CMPSC 24: Lecture 14
Trees, Binary Trees, & Binary Search Trees

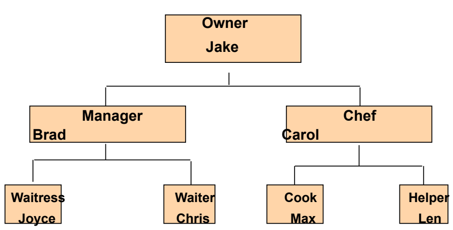
Divyakant Agrawal
Department of Computer Science
UC Santa Barbara

Lecture Plan

- Tree ADT
  - Binary Search Tree (BST) ADT

2

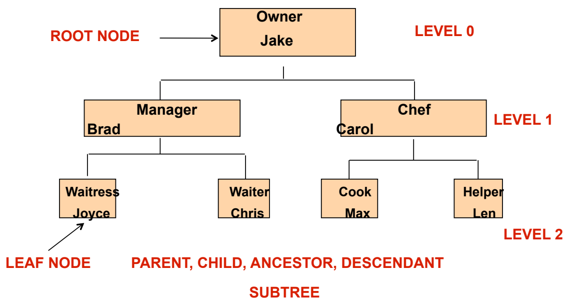
Jake's Pizza Shop



UNIQUE PATH BETWEEN NODES

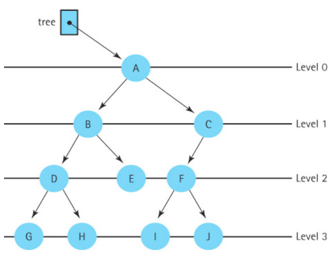
3

Nomenclature



4

Trees

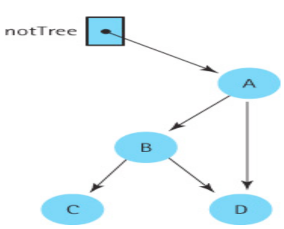


Level:
Distance of a node from root

Height:
The maximum level

5

Trees



Why is this not a tree ?

6

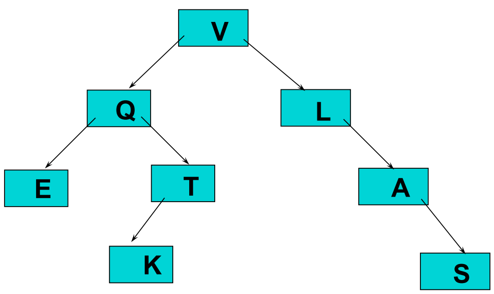
Binary Tree

A node can have at most two children.

The two children of a node are called the left child and the right child, if they exist.

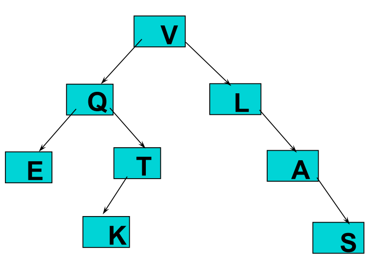
7

A Binary Tree



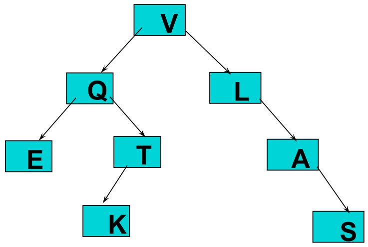
8

How Many Leaf Nodes?



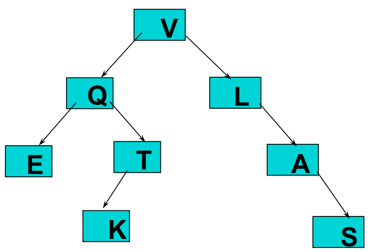
9

How Many Descendants of Q?



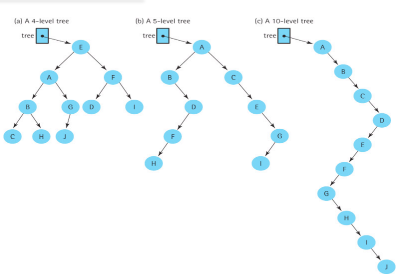
10

How Many Ancestors of K?



11

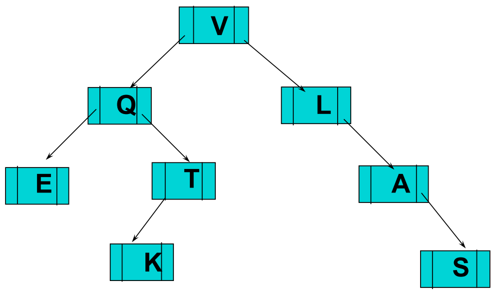
Trees



How many different binary trees can be made from 2 nodes? 4 nodes? 6 nodes?

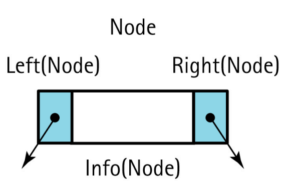
12

Implementing a Binary Tree with Pointers and Dynamic Data



13

Structure of a Tree Node



Possible to add
a parent pointer

14

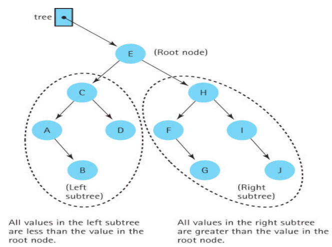
A Binary Search Tree (BST) is . . .

A special kind of binary tree in which:

1. **Each node contains a distinct data value,**
2. **The key values in the tree can be compared using "greater than" and "less than", and**
3. **The key value of each node in the tree is**
   less than every key value in its right subtree, and greater than every key value in its left subtree.

15

Binary Search Trees



Each node is the root of a subtree

How to define a tree and a BST in a recursive manner?

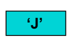
16

Shape of a Binary Search Tree . . .

Depends on its key values and their order of insertion.

Insert the elements 'J' 'E' 'F' 'T' 'A' in that order.

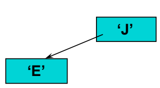
The first value to be inserted is put into the root node.



17

Inserting 'E' into the BST

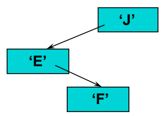
Thereafter, each value to be inserted begins by comparing itself to the value in the root node, moving left it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.



18

Inserting ‘F’ into the BST

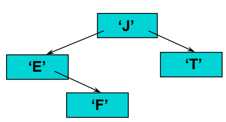
Begin by comparing ‘F’ to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



19

Inserting ‘T’ into the BST

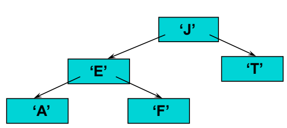
Begin by comparing ‘T’ to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



20

Inserting ‘A’ into the BST

Begin by comparing ‘A’ to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.

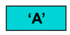


21

What BST . . .

is obtained by inserting

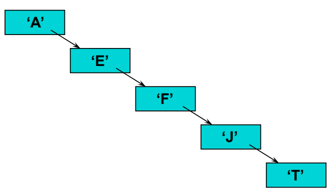
the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order?



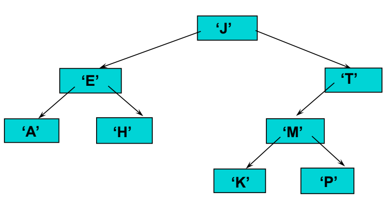
Binary Search Tree . . .

obtained by inserting

the elements ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order.



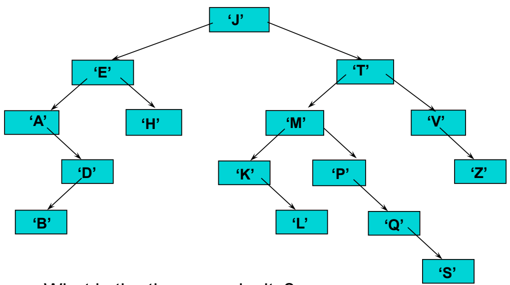
Another BST



Add nodes containing these values in this order:

‘D’ ‘B’ ‘L’ ‘Q’ ‘S’ ‘V’ ‘Z’

Is ‘F’ in the binary search tree?



What is the time complexity?

25

Tree ADT: IsFull and IsEmpty

```
bool TreeType::IsFull() const
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}

bool TreeType::IsEmpty() const
{
  return root == NULL;
}
```

How to Compute the Size of a Tree?

27

CountNodes(tree)

if tree is NULL
  return 0
else
  return CountNodes(Left(tree)) +
    CountNodes(Right(tree)) + 1

28

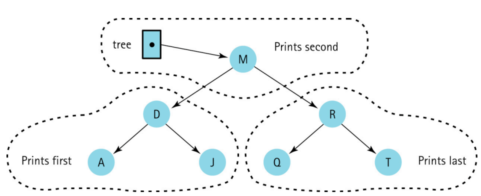
Implementation of LengthIs

```
int CountNodes(TreeNode* tree); // Prototype
int TreeType::LengthIs() const
{
  return CountNodes(root);
}

int CountNodes(TreeNode* tree)
// Recursive function that counts the nodes
{
  if (tree == NULL)
    return 0;
  else
    return CountNodes(tree->left) +
      CountNodes(tree->right) + 1;
}
```

Why do we need two functions?

Printing all the Nodes in Order



30

Function Print

Definition: Prints the items in the binary search tree in order from smallest to largest.

Base Case: If tree = NULL, do nothing.

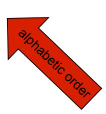
General Case: Traverse the left subtree in order.
Then print Info(tree).
Then traverse the right subtree in order.

Code for Recursive InOrder Print

```
void PrintTree(TreeNode* tree,
 std::ofstream& outFile)
{
 if (tree != NULL)
 {
  PrintTree(tree->left, outFile);
  outFile << tree->info;
  PrintTree(tree->right, outFile);
 }
}
```


Tree Traversals

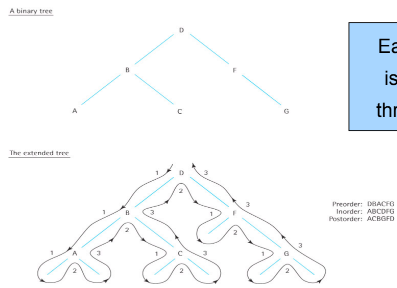
Inorder(tree)
if tree is not NULL
  Inorder(Left(tree))
  Visit Info(tree)
  Inorder(Right(tree))



PreOrder(tree)
if tree is not NULL
  Visit Info(tree)
  Preorder(Left(tree))
  Preorder(Right(tree))

PostOrder(tree)
if tree is not NULL
  Postorder(Left(tree))
  Postorder(Right(tree))
  Visit Info(tree)

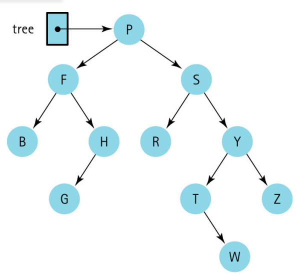
Traversals



Each node is visited three times

34

Traversals



Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

35

Iterator

The client program passes a parameter to ResetTree and GetNextItem indicating which of the three traversals to use

ResetTree generates a queues of node contents in the indicated order

GetNextItem processes the node contents from the appropriate queue: inQue, preQue, postQue

36

Iterator

```
void TreeType::ResetTree(OrderType order)
// Calls function to create a queue of the
   tree
// elements in the desired order.
{
  switch (order)
  {
    case PRE_ORDER : PreOrder(root, preQue);
                                break;
    case IN_ORDER     : InOrder(root, inQue);
                                break;
    case POST_ORDER: PostOrder(root, postQue);
                                break;
  }
}
```

37

Iterator

```
void TreeType::GetNextItem(ItemType& item,
     OrderType order, bool& finished)
{
  finished = false;
  switch (order)
  {
    case PRE_ORDER  : preQue.Dequeue(item);
                      if (preQue.IsEmpty())
                        finished = true;
                      break;
    case IN_ORDER   : inQue.Dequeue(item);
                      if (inQue.IsEmpty())
                        finished = true;
                      break;
    case  POST_ORDER: postQue.Dequeue(item);
                      if (postQue.IsEmpty())
                        finished = true;
                      break;
  }
}
```

Can you think of other implementations?