DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

Lecture 5

Binary Trees

Outlines of Class (Lecture 5)

- Introduction
- Binary Tree
- Binary Search Tree
- Heaps and Priority queues
- Huffman Coding Trees

Introduction

- The list representations have a fundamental limitation:
  - ✓ Either search or insert can be made efficient, but not both at the same time.
- Tree structures permit both efficient access and update to large collections of data.
- A tree is a collection of entities called nodes, connected by edges.
- Each node contains a value or data
- Each node may or may not have a child node
- Consists of nodes with a parent-child relation.
- Trees are hierarchical structure.
- Applications:
  - ✓ Organization charts
  - ✓ File systems
  - ✓ Programming environments

Binary Tree

- A binary tree is made up of a finite set of elements called nodes.
- This set either is empty or consists of a node called the root together with two binary trees, called the left and right subtrees
- A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.
- Applications of Binary Tree
  - ✓ prioritizing jobs,
  - ✓ describing mathematical expressions and the syntactic elements of computer programs, or
  - ✓ organizing the information needed to drive data compression algorithms.

Binary Tree (cont.)

- Three examples of binary trees used in specific applications:
  - ✓ the Binary Search Tree (BST) for implementing dictionaries,
  - ✓ heaps for implementing priority queues, and
  - ✓ Huffman coding trees for text compression.
- Binary trees are widely used and easy to implement.

Binary Tree (cont.)

- A (root)
- B and C (A's children/ siblings)
- B and D (subtree)
- B has two children (left child is empty and right child is D)
- Ancestors of G (A, C and E)
- D, E and F (level 2)
- ACEG (a path of length 3)
- D, G, H and I (leaves)
- A, B, C, E and F (internal nodes)
- The depth of I (3)
- The height of tree (4)

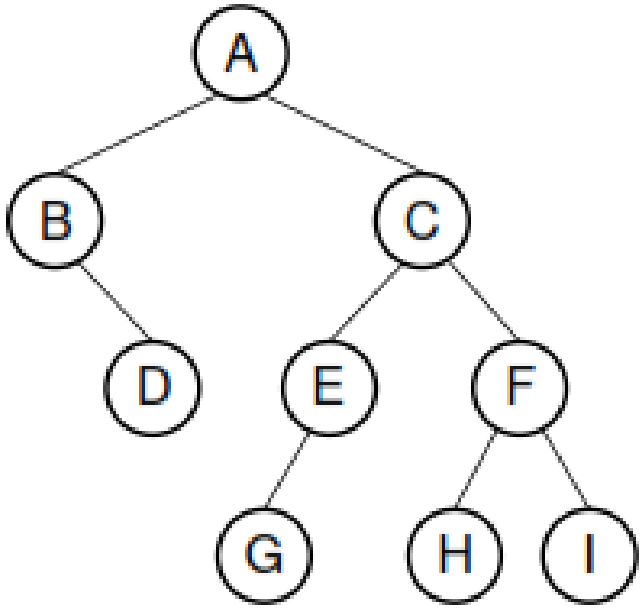


Fig: A Binary Tree

The Structure of Binary Trees

- A binary tree whose root has a non-empty left child
- A binary tree whose root has a non-empty right child
- The binary tree of (a) with the missing right child made explicit
- The binary tree of (b) with the missing left child made explicit

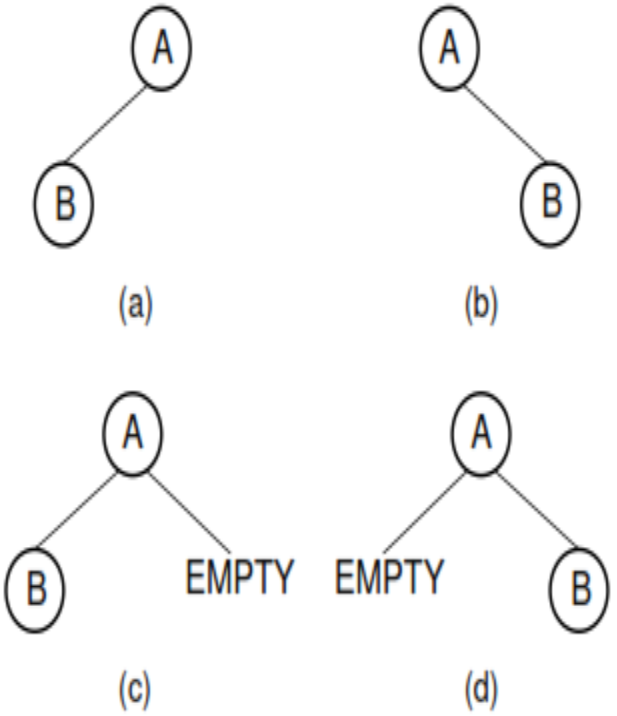


Fig: Two different Binary Trees

Full Binary Tree Vs. Complete Binary Tree

- Full Binary Tree
  - ✓ An internal node with exactly two non-empty children (or) a leaf
- Complete Binary Tree
  - ✓ By starting at the root and filling the tree by levels from left to right
  - ✓ A binary tree T with n levels is complete if all levels (except the last level) are completely full, and the last level has its nodes filled in from the left side.

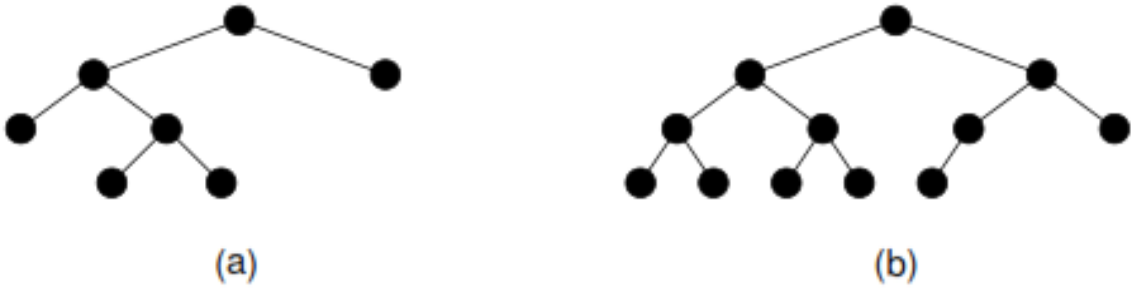


Fig: Examples of full and complete binary trees
(a) Full binary tree, (b) Complete binary tree

Full Binary Tree Vs. Complete Binary Tree (Contd.)

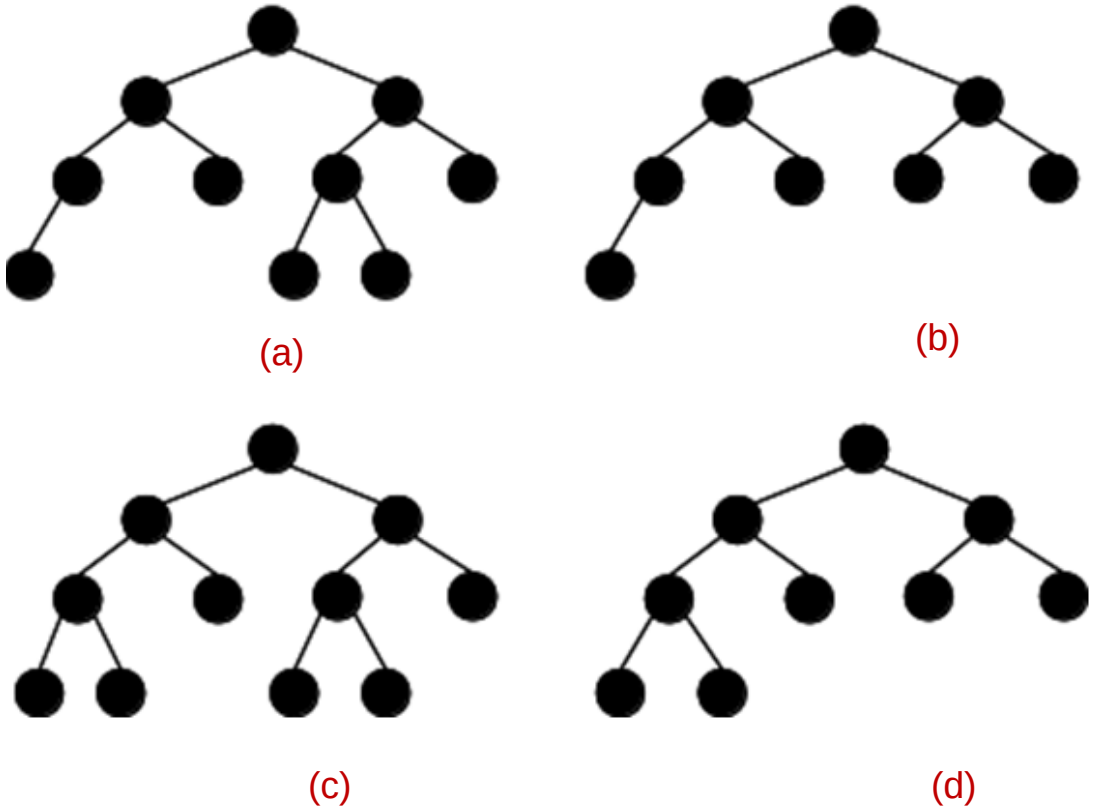


Fig: (a) Neither complete nor full, (b) Complete but not full, (c) Full but not complete, (d) Complete and full

Theorem 5.1 Full Binary Tree Theorem: *The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.*

Proof:

- **Base Cases**: The non-empty tree with zero internal nodes has one leaf node. A full binary tree with one internal node has two leaf nodes. Thus, the base cases for $n = 0$ and $n = 1$ conform to the theorem.
- **Induction Hypothesis**: Assume that any full binary tree $\mathbf{T}$ containing $n - 1$ internal nodes has n leaves.
- **Induction Step**: Given tree $\mathbf{T}$ with n internal nodes, select an internal node I whose children are both leaf nodes. Remove both of I's children, making I a leaf node. Call the new tree $\mathbf{T}'$. $\mathbf{T}'$ has $n - 1$ internal nodes. From the induction hypothesis, $\mathbf{T}'$ has n leaves. Now, restore I's two children. We once again have tree $\mathbf{T}$ with n internal nodes. How many leaves does $\mathbf{T}$ have? Because $\mathbf{T}'$ has n leaves, adding the two children yields $n+2$. However, node I counted as one of the leaves in $\mathbf{T}'$ and has now become an internal node. Thus, tree $\mathbf{T}$ has $n + 1$ leaf nodes and n internal nodes.

A Binary Tree Node ADT

```
// Binary tree node abstract class
template <typename E> class BinNode {
public:
  virtual ~BinNode() {} // Base destructor

  // Return the node's value
  virtual E& element() = 0;

  // Set the node's value
  virtual void setElement(const E&) = 0;

  // Return the node's left child
  virtual BinNode* left() const = 0;

  // Set the node's left child
  virtual void setLeft(BinNode*) = 0;

  // Return the node's right child
  virtual BinNode* right() const = 0;

  // Set the node's right child
  virtual void setRight(BinNode*) = 0;

  // Return true if the node is a leaf, false otherwise
  virtual bool isLeaf() = 0;
};
```

Binary Tree Traversal

- Traversal
  - Any process for visiting all of the nodes in some order
- Since a binary tree has three "parts," there are six possible ways to traverse the binary tree:

- root, left, right
- left, root, right
- left, right, root

- root, right, left
- right, root, left
- right, left, root

Binary Tree Traversal Method

- Depth First Search
  - ✓ Preorder Traversal
  - ✓ Postorder Traversal
  - ✓ Inorder Traversal
- Breath First Search
  - ✓ Level Order Traversal

Preorder Traversal

Any given nodes are visited before its children visited.

Procedure:

- The node's data (root) is processed.
- The node's left subtree is traversed.
- The node's right subtree is traversed.

Preorder Traversal (Cont.)

- Preorder Traversal (D, L, R)

  A B D C E G F H I

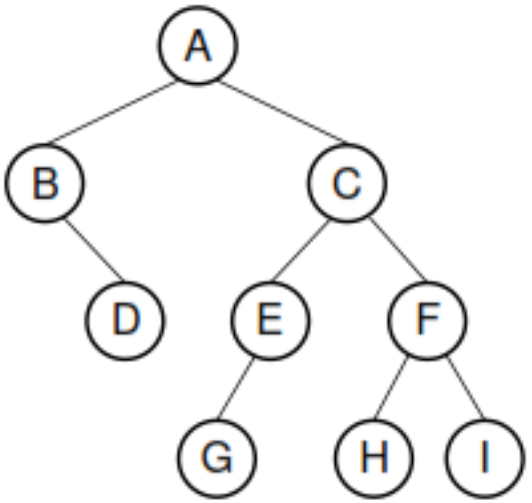


Fig: A Binary Tree

Postorder Traversal

Each node is visited only after if its children (and their subtrees) visited.

Procedure:

- The node's left subtree is traversed.
- The node's right subtree is traversed.
- The node's data (root) is processed.

Postorder Traversal (Cont.)

- Postorder Traversal (L, R, D)

  D B G E H I F C A

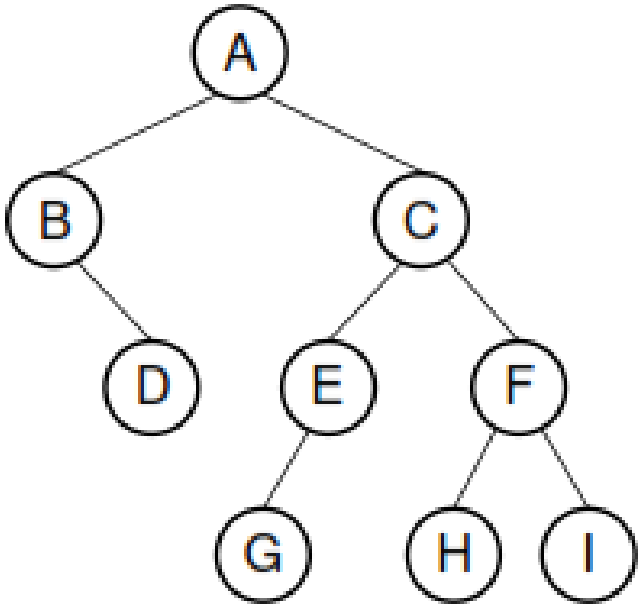


Fig: A Binary Tree

Inorder Traversal

First visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree) visited.

Procedure:

- The node's left subtree is traversed.
- The node's data (root) is processed.
- The node's right subtree is traversed.

Inorder Traversal (Cont.)

- Inorder Traversal (L,D,R)

  B D A G E C H F I

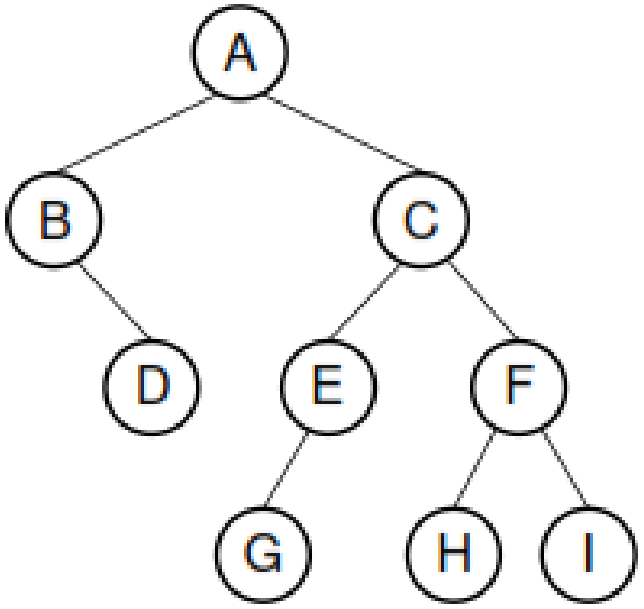


Fig: A Binary Tree

Level Order Traversal

- Visit every node on a level (from left to right) before going to a lower level.

- Level Order Traversal

  A B C D E F G H I

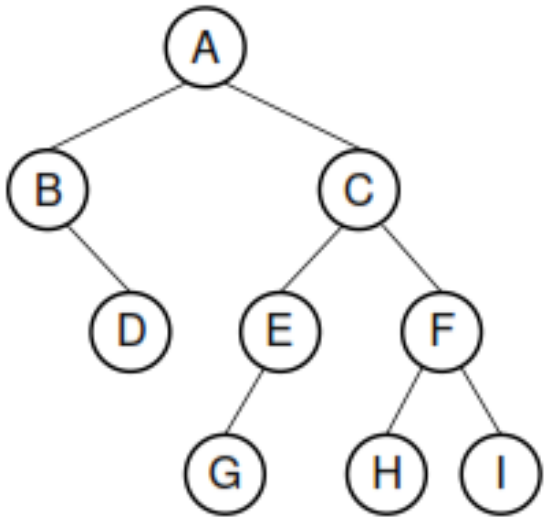


Fig: A Binary Tree

Binary Tree Node Implementation

- Pointer-based Implementation
- Array-based Implementation

Pointer-Based Node Implementation

- Node implementation
  - ✓ A value field
  - ✓ Pointers to the two children (left child and right child)

Pointer-Based Node Implementation (Cont.)

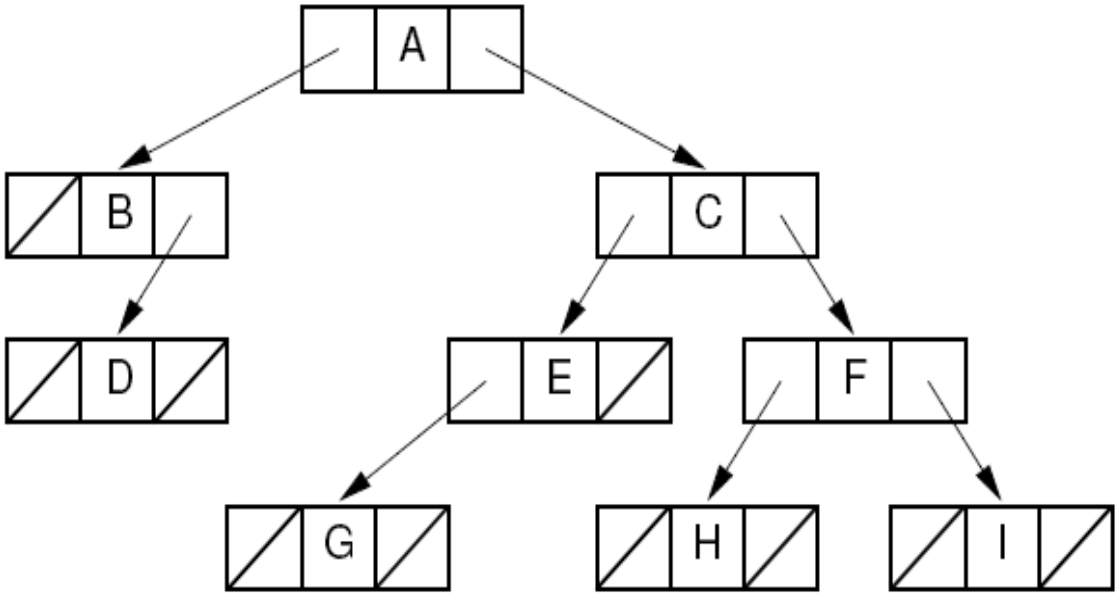


Fig : Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value

An Expression Tree using Pointer

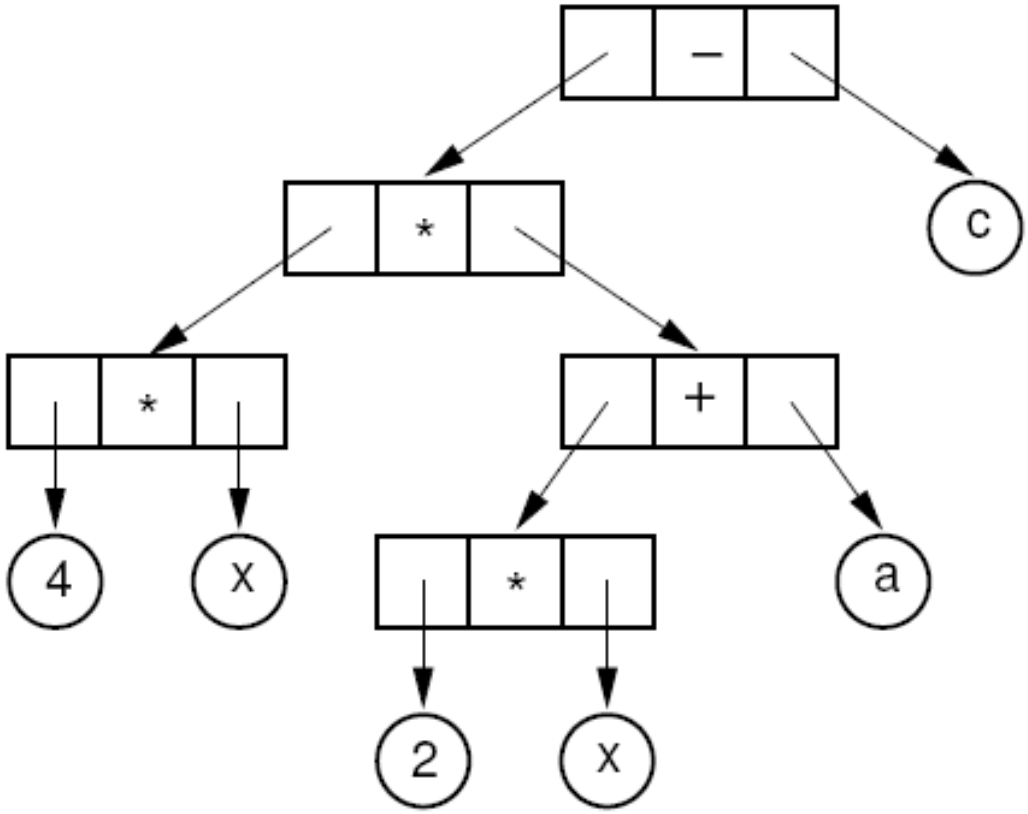
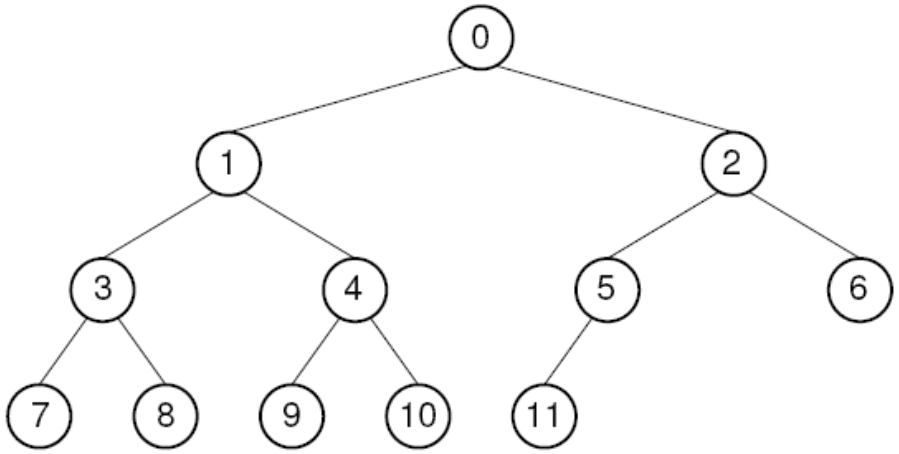


Fig: An expression tree for 4x(2x+a)-c

Array Implementation for Complete Binary Tree

- Assign numbers to the node positions in the complete binary tree, level by level, from left to right
- An array can store tree's data values efficiently, placing each data value in the array position corresponding to that node's position within the tree
- The formulae for calculating the array indices of relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node is r, the range 0 to n-1.

  - Parent$(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
  - Left child$(r) = 2r + 1$ if $2r + 1 < n$.
  - Right child$(r) = 2r + 2$ if $2r + 2 < n$.
  - Left sibling$(r) = r - 1$ if r is even.
  - Right sibling$(r) = r + 1$ if r is odd and $r + 1 < n$.



(a)

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

(b)

Fig: A complete binary tree and its array implementation (a) complete binary tree with twelve nodes (b) the positions for the relatives of each node

“In computer science, a binary tree is a tree data structure in which each node has at the most two children, which are referred to as the left child and the right child.”—Wikipedia

Binary Search Tree (BST)

- BST is a binary tree that conforms the condition such as Binary Search Tree Property.
- BST has a better performance than any of data structures when the functions (search, insert and delete) to be performed.
- All nodes stored in the left subtree of a node whose key value is K have key values less than K.
- All nodes stored in the right subtree of a node whose key value is K have key values greater than or equal to K.
- Result:
  - ✓ Easy to find any given key
  - ✓ Insert/delete by changing links

Binary Search Tree (Cont.)

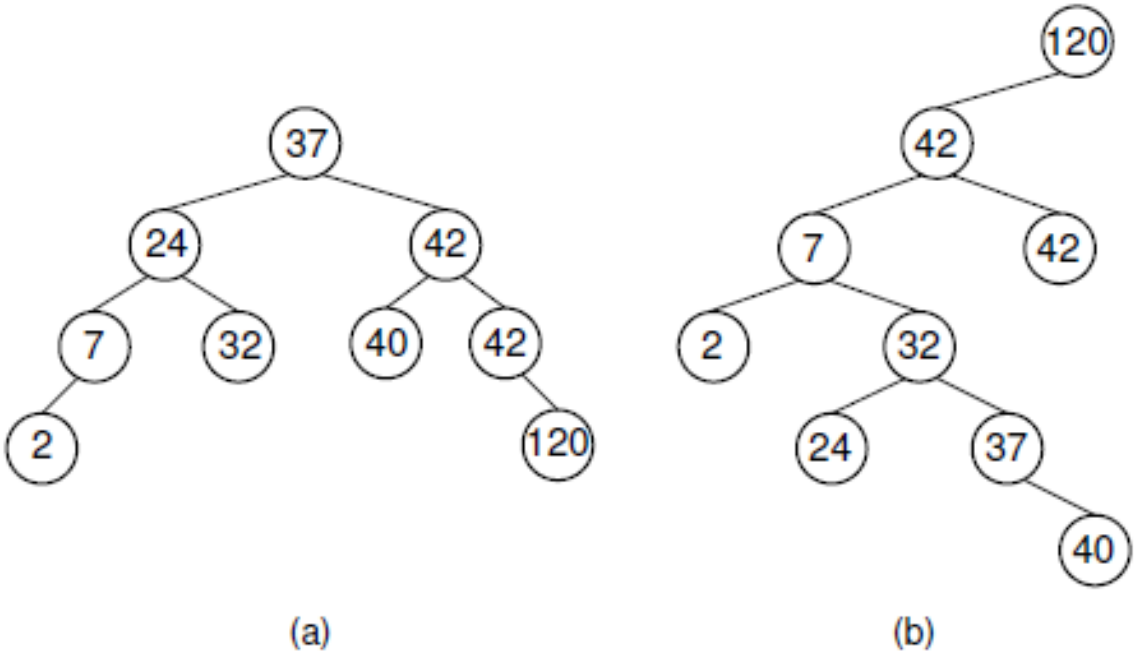
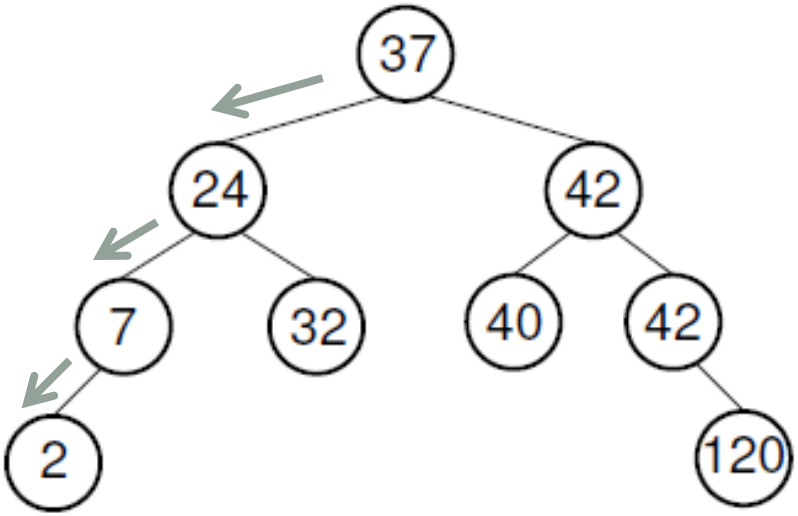


Fig: Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

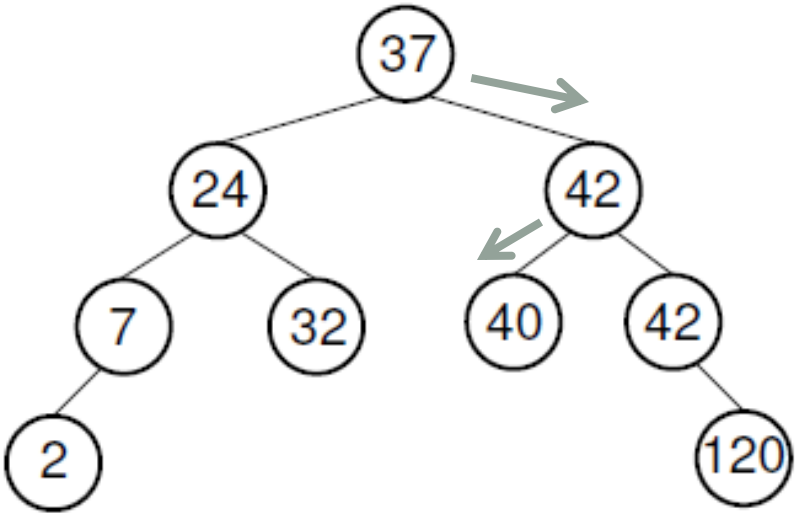
Binary Search Tree Operations

- Searching a Node (K=2)



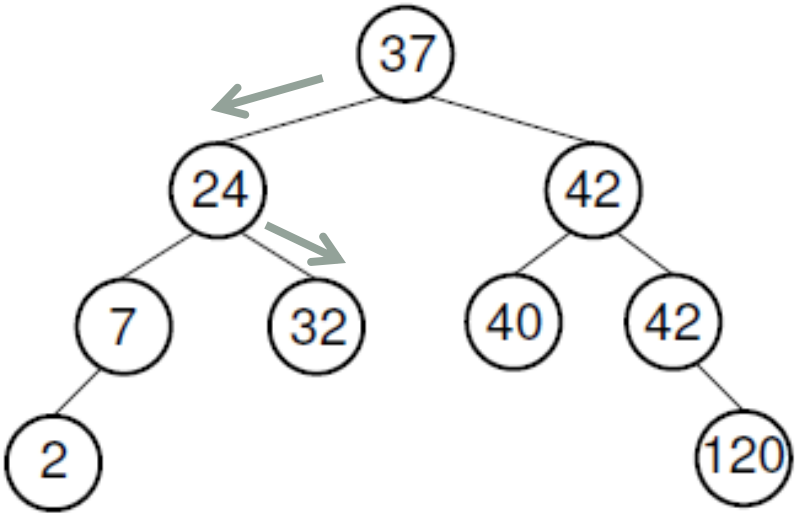
Binary Search Tree Operations (Cont.)

- Searching a Node (K=40)



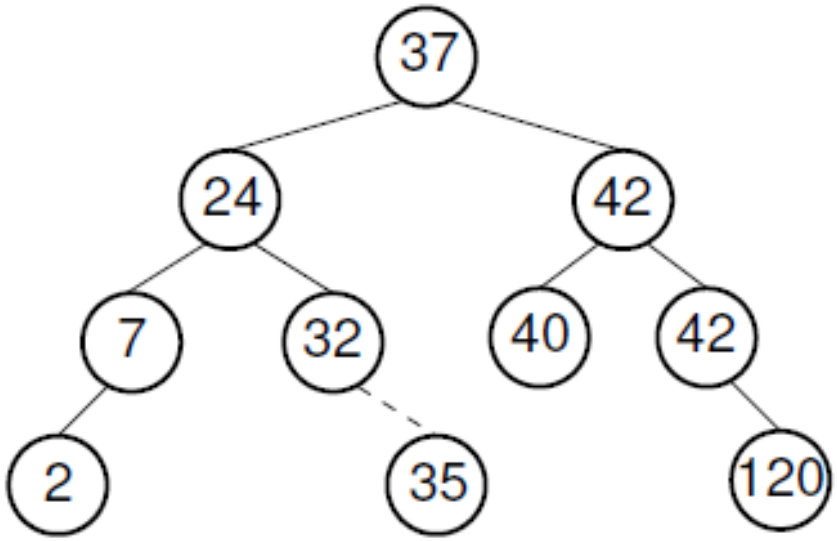
Binary Search Tree Operations (Cont.)

- Searching a Node (K=32)



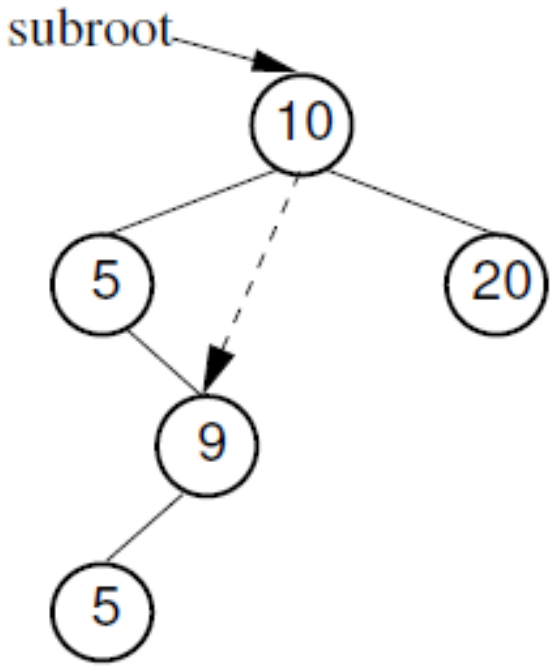
Binary Search Tree Operations (Cont.)

- Inserting a Node (K=35)



Binary Search Tree Operations (Cont.)

- Deleting a Node (K= 5)



Heaps and Priority queues

- When a collection of objects is organized by importance or priority, called as a priority queue.
- Heap data structure is guaranteed to have good performance for this special application.
  - ✓ Two properties
    - ✓ A complete binary tree, implemented using the array representation
    - ✓ The values stored in a heap are partially ordered
- Logical representation of a heap
  - ✓ Tree structure
- Physical implementation of a heap
  - ✓ Array

Variants of the Heap

- Two variants of heap
  - ✓ Max-heap
  - ✓ Min-heap
- Max-Heap
  - ✓ every node stores a value that is greater than or equal to the value of either of its children (The root stores the maximum of all values in the tree)
- Min-Heap
  - ✓ every node stores a value that is less than or equal to that of its children (The root stores the minimum of all values in the tree)

Building a Heap

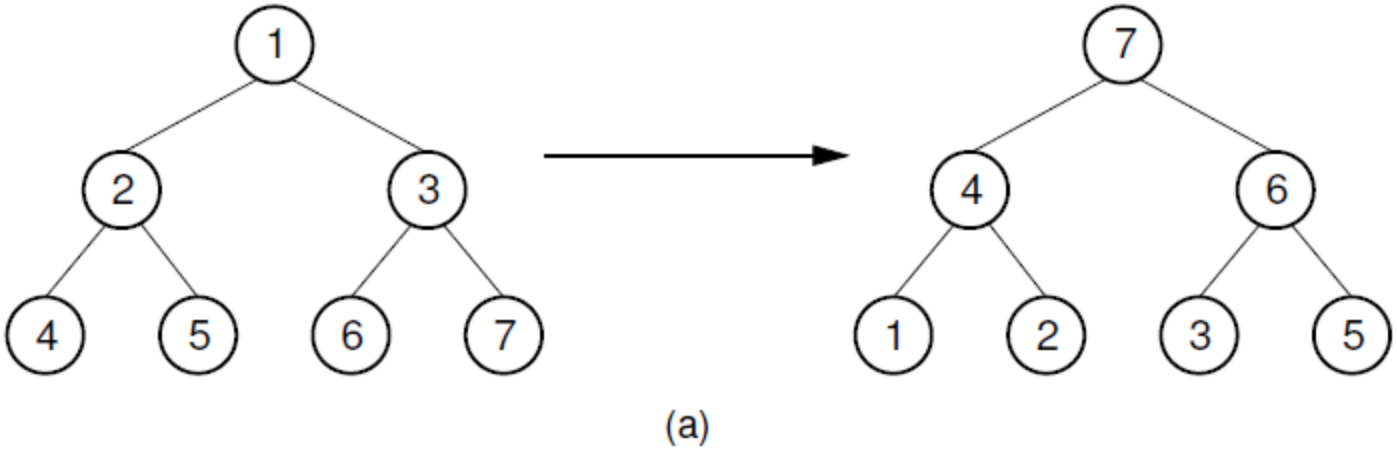


Fig: Two Series of exchanges to build a max-heap

- This heap is built by a series of nine exchanges in the order (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6) .

Building a Heap (Cont.)

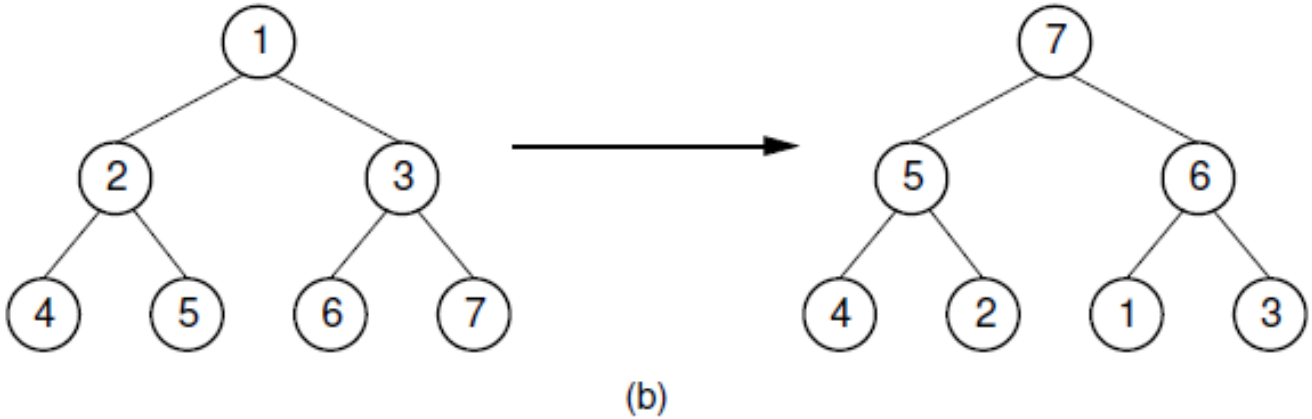


Fig: Two Series of exchanges to build a max-heap

- This heap is built by a series of four exchanges in the order (5-2), (7-3), (7-1), (6-1).

How to Build a Max Heap

- There are two possibilities:
  - ✓ R has a value greater than or equal to its two children. Construction is complete.
  - ✓ R has a value less than one or both of its children. Exchange R with the greater-valued child until it reaches its proper place.

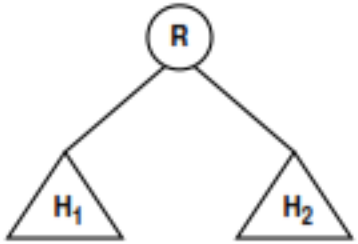


Fig: Final stage in the heap-building algorithm

Huffman Coding Trees

- Using fixed-length code can waste space.
  - ✓ ASCII: 8 bits per character

Letter	ASCII Code	Binary	Letter	ASCII Code	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
d	100	01100100	D	068	01000100
e	101	01100101	E	069	01000101
u	117	01110101	U	085	01010101
v	118	01110110	V	086	01010110
w	119	01110111	W	087	01010111
x	120	01111000	X	088	01011000
y	121	01111001	Y	089	01011001
z	122	01111010	Z	090	01011010

If some characters are used more frequently than others, assign them shorter codes?

Huffman coding
(variable-length codes)

Building Huffman Coding Trees

- Assigns codes to characters such that the length of the code depends on the relative frequency or weight of the corresponding character
- The length of the message will be less than a fixed-length code
- The Huffman code for each letter is derived from a full binary tree called the Huffman coding tree, or simply the Huffman tree.
- Each leaf of the Huffman tree corresponds to a letter, and the weight of the leaf node is defined as the weight (frequency) of its associated letter.
- The goal is to build a tree with the minimum external path weight.

Relative Frequencies

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

Fig: Relative frequencies for the 26 letters of the alphabet as they appear in a selected set of English documents.

Building Huffman Coding Trees

Letter	C	D	E	K	L	M	U	Z
Frequency	32	42	120	7	42	24	37	2

Fig: The relative frequencies for eight selected letters

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Fig: Part of the Huffman tree construction process for the eight letters of the above figure

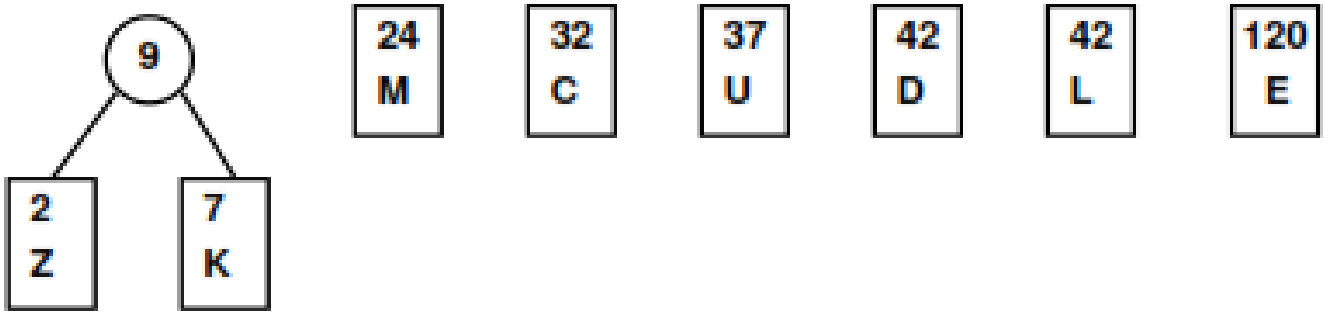
Huffman Tree Construction

- Create a list of nodes
  - Node contains letter/frequency pairs
  - Nodes are in increasing order of frequency
  - At each step, combine two smallest nodes into a binary tree and reorder as needed

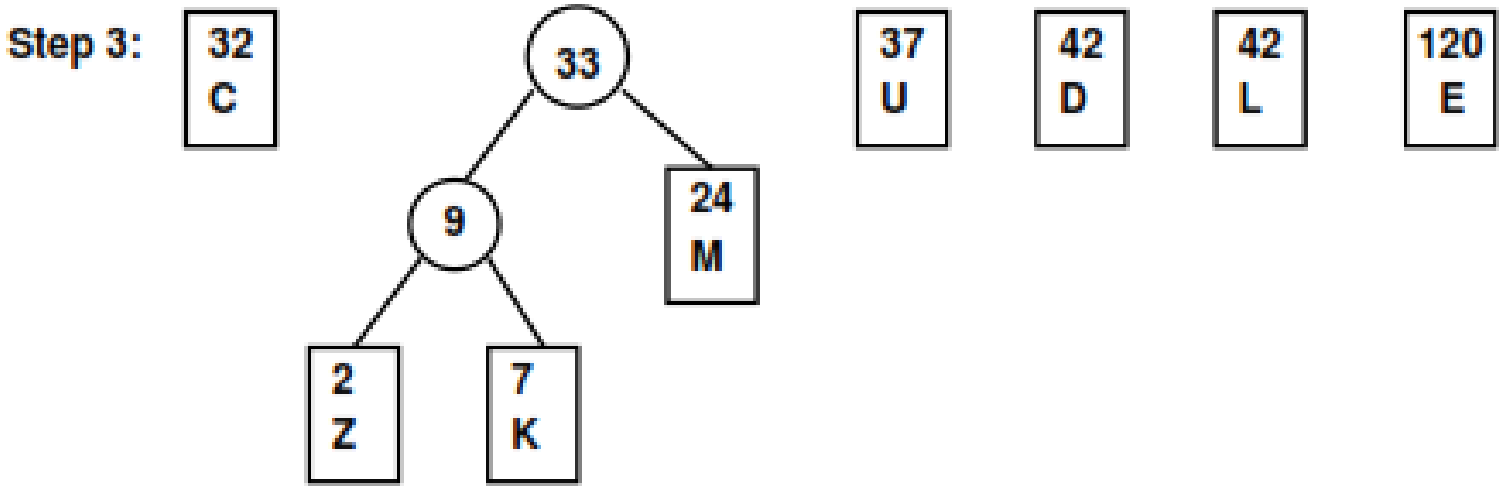
Huffman Tree Construction (Cont.)

Step 1:

Step 2:

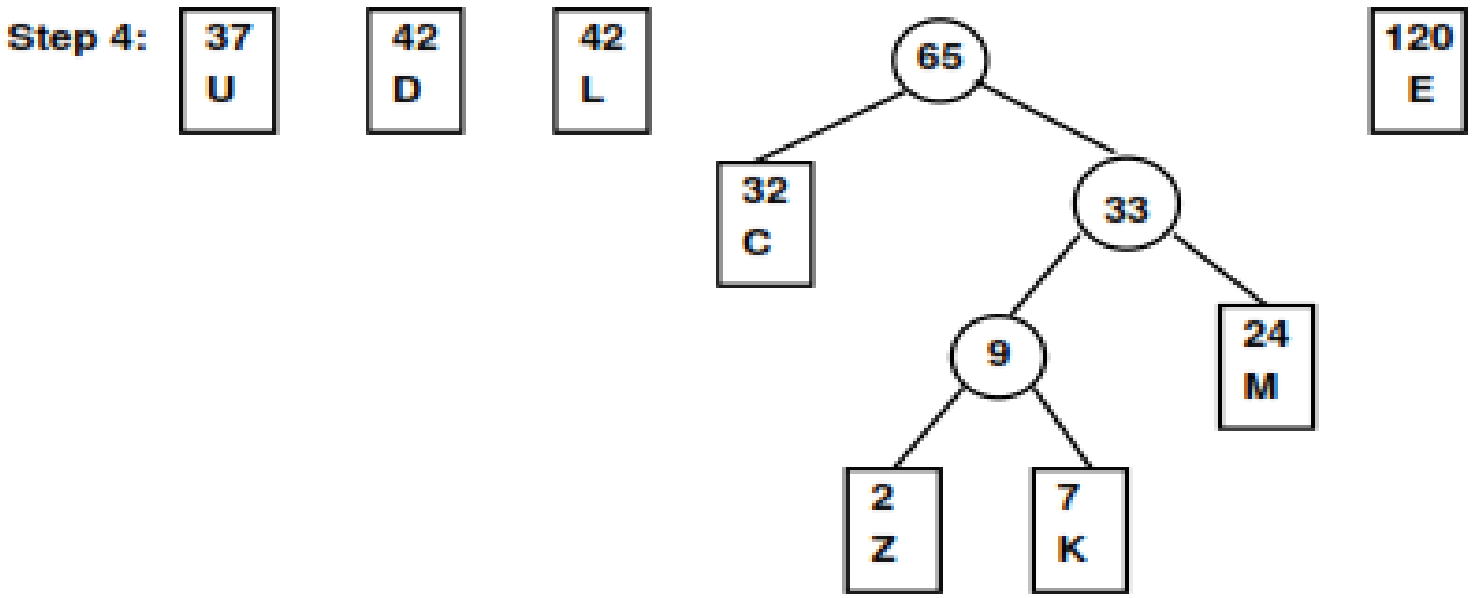


Huffman Tree Construction (Cont.)

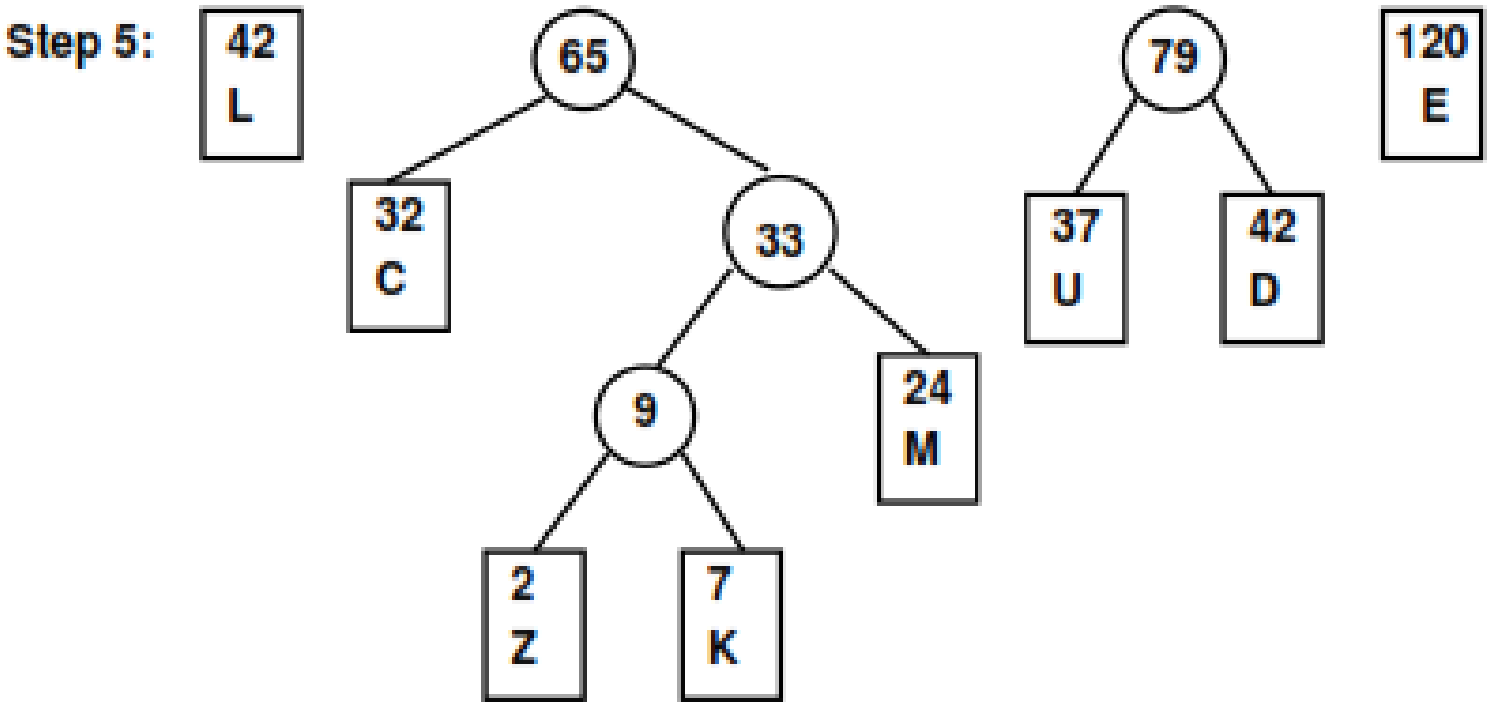


Huffman Tree Construction (Cont.)

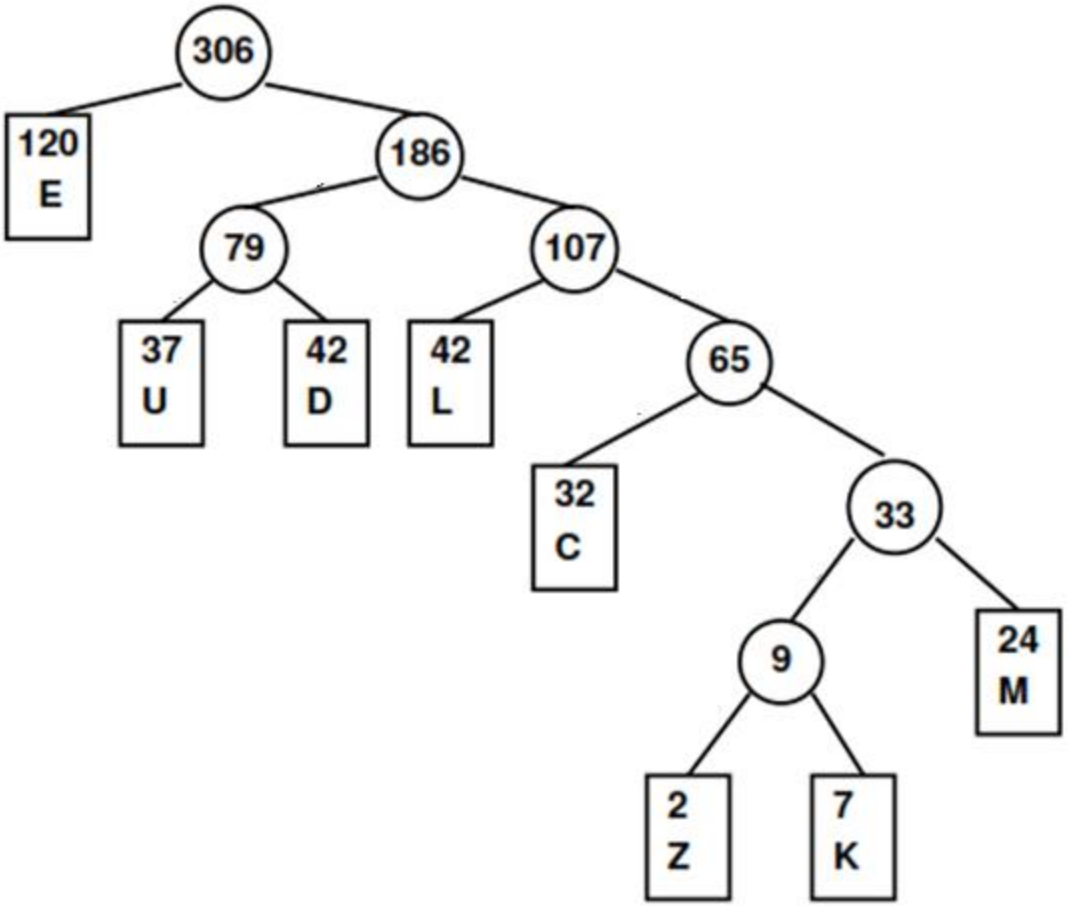
- Process continues until entire tree is built.



Huffman Tree Construction (Cont.)



Huffman Tree Construction (Cont.)

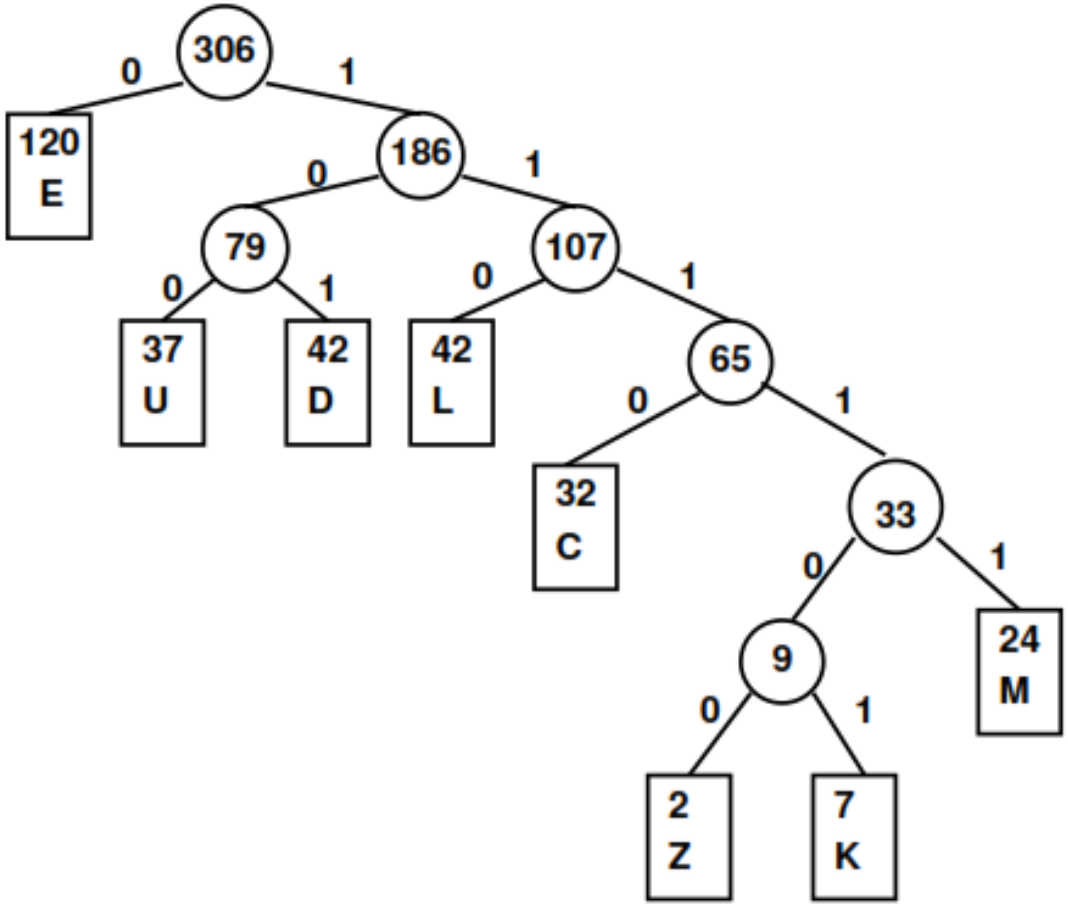


Assigning and Using Huffman Codes

- Assign either a '0' or a '1' to each edge in the tree
- Left branch (edge connecting a node with its left child): '0'
- Right branch (edge connecting a node with its right child): '1'

The Huffman code for a letter is simply a binary number determined by the path from the root to the leaf corresponding to that letter.

Assigning Codes (Cont.)



Using Huffman Codes

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

Fig: The Huffman Codes for the letters

Expected Cost for Huffman Coding

$$\frac{c_1 f_1 + c_2 f_2 + \cdots + c_n f_n}{f_T}$$

where f_i is the (relative) frequency of letter i and f_T is the total for all letter frequencies. For this set of frequencies, the expected cost per letter is

$$[(1\times120)+(3\times121)+(4\times32)+(5\times24)+(6\times9)]/306 = 785/306 \approx 2.57$$

A fixed-length code for these eight characters would require $\log 8 = 3$ bits per letter as opposed to about 2.57 bits per letter for Huffman coding. Thus, Huffman coding is expected to save about 14% for this set of letters.

Coding and Decoding

Examples:

- Code for DEED: ……
- Code for MUCK: ……
- Decode 1011001110111101: ……..
- Decode 11110100110 : ……..

Next Week Lecture (Week 6)

Lecture 6: Non-Binary Trees

Thank you!