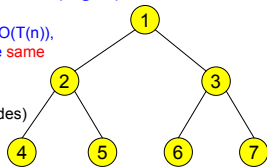
Minimum depth vs node count

- 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)



3/28/04

Trees

11

Maximum depth vs node count

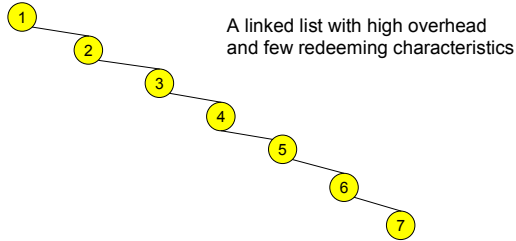
- What is the maximum depth of a binary tree?
 - Degenerate case: Tree is a linked list!
 - Maximum depth = N-1
- Goal: Would like to keep depth at around $\log N$ to get better performance than linked list for operations like Find

3/28/04

Trees

12

A degenerate tree



3/28/04

Trees

13

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

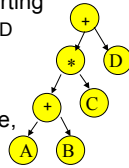
3/28/04

Trees

14

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 +



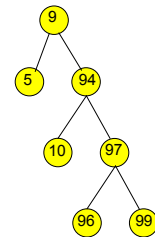
3/28/04

Trees

15

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?

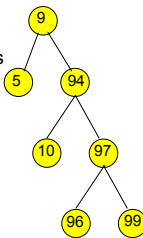
3/28/04

Trees

16

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

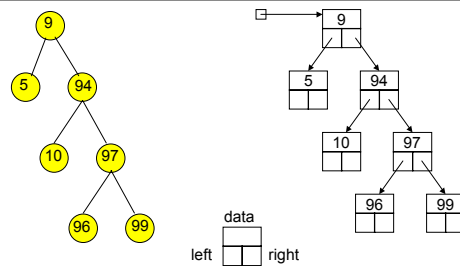


3/28/04

Trees

17

Binary Search Tree



3/28/04

Trees

18

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)
  }
}
```

3/28/04

Trees

19

FindMin

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

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

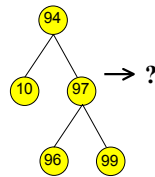
3/28/04

Trees

20

Insert Operation

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

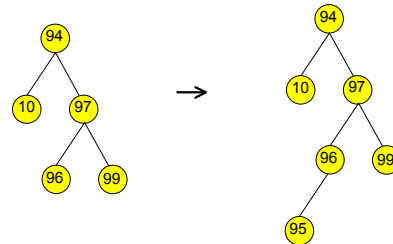


3/28/04

Trees

21

Insert 95



3/28/04

Trees

22

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.

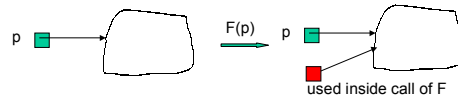
3/28/04

Trees

23

Call by Value vs Call by Reference

- Call by value
 - > Copy of parameter is used



- Call by reference
 - > Actual parameter is used

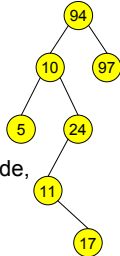
3/28/04

Trees

24

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?



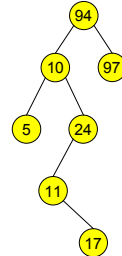
3/28/04

Trees

25

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)



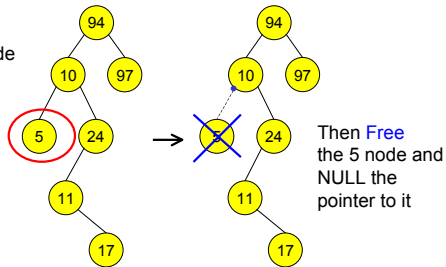
3/28/04

Trees

26

Delete "5" - No children

Find 5 node



Then Free the 5 node and NULL the pointer to it

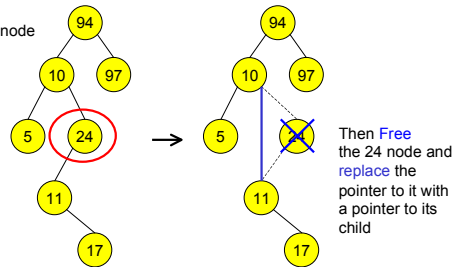
3/28/04

Trees

27

Delete "24" - One child

Find 24 node



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

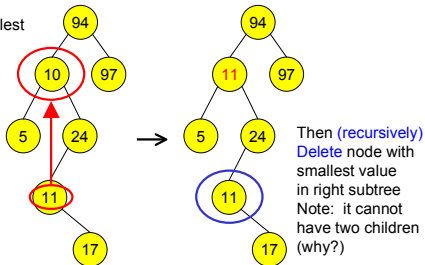
3/28/04

Trees

28

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?)

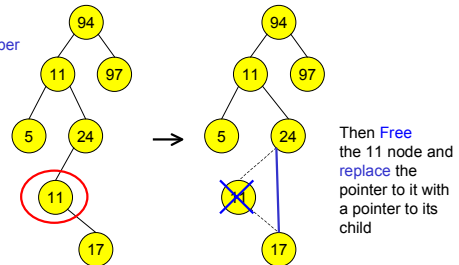
3/28/04

Trees

29

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

3/28/04

Trees

30

Remove from Text

```
private BinaryNode remove( Comparable x, BinaryNode t) {
    if ( t == null) return t;           // not found
    if ( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );   // search left
    else if ( x.compareTo( t.element ) > 0 )
        t.right = remove(x, t.right);   // search right
    else if ( t.left != null && t.right != null) // found it; two children
        { t.element = findMin( t.right ).element; // find the min, replace,
          t.right = remove( t.element, t.right); } // and remove it
    else t = (t.left != null ) ? t.left : t.right; // found it; one child
    return t; }

```

3/28/04

Trees

31

FindMin Solution

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

```

Note: Look at the "remove" method in the book.

3/28/04

Trees

32