

3.1: linear DS: list, stack, queue.  
3.5: Non-linear DS: tree, graph  
6 weeks.

# Lecture 7: tree

---

# Our Roadmap

- ◆ Tree
  - ◆ Basic Concepts
  - ◆ Properties of Tree (focus on binary tree)
  - ◆ Binary Tree Traversal
- ◆ Binary Tree Applications
  - ◆ Algebraic expression
  - ◆ Huffman encoding

# Tree : organize data

- ◆ This lecture provides a formal definition of **trees**, which constitute an important approach to organize data in computer science. We will also prove some basic properties of trees that will be useful in computer science.

例: data dictionary

- ① ordered
- ② update.

→

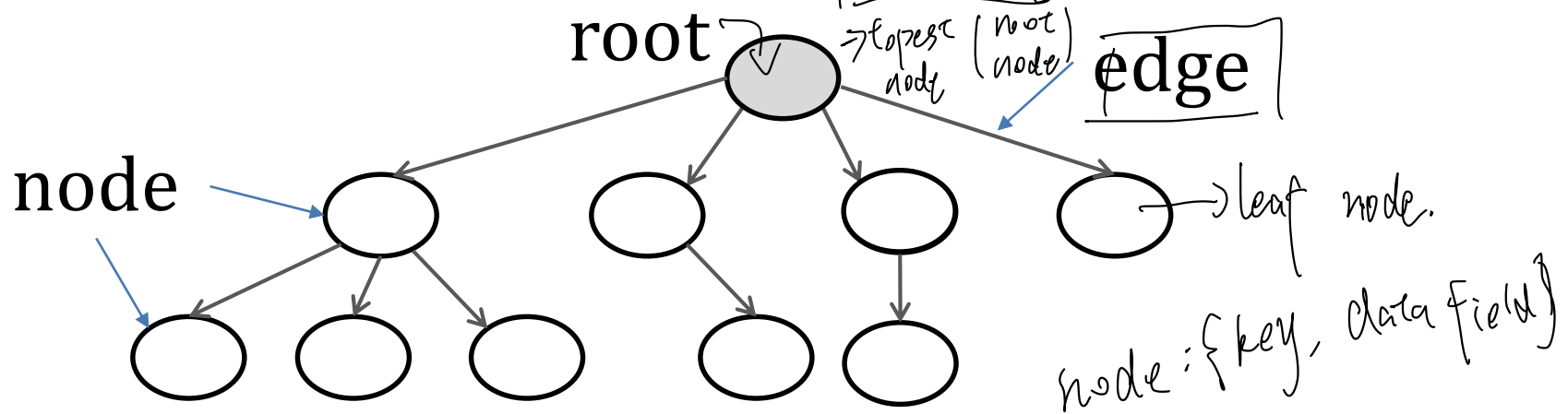
- ③ search
- ④ insert

A: linked list	① $O(n)$	② $O(n)$
B: Array	① $O(\log n)$	② $O(n)$
C: Tree	① $O(\log n)$	② $O(\log n)$

# Motivation

- ◆ Data Dictionary: maintain a sorted collection of data
  - ◆ *search* for an item (with possibly delete it)
  - ◆ *insert* a new item
- ◆ A list implemented using an array
  - ◆ Searching for an item,  $O(\log n)$
  - ◆ Inserting an item,  $O(n)$
- ◆ A list implemented using a linked list
  - ◆ Searching for an item,  $O(n)$
  - ◆ Inserting an item,  $O(n)$
- ◆ In the next few lectures, we will look at data structures (**trees**) that can be used for a more efficient data dictionary

# What is a tree?



- ◆ A tree consist of:
  - ◆ A set of nodes, *??*
  - ◆ A set of edges, each of which connects a pair of nodes
- ◆ Each node may have one or more data items
  - ◆ Key field = the field used when searching for a data item
  - ◆ Multiple data items with the same key are referred to as duplicates
- ◆ The node at the “top” of the tree is the “root” of the tree

# Tree Property I

◆ A tree with  $n$  nodes with  $n-1$  edges

◆ Proof?

$n=1$  时, we have  $edge=0$ . 从叶子节点删除,

当  $n=k$  时, 我们有  $edge=k-1$

树中没有边,

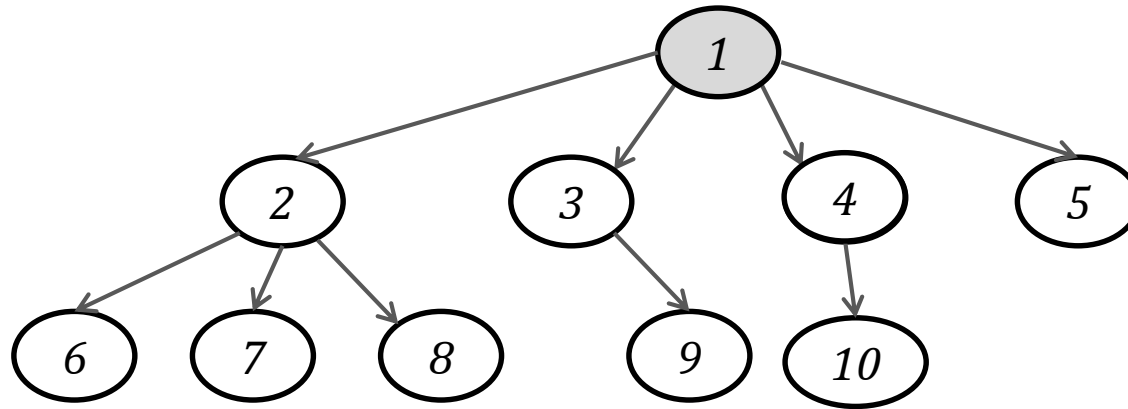
叶子节点只有1个边,

推论: 树上没有环. 环意味着  $edges \geq nodes$

边上的父节点是 parent 子节点是 son, 兄弟节点是

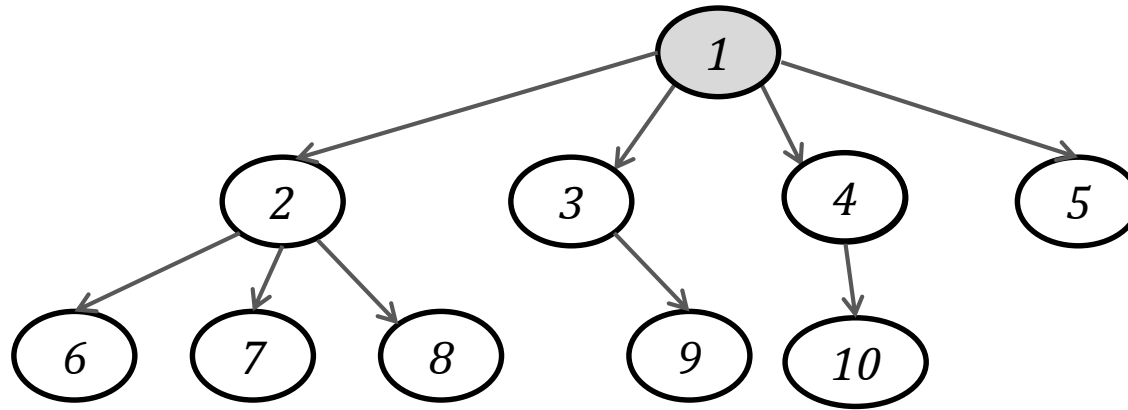
sibling: 678.

# Relationship between nodes



- ◆ Consider a tree  $T$ , let  $u$  and  $v$  be two nodes in  $T$ . We say that  $u$  is the parent of  $v$  if  $v$  is the node directly below  $u$
- ◆ Accordingly, we say that  $v$  is a *child* of  $u$ .
  - ◆ e.g., node 1 is the parent of node 2, 3, 4, 5, and node 2 is the child of node 1.
- ◆ Each node is the child of at most one parent
- ◆ Node with the same parent are siblings
  - ◆ e.g., node 2, 3, 4, 5 are siblings *6, 7, 8, 9, 10*

# Relationship between nodes



- ◆ Consider a tree  $T$ , let  $u$  and  $v$  be two nodes in  $T$ . We say that  $u$  is an ancestor of  $v$  if one of the following holds:

- ◆  $u = v$

- ◆  $u$  is the parent of  $v$ , or

- ◆  $u$  is the parent of an ancestor of  $v$ .

三条任一条即可

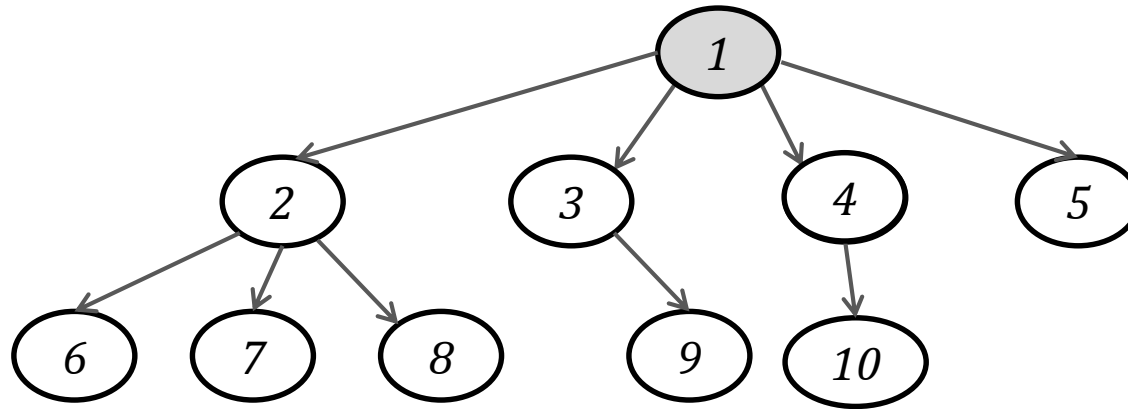
- ◆ Accordingly, we say that  $v$  is a descendant of  $u$ .

- ◆ In particular, if  $u \neq v$ ,  $u$  is a proper ancestor of  $v$ , and likewise,  $v$  is a proper descendant of  $u$ .

真祖先



# Tree node types



- ◆ A *leaf* node is a node without children
- ◆ An internal node is a node with one or more children
- ◆ E.g.,
  - ◆ Leaf nodes: 5, 6, 7, 8, 9, 10
  - ◆ Internal nodes: 1, 2, 3, 4

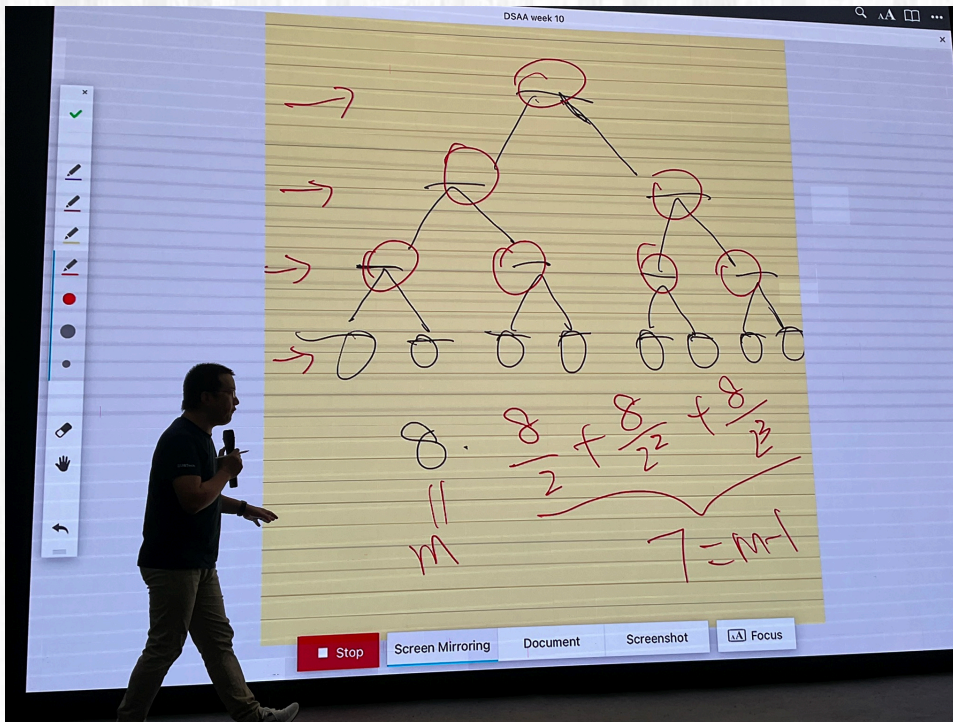
# Tree Property II

- Let  $T$  be a tree where every internal node has at least 2 child nodes. If  $m$  is the number of leaf nodes, then the number of internal nodes is at most  $m-1$ .

Handwritten notes and formula:

$$m \rightarrow \frac{m}{x} + \dots + \frac{m}{x^h}$$

$x \rightarrow$  2 or 3



## EASTERN CONFERENCE

[1] BOS 4

[4] WAS 3

[1] BOS 1

[2] CLE 4

[3] TOR 0

[2] CLE 4

[1] BOS 4

[8] CHI 2

[4] WAS 4

[5] ATL 2

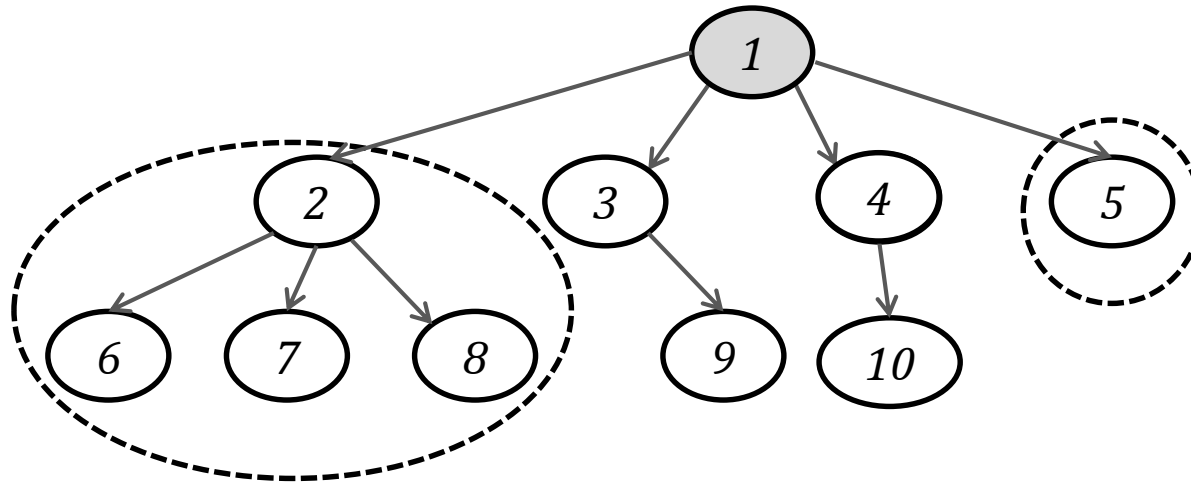
[3] TOR 4

[6] MIL 2

[2] CLE 4

[7] IND 0

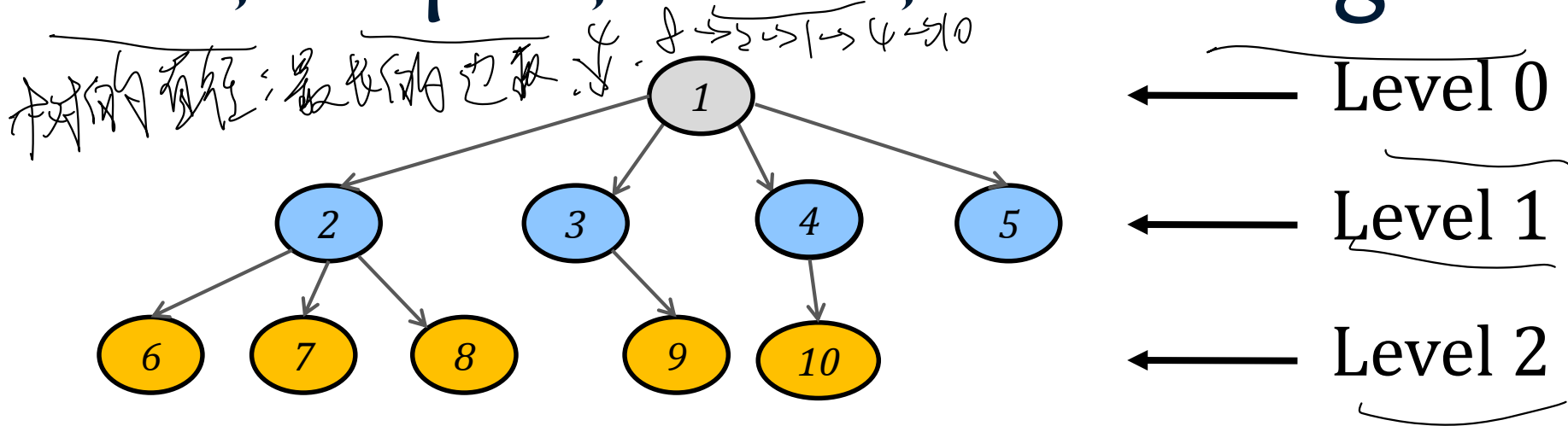
# Tree: a Recursive Data Structure



树的递归定义  
递归树

- ◆ Each node in the tree is the root of a smaller tree!
  - ◆ Refer to such trees as subtrees to distinguish them from the tree as a whole
  - ◆ Example: node 2 is the root of the subtree circled above
  - ◆ Example: node 5 is the root of a subtree with only one node.
- ◆ We will see that tree algorithms often lend themselves to recursive implementations

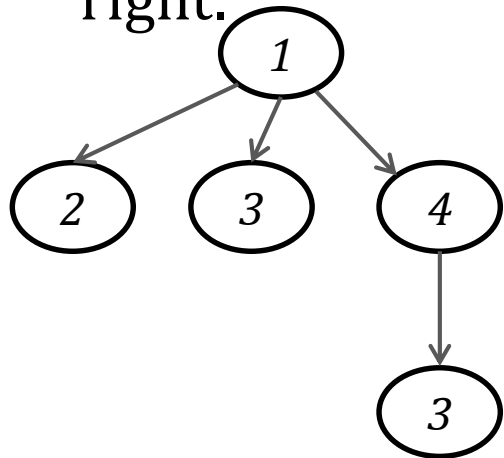
# Path, Depth, Level, and Height



- ◆ There is exactly one path (one sequence of edges) connecting each node to the root.
- ◆ depth of a node = # of edges on the path from it to the root.
- ◆ Nodes with the same depth form a level of the tree
- ◆ The height of a tree is the maximum depth of its nodes: the tree above has a height of 2.

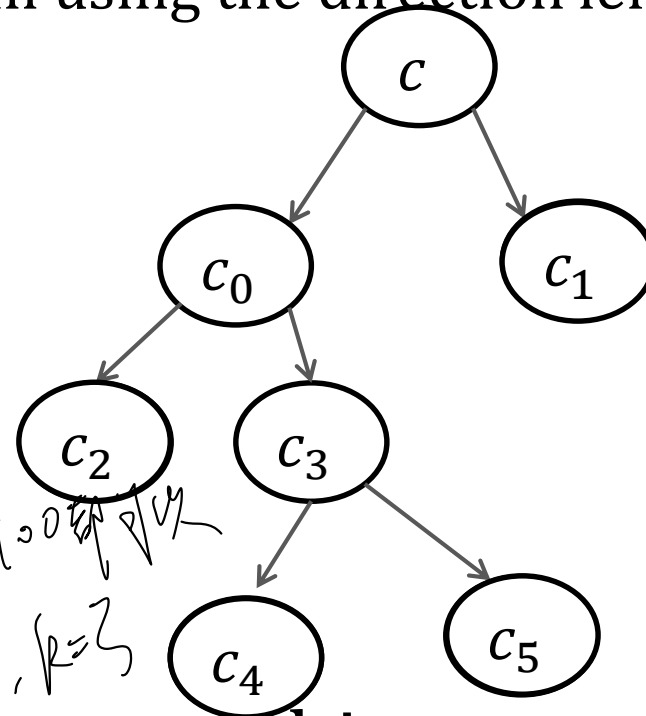
# ~~k-ary~~ → k-ary and Binary Tree

- ◆ A k-ary tree is a rooted tree where every internal node has at most k child nodes.
- ◆ A 2-ary tree is called a binary tree
- ◆ In a binary tree, nodes have at most two children.
  - ◆ Distinguish between them using the direction left and right.



3-ary

$k=2, 4, 5, \dots$  都可以  
都为正确的,  $k=2$



binary

# Binary Tree Definition

◆ Binary tree recursive definition:

◆ A binary tree is either:

◆ 1) empty or

◆ 2) a node (the root of the tree) that has

◆ one or more pieces of data (the key, and possibly others)

◆ a left subtree, which is itself a binary tree

◆ a right subtree, which is itself a binary tree

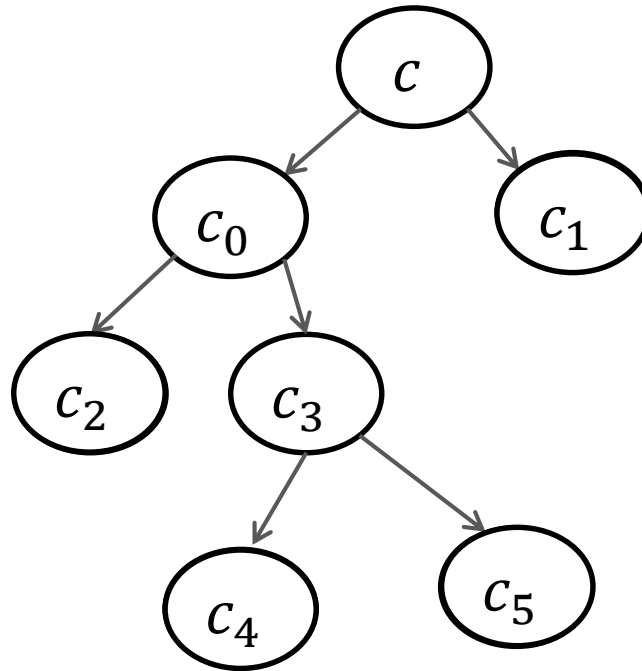
◆ A binary tree implies an ordering among the nodes at the same level.

第  $h$  层: 第  $h$  层有  $2^h$  个节点

①  
② At level  $h$ , the left node are as left as possible

# Binary Tree: Full Level

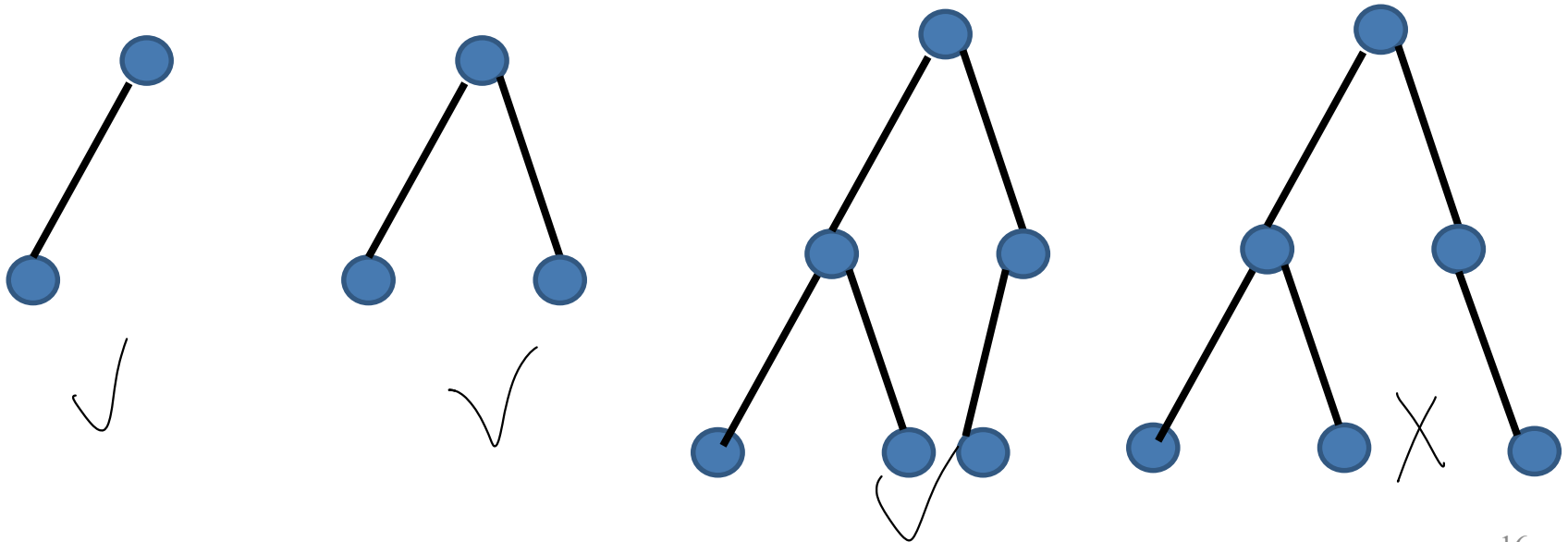
- Consider a binary tree with height  $h$ , its level  $l$  ( $0 \leq l \leq h$ ) is full if it contains  $2^l$  nodes.



- Levels 0 and 1 are full, but levels 2 and 3 are not.

# Binary Tree: Complete Binary Tree

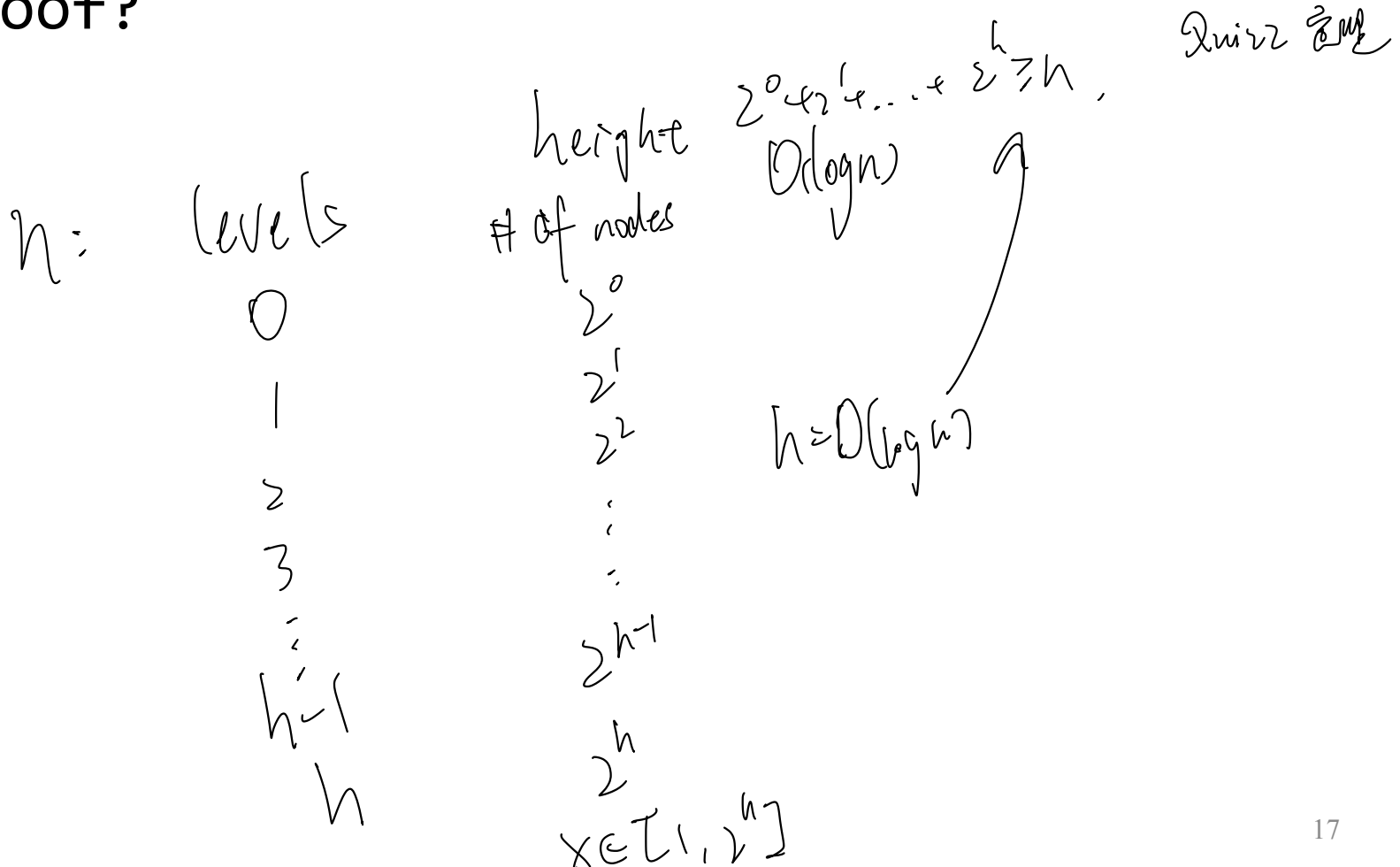
- ◆ A binary tree of height  $h$  is complete if:
  - ◆ Level 0, 1, ...,  $h-1$  are all full
  - ◆ At level  $h$ , the leaf nodes are “as far left as possible”
    - ◆ This means that if you want to add a leaf node  $v$  at level  $h$ ,  $v$  would need to be on the right of all the existing leaf nodes.





# Tree Property III

- ◆ A complete binary tree with  $n \geq 2$  nodes has height  $O(\log n)$
- ◆ Proof?



# Binary Tree Implementation

◆ Struct treeNode

{

int key;

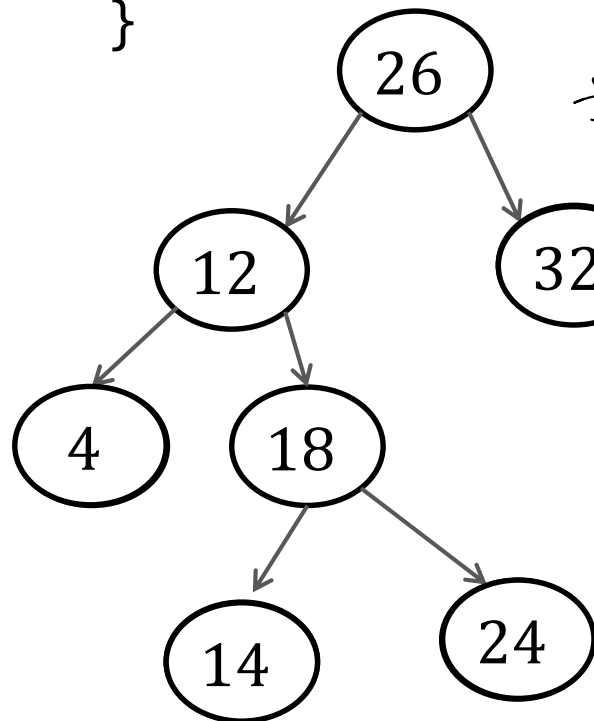
treeNode left;

treeNode right;

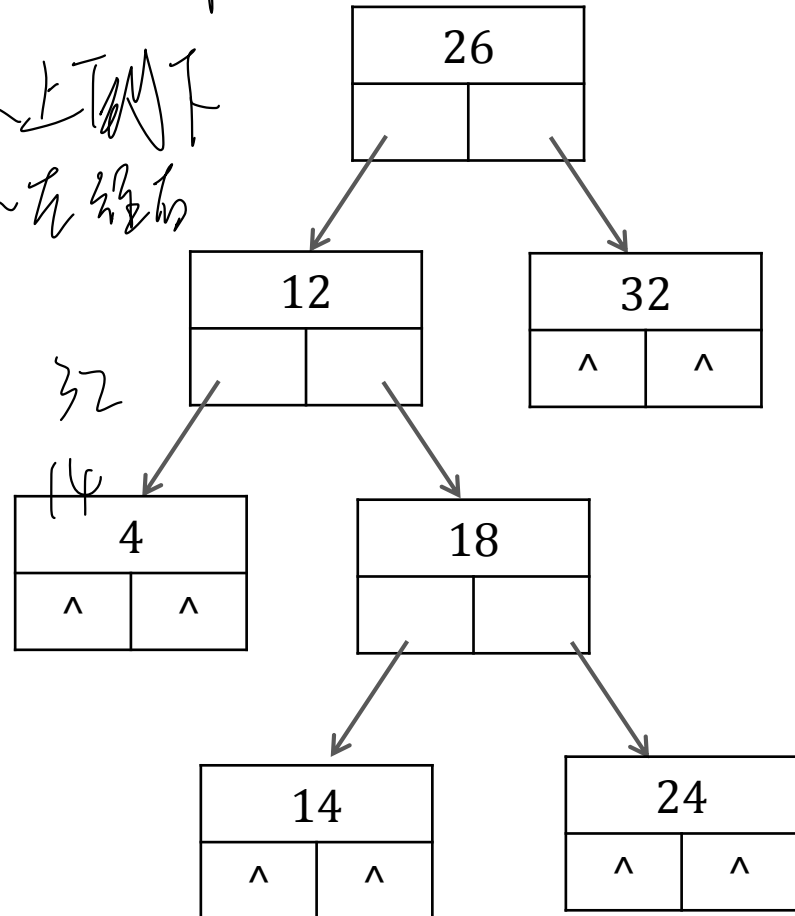
}

root - left - right

从上到下  
从左到右



⇒ 26 12 32  
4 18 14 24

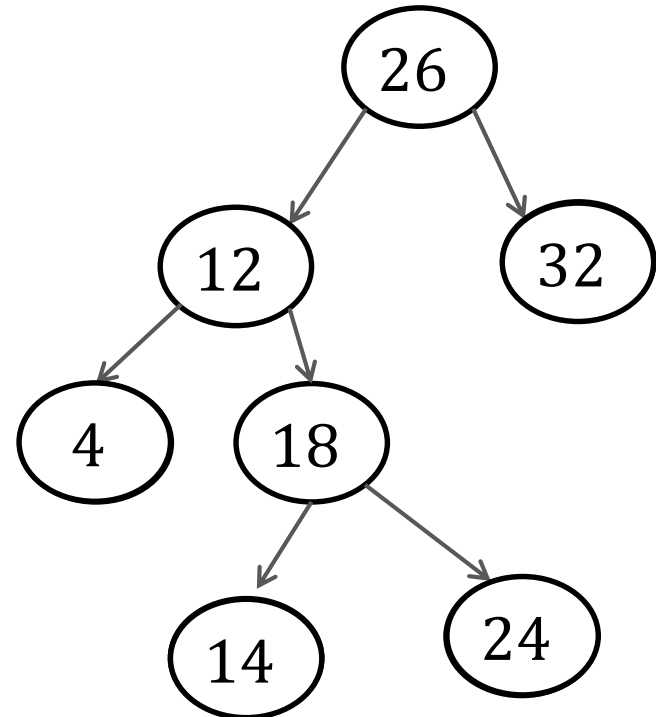


# Traversing a Binary Tree

- ◆ Traversing a tree involves visiting all of the nodes in the tree.
  - ◆ Visiting a node = processing its data in some way
    - ◆ example: print the key
- ◆ We will look at four types of traversals. Each of them visits the nodes in a different order.
- ◆ To understand traversals, it helps to remember the recursive definition of a binary tree, in which every node is the root of a subtree.

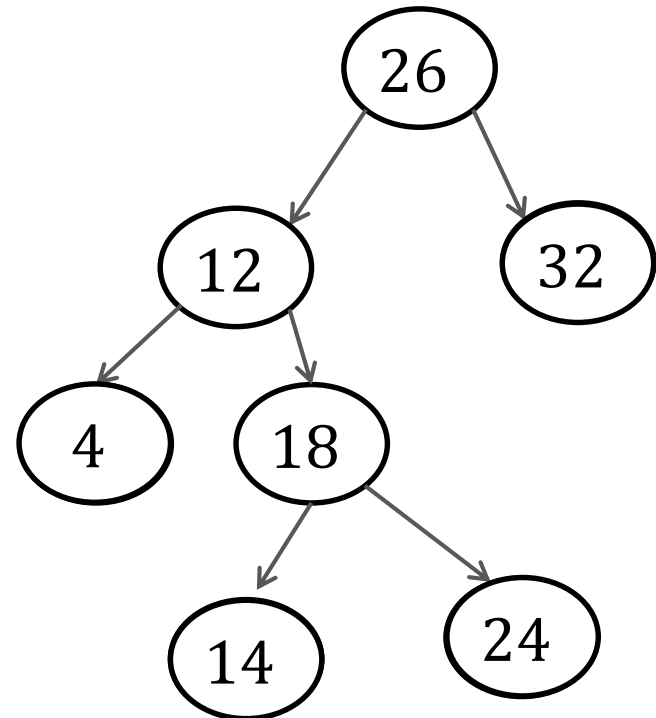
# Preorder Traversal

- ◆ Preorder traversal of the tree whose root is N
  - ◆ visit the root N
  - ◆ Recursively perform a preorder traversal of N's left subtree
  - ◆ Recursively perform a preorder traversal of N's right subtree
- ◆ Preorder traversal
  - ◆ 26, 12, 4, 18, 14, 24, 32



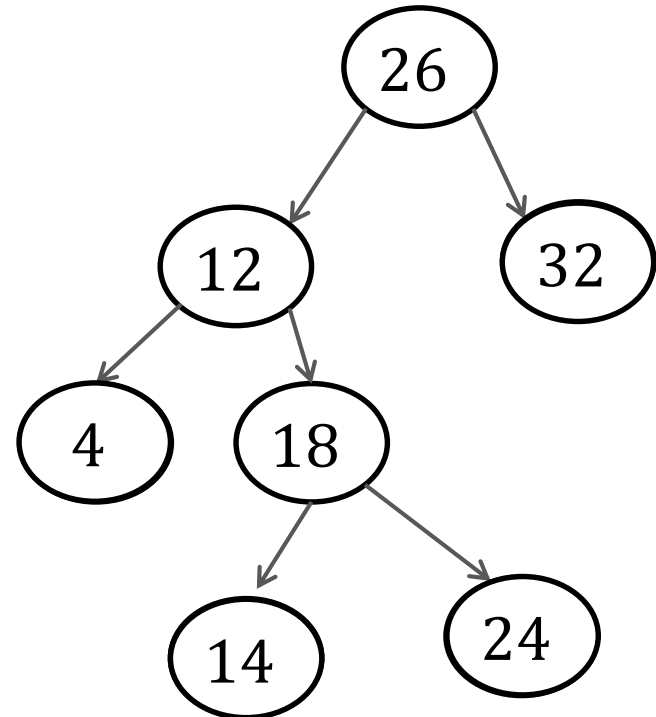
# Postorder Traversal

- ◆ Postorder traversal of the tree whose root is N
  - ◆ Recursively perform a postorder traversal of N's left subtree
  - ◆ Recursively perform a postorder traversal of N's right subtree
  - ◆ visit the root N
- ◆ Postorder traversal
  - ◆ 4, 14, 24, 18, 12, 32, 26



# Inorder Traversal

- ◆ Inorder traversal of the tree whose root is N
  - ◆ Recursively perform a inorder traversal of N's left subtree
  - ◆ Visit the root N
  - ◆ Recursively perform a inorder traversal of N's right subtree
- ◆ Inorder traversal
  - ◆ 4, 12, 14, 18, 24, 26, 32



# Preorder Traversal

- ◆ Implementation:

- ◆ Recursive Implementation? So easy?

- ◆ preorderprint(treeNode root):

- 1. print(root)

- 2. if(root->left!=null)

- 3.     preorderprint(root->left)

- 4. if(root->right!=null)

- 5.     preorderprint(root->right)




# Preorder Traversal

- ◆ Implementation:

- ◆ Iterative Implementation?

- ◆ preorderiterative(treeNode root):

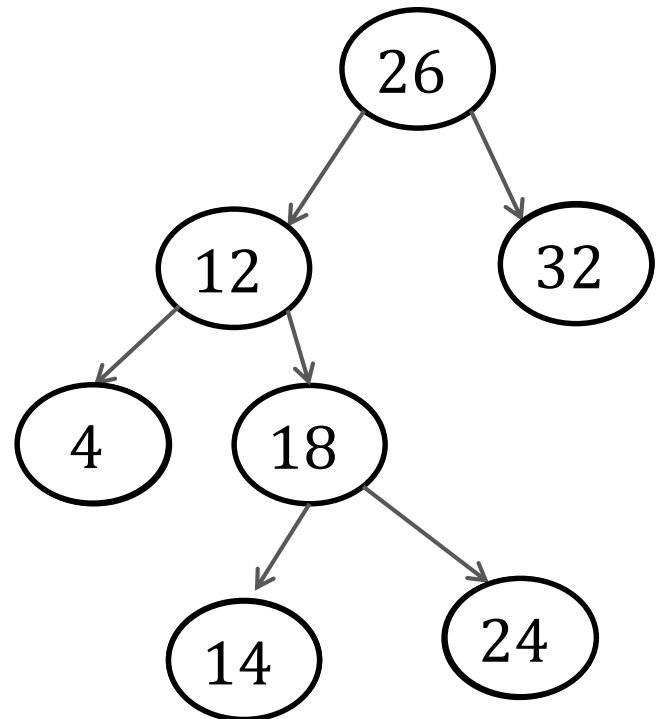


```
1. treeNode stack s
2. s.push(root)
3. while(s!=empty)
4.     treeNode node= s.top()
5.     print(node)
6.     s.pop()
7.     if(node->right!=null)
8.         s.push(node->right)
9.     if(node->left!=null)
10.        s.push(node->left)
```



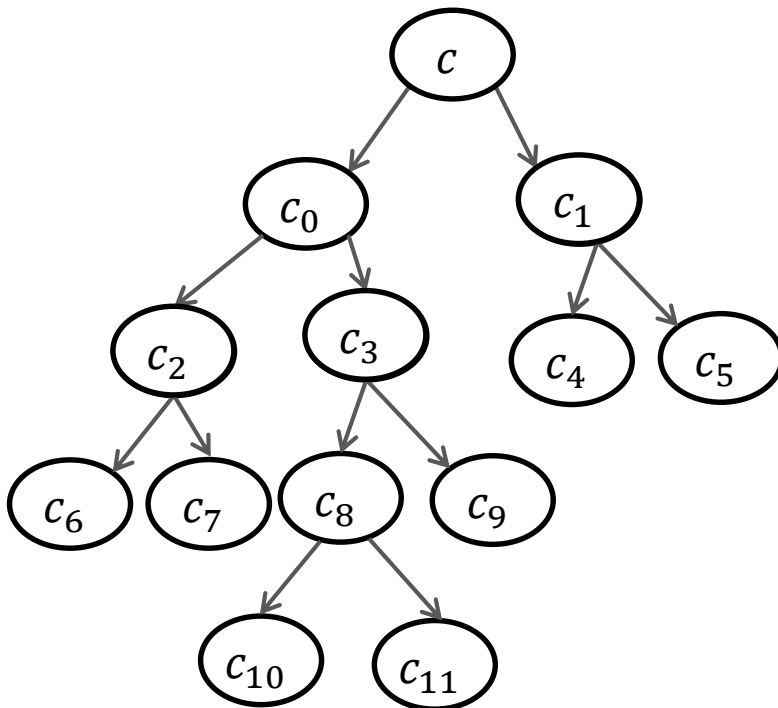
# Level Traversal

- ◆ Visit the nodes one level at a time, from top to bottom, and left to right.
- ◆ Level-order of the tree:
  - ◆ 26, 12, 32, 4, 18, 14, 24
- ◆ How to implement?



# Summary

- ◆ Preorder: root, left subtree, right subtree
- ◆ Postoder: left subtree, right subtree, root
- ◆ Inorder: left subtree, root, right subtree
- ◆ Level-order: top to bottom, left to right

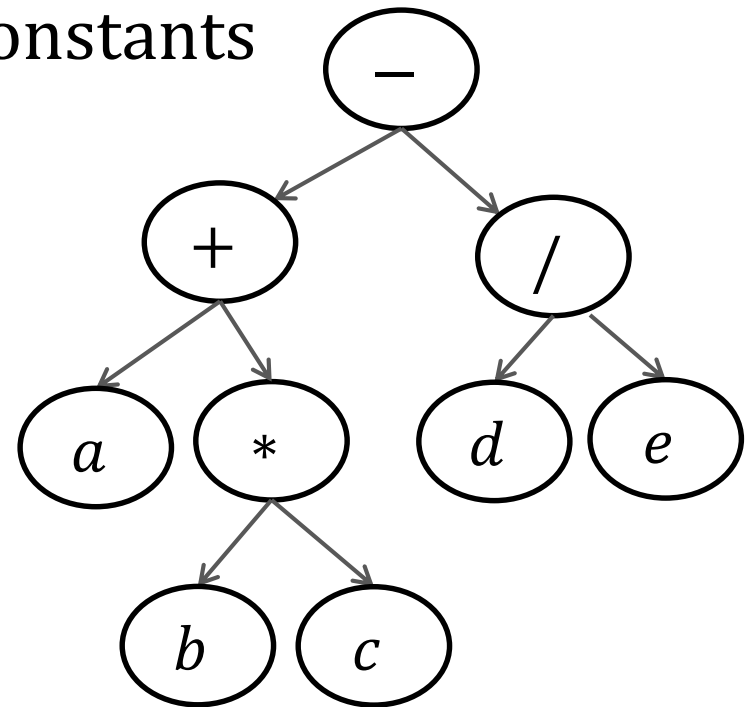


# Our Roadmap

- ◆ Tree
  - ◆ Basic Concepts
  - ◆ Properties of Tree (focus on binary tree)
  - ◆ Binary Tree Traversal
- ◆ Binary Tree Applications
  - ◆ Algebraic expression
  - ◆ Huffman encoding

# Algebraic Expression

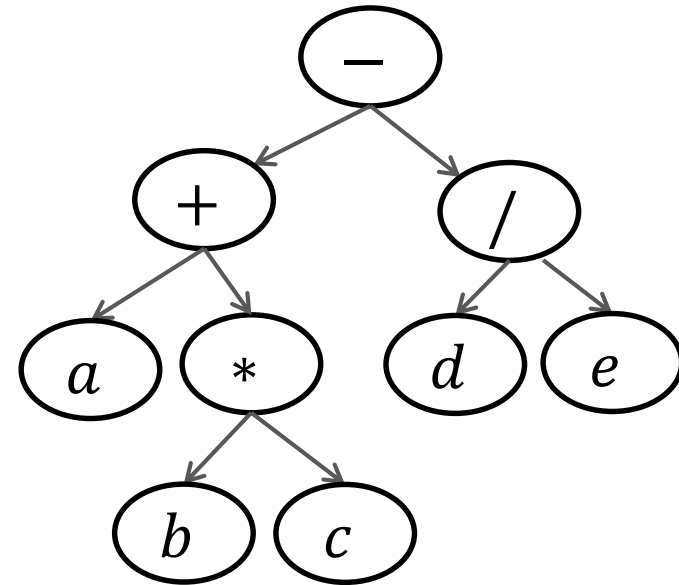
- ◆ We only consider fully parenthesized expressions with binary operators:  $+$ ,  $-$ ,  $*$ ,  $/$
- ◆ Example expression:  $((a+(b*c))-(d/e))$
- ◆ Leaf nodes are variables or constants
- ◆ Internal nodes are operators
- ◆ Why is it a binary tree?
  - ◆ Binary operators



# Algebraic-Expression Tree Traversal

- ◆ Inorder gives conventional algebraic notation

- ◆ print “(” before visit left tree
- ◆ print “)” after visit right tree
- ◆ for tree at right:  $((a+(b*c))-(d/e))$



- ◆ Preorder gives functional notations
  - ◆ Print “(” and “)” as for inorder, and commas after visit left subtree
  - ◆ for tree above: `subtr(add(a,mult(b,c)),divide(d,e))`
- ◆ Postorder gives the order in which the computation must be carried out on a stack.
  - ◆ for tree above: push a, push b, push c, multiply, add, ...

# Character Encoding

- ◆ A character encoding maps each character to a number
- ◆ Computers usually use fixed-length character encodings
  - ◆ ASCII uses 8 bits per character
    - ◆ “bat” in computer: 01100010 01100001 01110100
  - ◆ Unicode uses 16 bits per character
    - ◆ ASCII codes are a subset
  - ◆ Fixed-length encoding are simple, because:
    - ◆ All character encodings have the same length
    - ◆ A given character always has the same encoding
  - ◆ Problem: fixed length encoding waste space
    - ◆ Solution: a variable-length encoding

# Variable-Length Character Encodings

- ◆ Variable-length encoding
  - ◆ Use encodings of different lengths for different characters
  - ◆ Assign shorter encodings to frequently occurring characters
- ◆ Example: “test” would be encoded as:

e	01	s	111
o	100	t	00

  - ◆ 00 01 111 00 → 000111100
- ◆ Challenge: when decoding an encoded document, how do we determine the boundaries between characters?
  - ◆ For the above example, how do we know whether the next character is 2 bits or 3 bits
- ◆ Requirement: no character's encoding can be the prefix of another character's encoding (e.g., couldn't have 00 and 001)

# Huffman Encoding

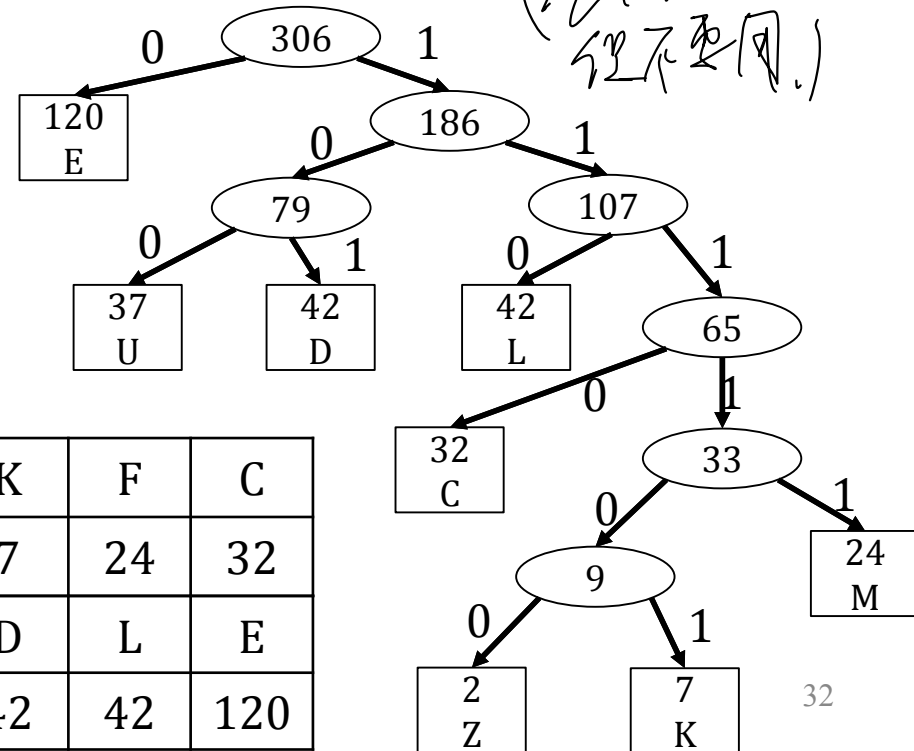
→ 最优前缀码  
树.

- ◆ Huffman encoding is a type of variable-length encoding that is based on the actual character frequencies in a given document.
- ◆ Huffman encoding uses a binary tree:
  - ◆ to determine the encoding of each character
  - ◆ to decode an encoded file

左 0 右 1  
(左 1 右 0 也可以  
经不用.)

## ◆ Example:

- ◆ Leaf nodes are characters
- ◆ 101 = 'D'
- ◆ "000110" = "EEEL"



Z	K	F	C
2	7	24	32
U	D	L	E
37	42	42	120



# Building a Huffman tree

- ◆ 1) Begin by reading through the text to determine the frequencies
- ◆ 2) Create a list of nodes that contain (character, frequency) pairs for each character that appears in text
- ◆ 3) Remove and “merge” the nodes with the two lowest frequencies, forming a new node that is their parent
- ◆ 4) Add the parent to the list of nodes
- ◆ 5) Repeat steps 3) and 4) until there is only a single node in the list, which will be the root of the Huffman tree.
- ◆ Example: build the Huffman tree for the following (character, frequency) pairs:

Z	K	F	C	U	D	L	E
2	7	10	12	27	30	43	65

# Correctness of Huffman tree

- ◆ Given a Huffman tree, it includes at least 2 nodes, assume node  $u$  and node  $v$  have the top-2 lowest frequencies, then
  - ◆ 1) node  $u$  and  $v$  have the same parent node
  - ◆ 2)  $\text{depth}(u)$  and  $\text{depth}(v) \geq \text{depth}(x)$ , where node  $x$  is any leaf node in the Huffman tree.
    - ◆ proof?
- ◆ Huffman encoding is the optimum prefix code, i.e., the space cost is minimized.
  - ◆ Proof.
  - ◆ <http://home.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL17.pdf>

Thank You!

# Tree Property I

- ◆ A tree with  $n$  nodes with  $n-1$  edges
- ◆ Proof?
  - ◆ For each non-root node  $v$ , it has one and only one edge point to itself.
  - ◆ A tree with  $n$  nodes, thus the number of non-root nodes is  $n-1$ .
  - ◆ Thus, this tree has  $n-1$  edges.

# Tree Property II

- ◆ Let  $T$  be a tree where every internal node has at least 2 child nodes. If  $m$  is the number of leaf nodes, then the number of internal nodes is at most  $m-1$ .
  - ◆ Suppose internal node  $v$  has  $x_v$  child nodes
  - ◆ The average child nodes of each internal node is  $x$
  - ◆ It has  $m$  leaf nodes, thus it has  $m/x$  parent nodes at most, i.e., they are parent of leaf nodes.
  - ◆ For  $m/x$  internal nodes, it has at most  $m/x^2$  parents.
  - ◆ For  $m/x^2$  internal nodes, it has at most  $m/x^3$  parents.
  - ◆ ...
  - ◆ The total number of internal nodes is
$$m/x + m/x^2 + \dots + 1$$
  - ◆ It is at most  $m-1$ .

# Tree Property III

- ◆ A complete binary tree with  $n \geq 2$  nodes has height  $O(\log n)$
- ◆ Proof?
  - ◆ Suppose the height is  $h$ .
  - ◆ The number of nodes at each level:
    - ◆ Level 0:  $2^0 = 1$ , Level 1:  $2^1 = 2$
    - ◆ Level 2:  $2^2 = 4$ , Level 3:  $2^3 = 8$
    - ◆ ...
    - ◆ Level  $h-1$ :  $2^{h-1}$ , Level  $h$ :  $x$  ( $x \geq 1$ )
  - ◆ Thus,  $2^0 + 2^1 + \dots + 2^{h-1} + x = n$
  - ◆  $\Rightarrow (1-2^h)/(1-2) = n-x \Rightarrow 2^h < n$
  - ◆ Thus,  $h = O(\log n)$