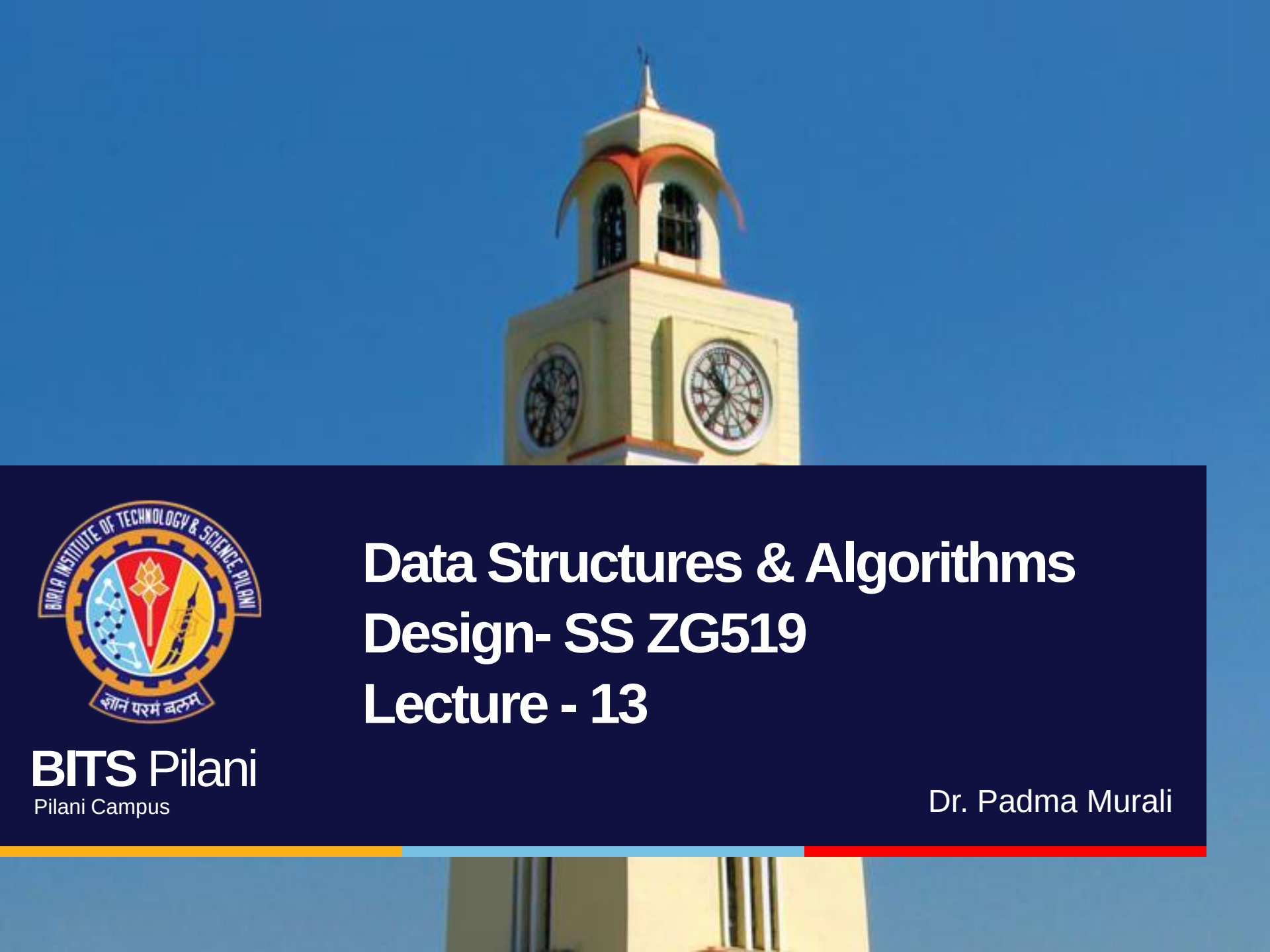


Data Structures & Algorithms Design- SS ZG519
Lecture - 13

BITS Pilani
Pilani Campus

Dr. Padma Murali

Lecture 13 Topics

- Greedy Algorithms- Huffman codes
- Graph Algorithms- Introduction

Huffman Codes

- Widely used technique for data compression
- Assume the data to be a sequence of characters
- Looking for an effective way of storing the data

Binary character code

  - Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- 3 bits needed
- a = 000, b = 001, c = 010, d = 011, e = 100, f = 101
- Requires: 100,000 · 3 = 300,000 bits

Huffman Codes

innovate achieve lead

- **Use the frequencies of occurrence of characters to build a optimal way of representing each character**

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Huffman Codes

Huffman Trees

* Huffman trees are used to encode a text that comprises n characters from some alphabet by assigning to each of the text's characters some sequence of bits called the codeword.

Huffman Codes

* creates variable length coding
  → (i.e.) different lengths for different alphabets according to the frequency of occurrence.
* creates prefix-free codes.
  (i.e.) no codeword is a prefix of a codeword of another character.

Huffman Codes

* While creating a binary prefix code, the characters of the text are represented by the leaves of the binary tree
* Left edges are labelled 0 and right edges are labelled 1.
* the codeword of a character is obtained by recording the labels on the path from the root to the character's leaf.

Huffman's Algorithm

step 1 Initialise n one-node trees and label them with the characters of the alphabet.
Record the frequency of each character in its' tree's root to indicate the tree's weight.
More generally the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.

Huffman's Algorithm

Step 2 Repeat the following operation until a single tree is obtained:
Find two trees with the smallest weight.
Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

Huffman's Algorithm

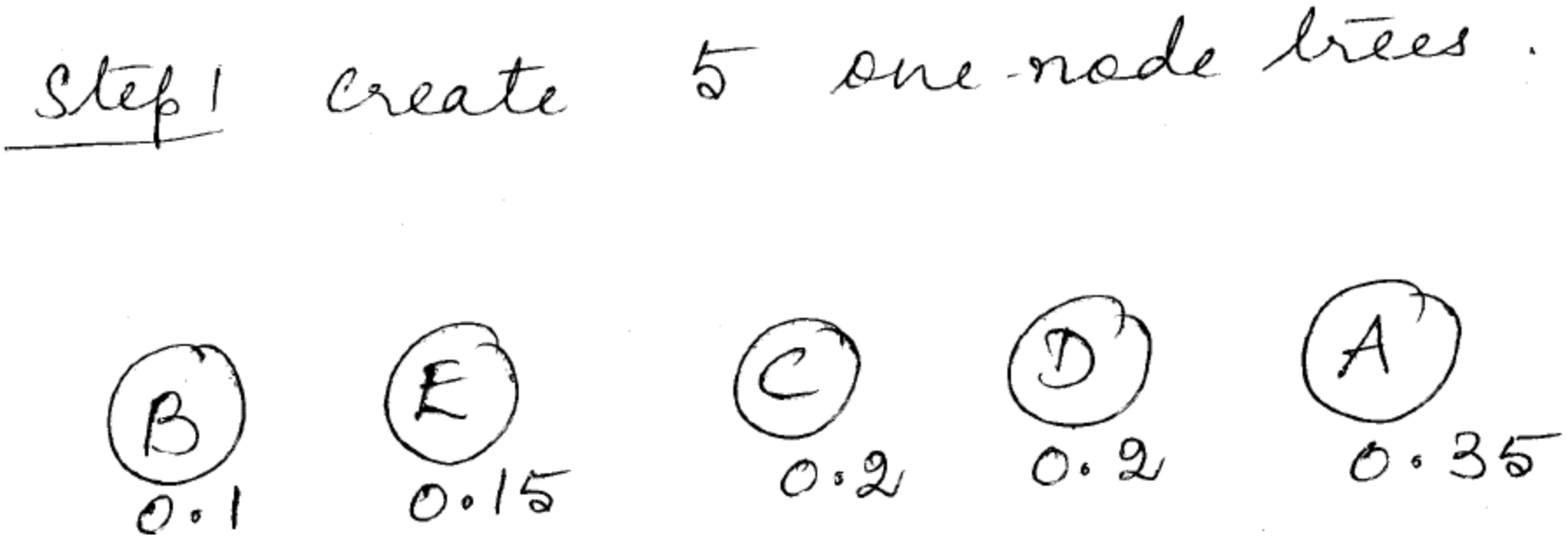
A tree constructed by the above algorithm is called a Huffman tree.
It defines in the manner described.
a Huffman code.

Example 1

Construct a Huffman code for the following data.

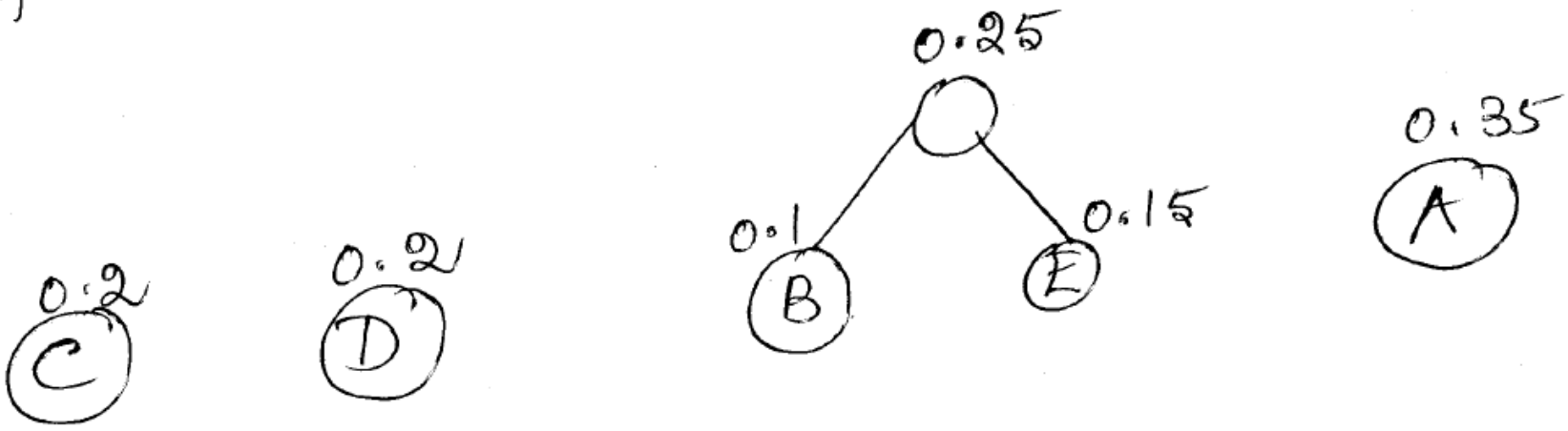
Character	A	B	C	D	E
Frequency	0.35	0.1	0.2	0.2	0.15

Example 1

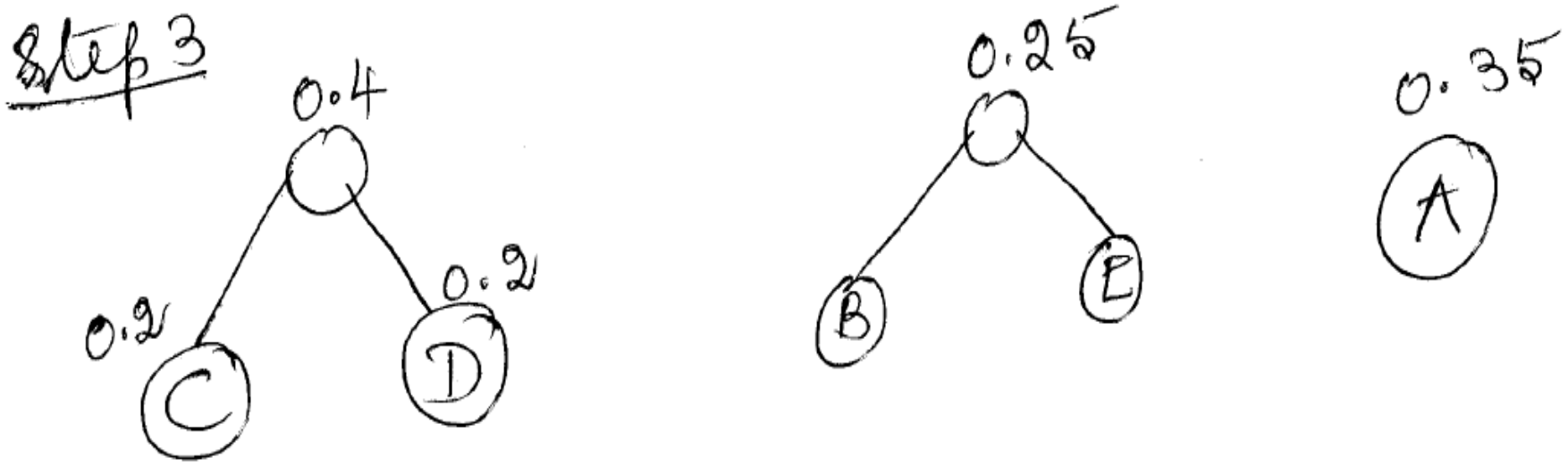


Example 1

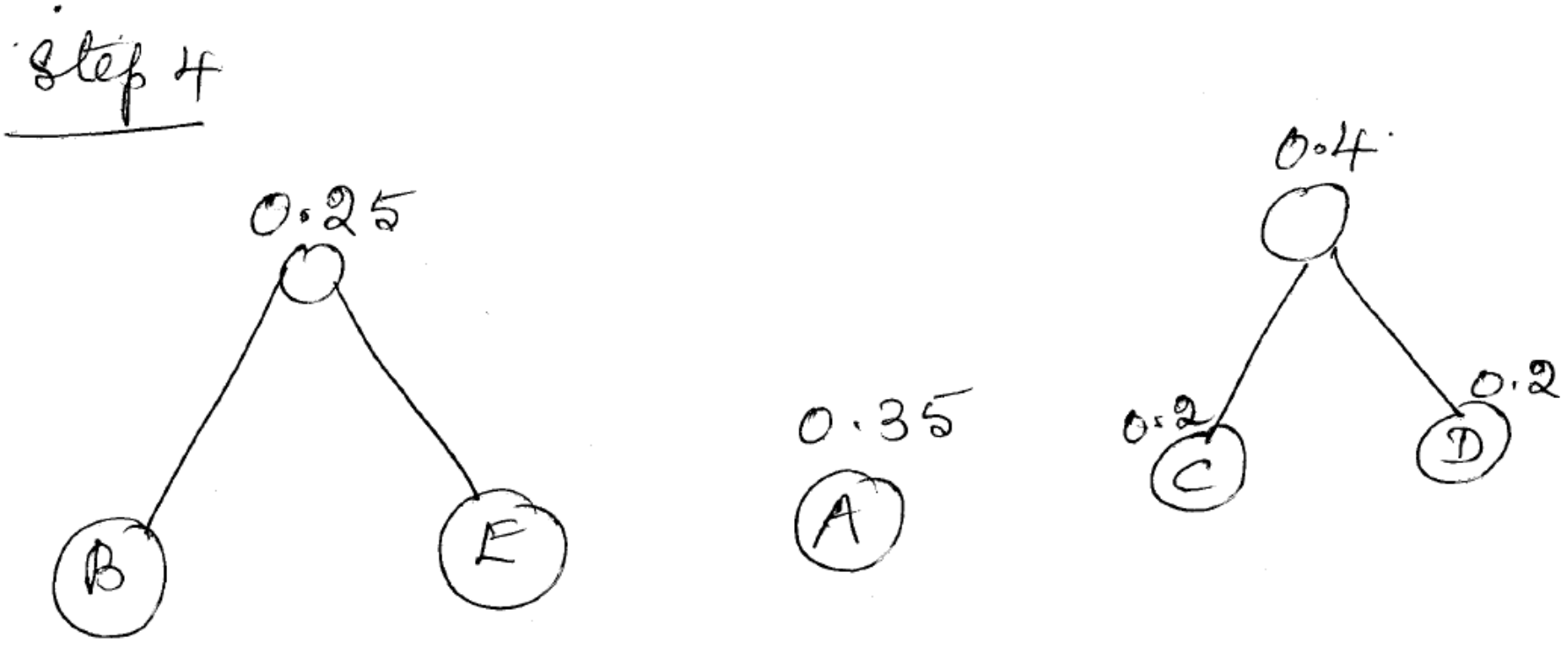
Step 2 choose the 2 characters with least frequencies and combine them to form a new tree.



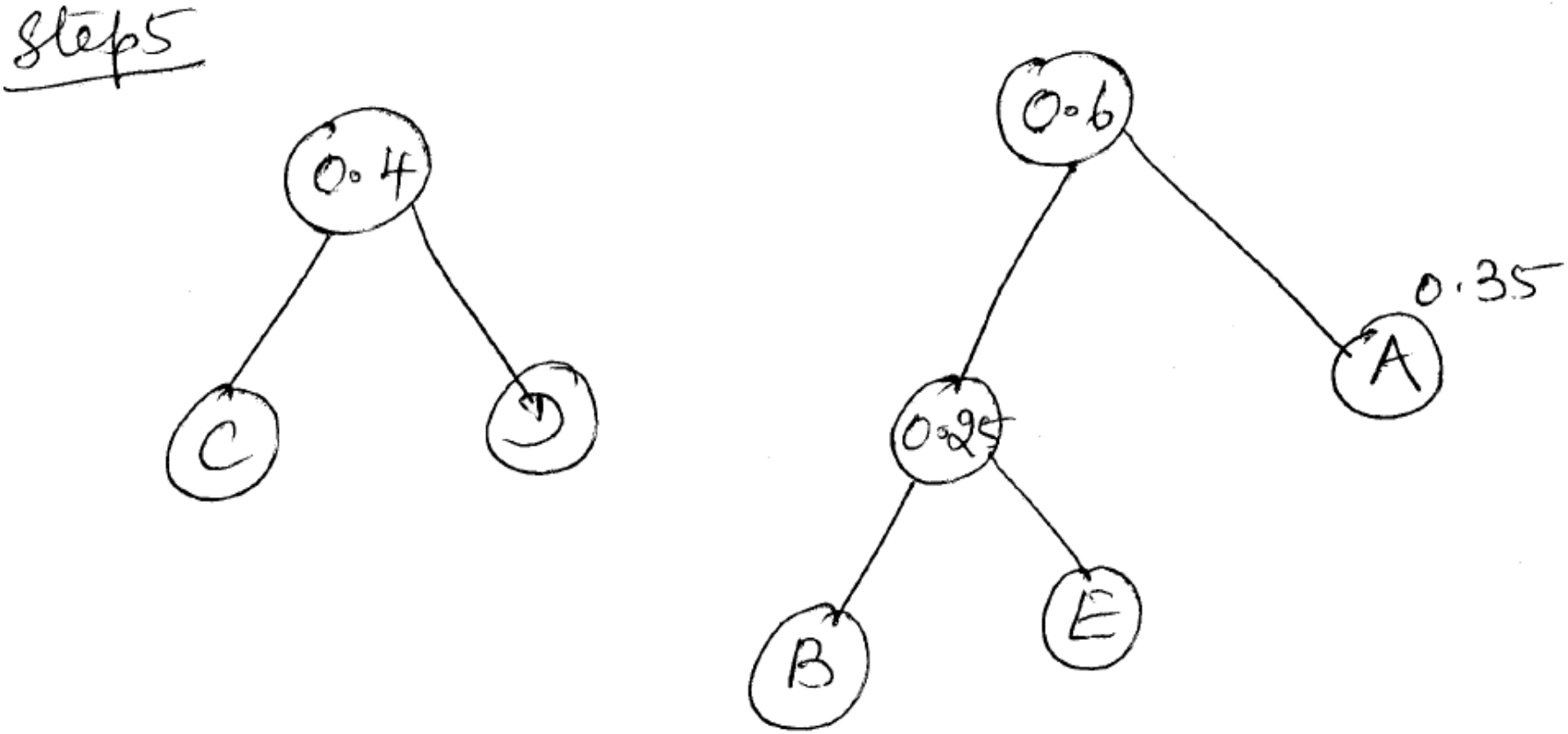
Example 1



Example 1

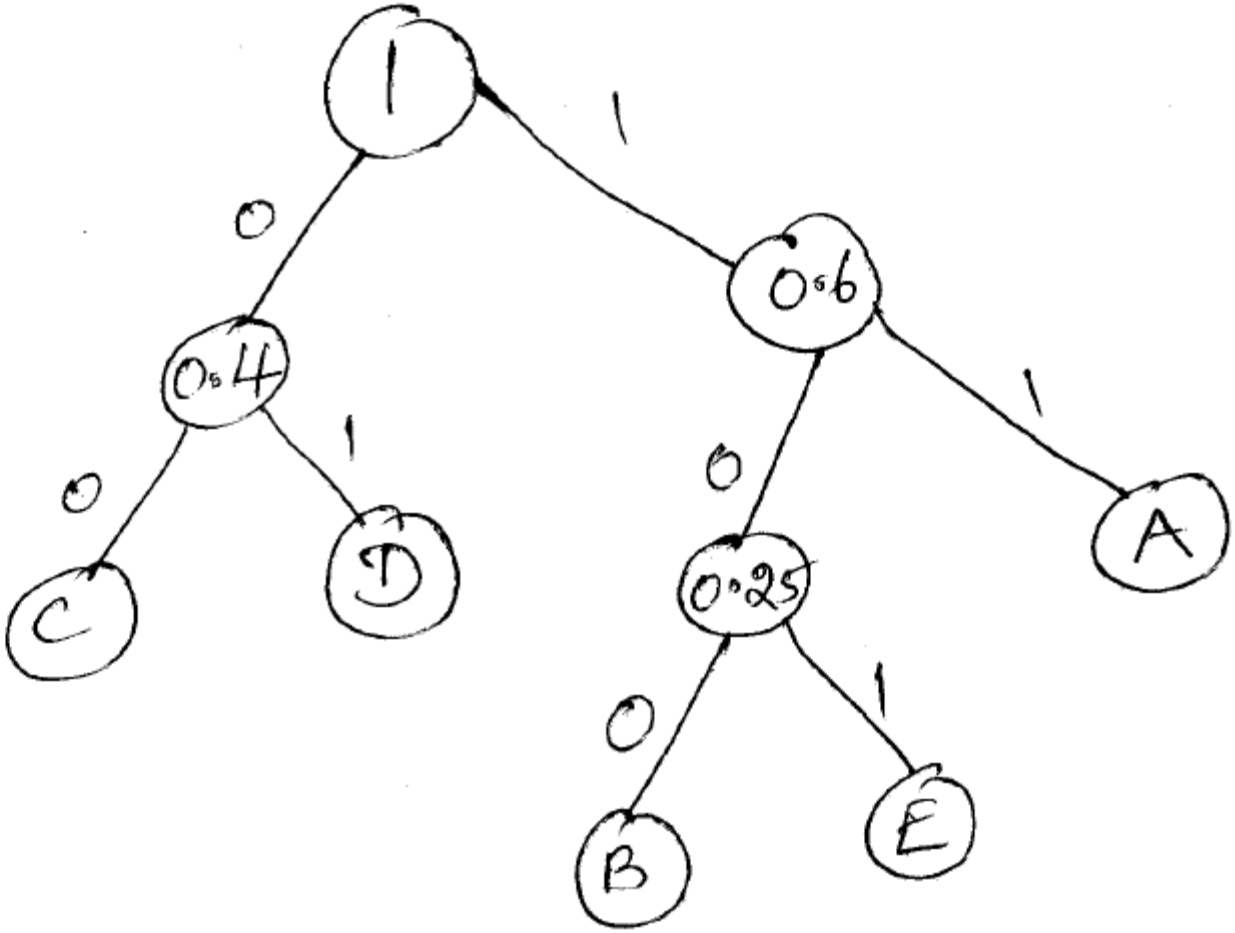


Huffman Codes



Huffman Codes

Step 6



Huffman Codes

Codewords for the character

Character	A	B	C	D	E
code	11	100	00	01	101

Huffman Codes

* Huffman's encoding - one of the most important file compression methods.
* It yields an optimal (ie) a minimal length encoding.

Huffman Codes

Using the above example,

1) DAD is encoded as

011101.

2) 1001110111101 is decoded as

BADEAD.

Huffman tree construction

– Example of greedy technique

Construction of the Huffman Tree

The Huffman algorithm generates the most efficient binary code tree at given frequency distribution. Prerequisite is a table with all symbols and their frequency. Any symbol represents a leaf node within the tree.

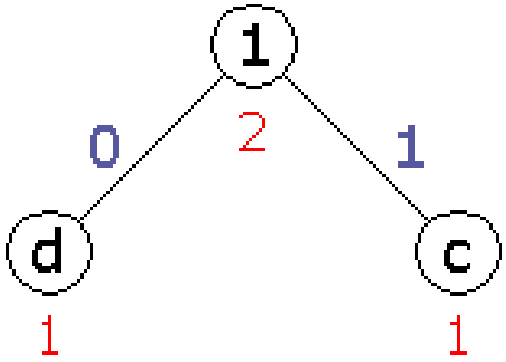
The following general procedure has to be applied:

- search for the two nodes providing the lowest frequency, which are not yet assigned to a parent node
- couple these nodes together to a new interior node
- add both frequencies and assign this value to the new interior node

- The procedure has to be repeated until all nodes are combined together in a root node.
- Example: "abracadabra"
-
- Symbol Frequency
- a 5
- b 2
- r 2
- c 1
- d 1
-

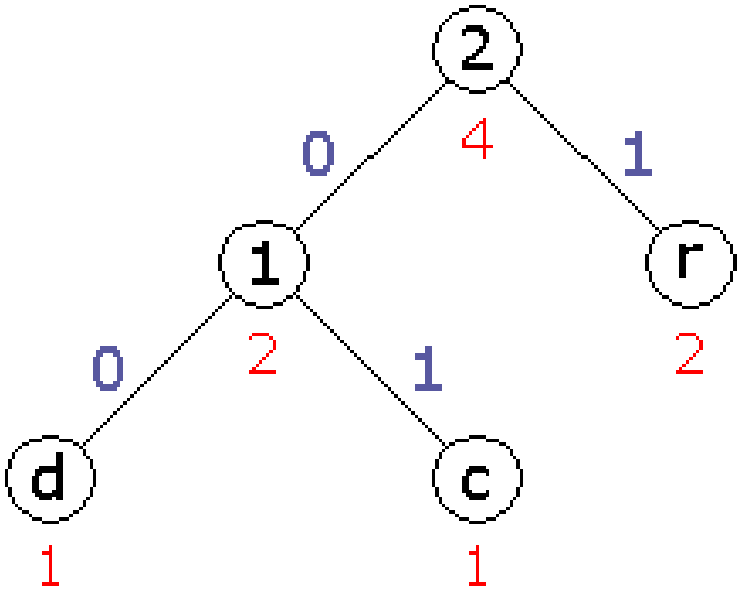
- According to the outlined coding scheme the symbols "d" and "c" will be coupled together in a first step. The new interior node will get the frequency 2.
- 1. Step
-
- Symbol Frequency Symbol Frequency
- a 5 a 5
- b 2 b 2
- r 2 r 2
- c 1 ----------> 1 2
- d 1

- Code tree after the 1st step:



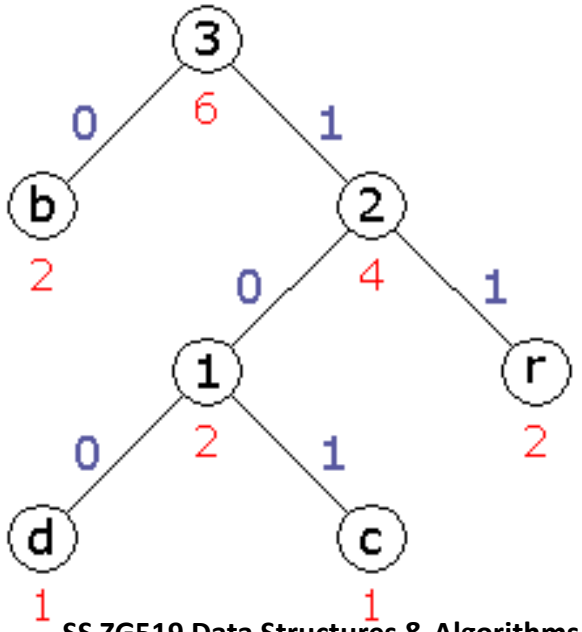
- 2. Step
-
- Symbol Frequency Symbol Frequency
- a 5 a 5
- b 2 b 2
- r 2 ----------> 2 4
- 1 2

- Code tree after the 2nd step:



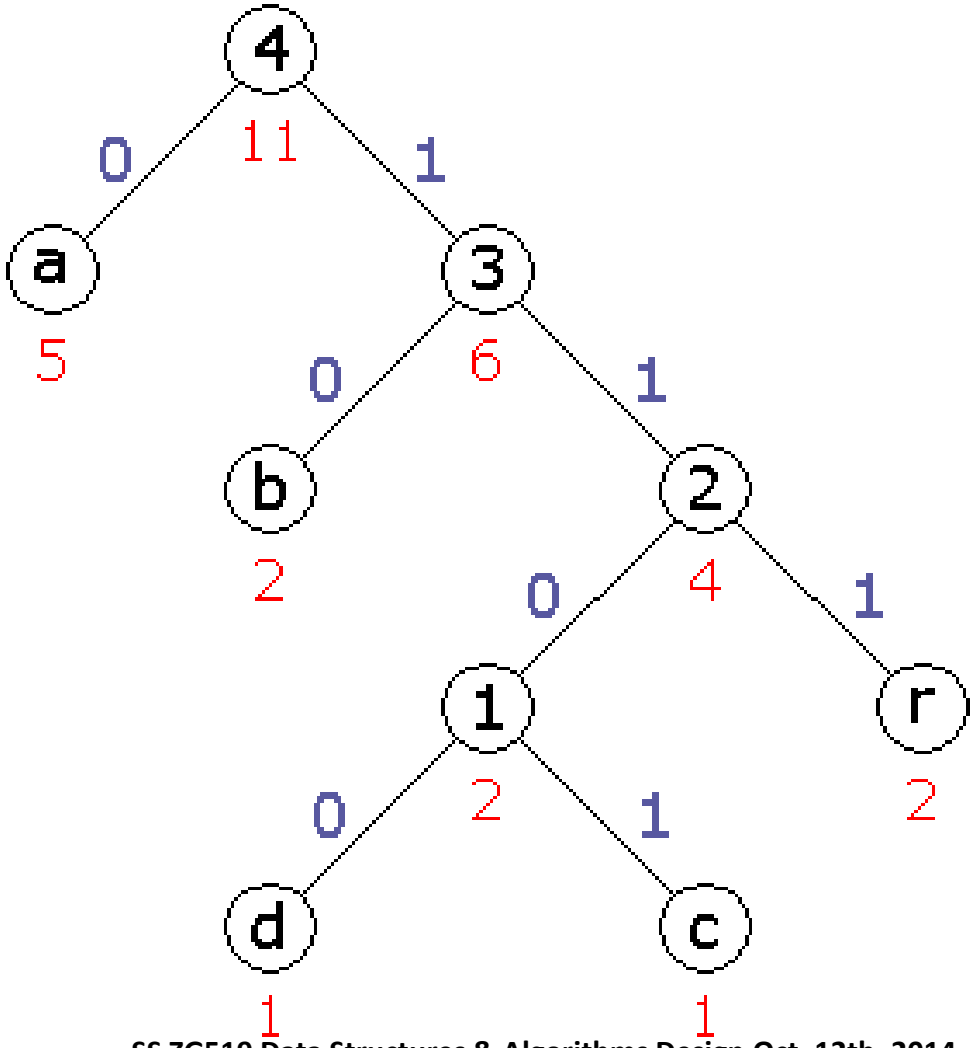
- 3. Step
-
- Symbol Frequency Symbol Frequency
- a 5 a 5
- 2 4 ----------> 3 6
- b 2

- Code tree after the 3rd step:



- 4. Step
-
- Symbol Frequency Symbol Frequency
- 3 6 ----------> 4 11
- a 5
-
-

- Code

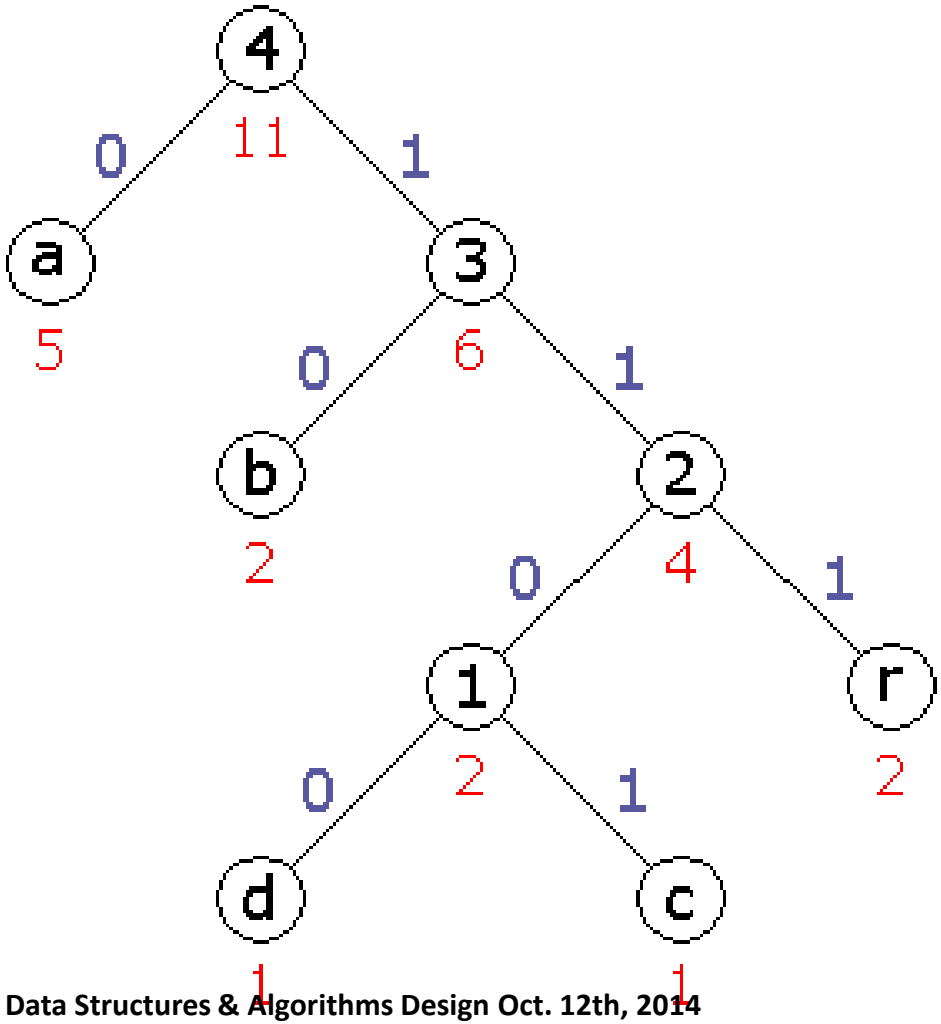


- Code Table

- If only one single node is remaining within the table, it forms the root of the Huffman tree. The paths from the root node to the leaf nodes define the code word used for the corresponding symbol:

Symbol	Frequency	Code Word
a	5	0
b	2	10
r	2	111
c	1	1101
d	1	1100

- Complete Huffman Tree:



- Encoding

- The original data will be encoded with this code table as follows:
-
- Symbol Frequency Code Word
- a 5 0
- b 2 10
- r 2 111
- c 1 1101
- d 1 1100
-

- a b r a c a d a b r a
- 0 10 111 0 1101 0 1100 0 10 111 0
-
- encoded data: 23 Bit
-

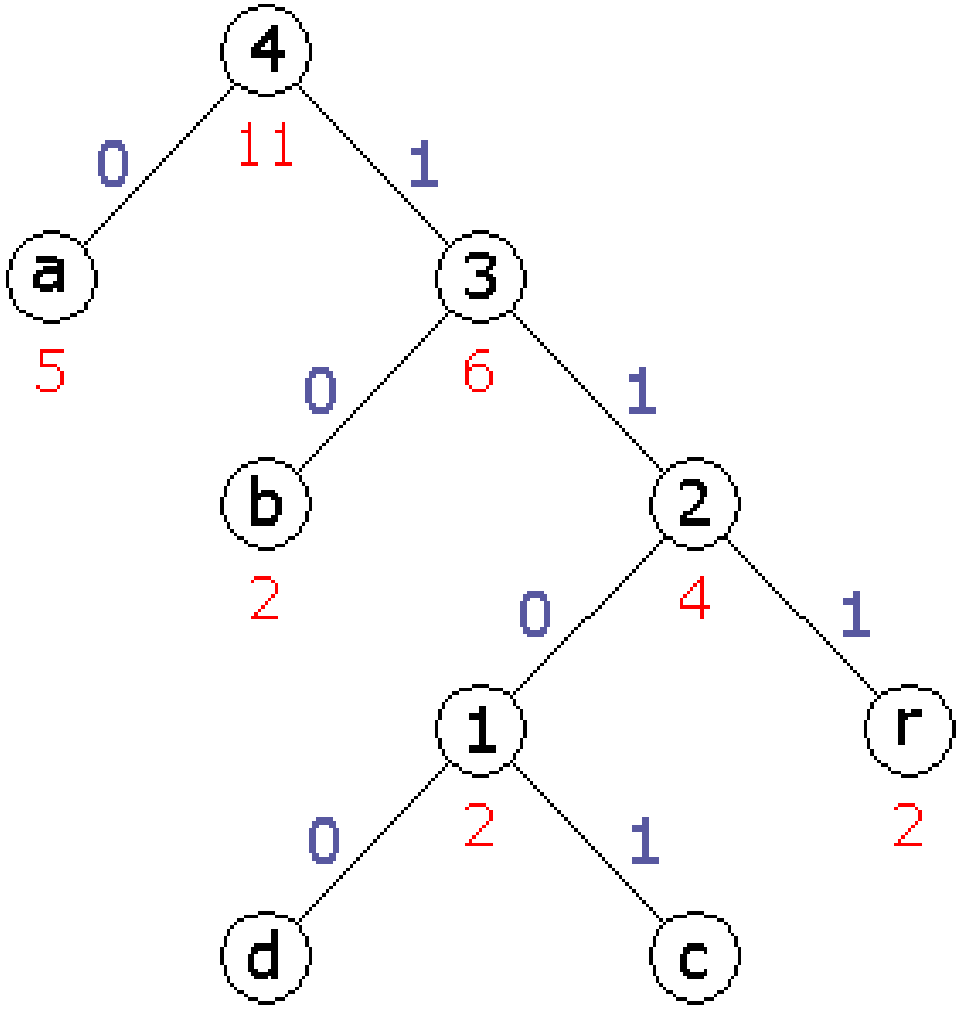
- Decoding

- For decoding the Huffman tree is passed through with the encoded data step by step. Whenever a node not having a successor is reached, the assigned symbol will be written to the decoded data.

01011101101011000101110

- encoded decoded
- 0 a
- 10 b
- 111 r
- 0 a
- 1101 c
- 0 a
- 1100 d
- 0 a
- 10 b
- 111 r

- Hu



Variable-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- Assign short codewords to frequent characters and long codewords to infrequent characters
- a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
- $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000$
  $= 224{,}000$ bits

Prefix Codes

- **Prefix codes:**
  - **Codes for which no codeword is also a prefix of some other codeword**
  - **Better name would be "prefix-free codes"**
- **We can achieve optimal data compression using prefix codes**
  - **We will** restrict our attention to prefix codes

Encoding with Binary Character Codes

- Encoding
  - Concatenate the codewords representing each character in the file

E.g.:

- a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
- abc = 0 · 101 · 100 = 0101100

Decoding with Binary Character Codes

Prefix codes simplify decoding

- No codeword is a prefix of another $\Rightarrow$ the codeword that begins an encoded file is unambiguous

Approach

- Identify the initial codeword
- Translate it back to the original character
- Repeat the process on the remainder of the file

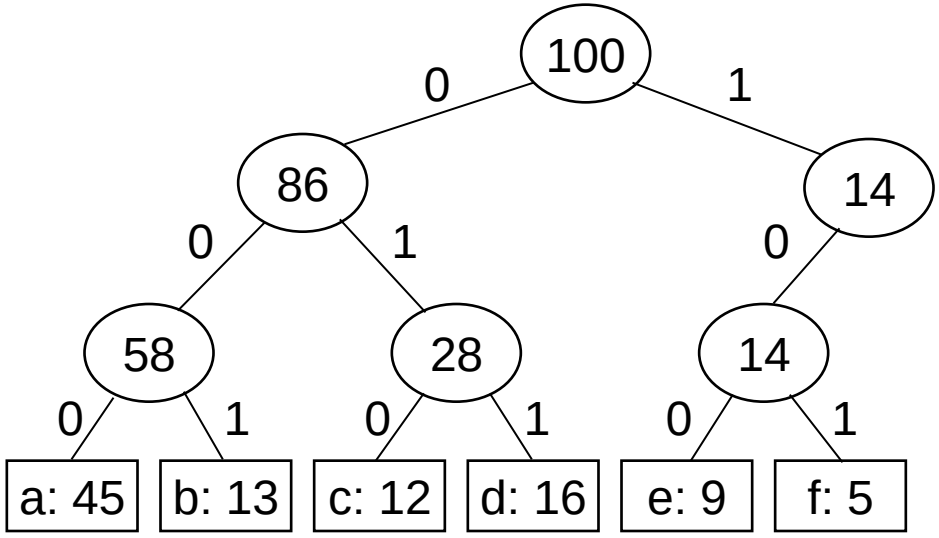
E.g.:

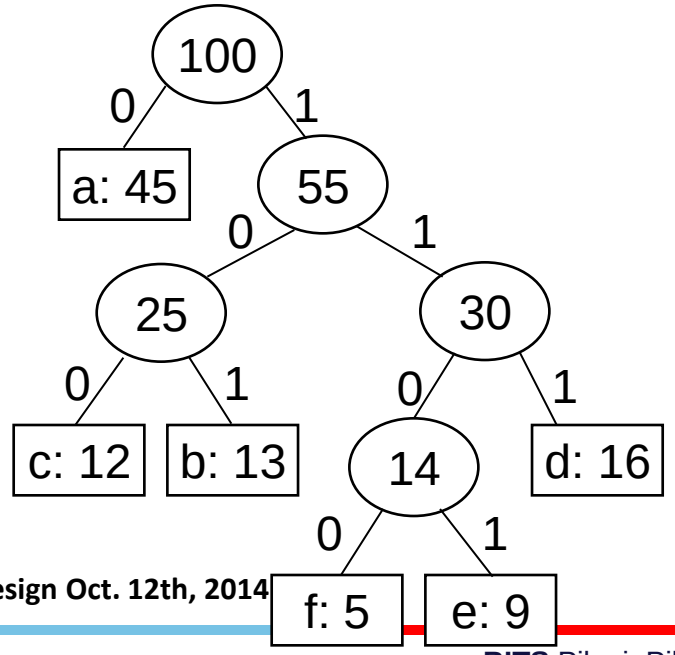
- a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

- 001011101 = $0 \cdot 0 \cdot 101 \cdot 1101 = aabe$

Prefix Code Representation

- Binary tree whose leaves are the given characters
- Binary codeword
  - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”
- Length of the codeword
  - Length of the path from root to the character leaf (depth of node)





Optimal Codes

- An optimal code is always represented by a **full binary tree**
  - Every non-leaf has two children
  - Fixed-length code is not optimal, variable-length is
- How many bits are required to encode a file?
  - Let C be the alphabet of characters
  - Let $f(c)$ be the frequency of character c
  - Let $d_T(c)$ be the depth of c's leaf in the tree T corresponding to a prefix code

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$ the cost of tree T

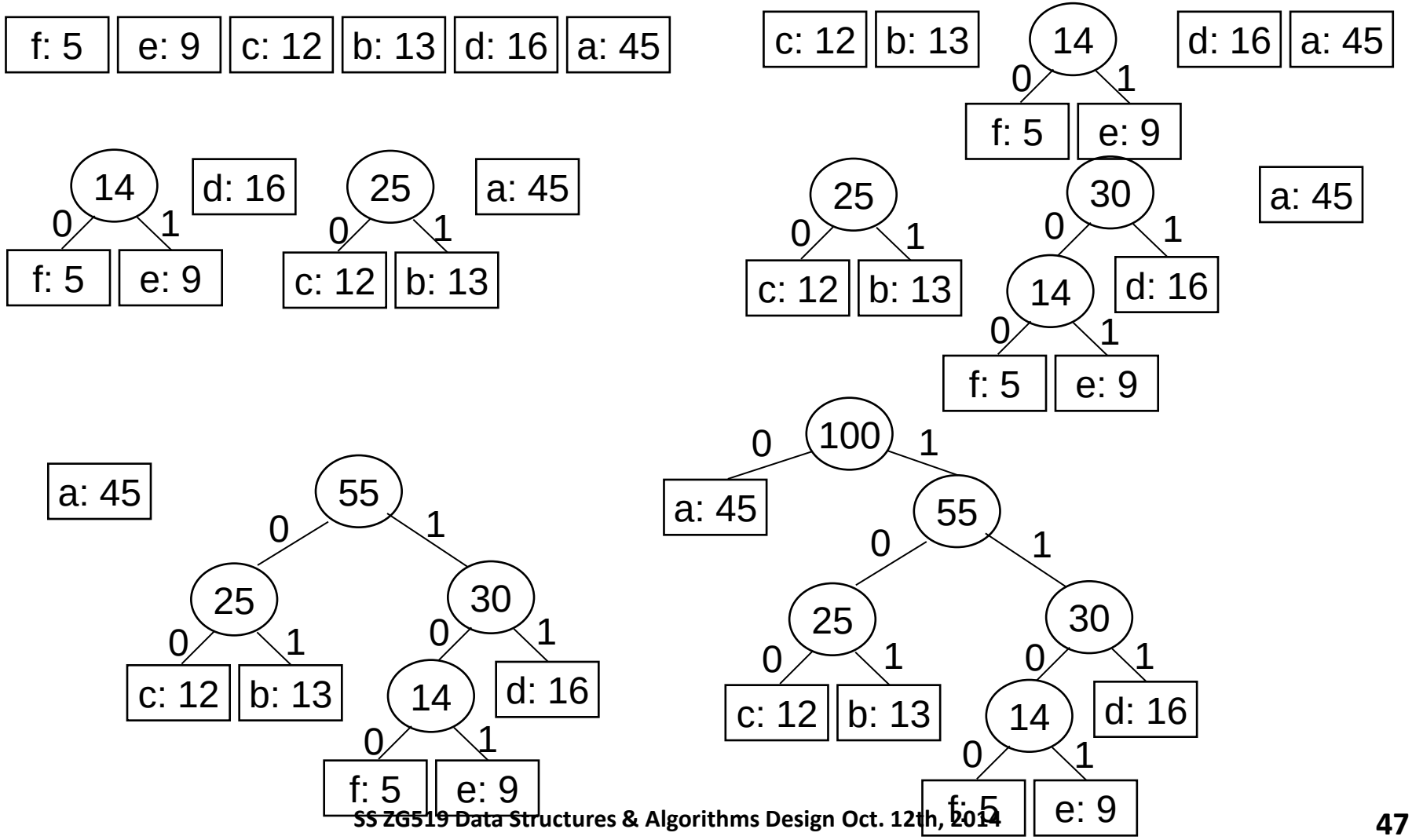
Constructing a Huffman Code

- **A greedy algorithm that constructs an optimal prefix code called a Huffman code**
- Assume that:
  - C is a set of n characters
  - Each character has a frequency $f(c)$
  - The tree T is built in a bottom up manner
- Idea:

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

  - Start with a set of $|C|$ leaves
  - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
  - Use a min-priority queue Q, keyed on f to identify the two least frequent objects

Example


f: 5
e: 9
c: 12
b: 13
d: 16
a: 45
14
0
1
f: 5
e: 9
d: 16
25
0
1
c: 12
b: 13
a: 45
c: 12
b: 13
14
0
1
f: 5
e: 9
d: 16
a: 45
25
0
1
c: 12
b: 13
30
0
1
14
0
1
f: 5
e: 9
d: 16
a: 45
55
0
1
25
0
1
c: 12
b: 13
30
0
1
14
0
1
f: 5
e: 9
d: 16
a: 45
100
0
1
55
0
1
25
0
1
c: 12
b: 13
30
0
1
14
0
1
f: 5
e: 9
d: 16

Building a Huffman Code

Running time: $O(n \lg n)$

Alg.: HUFFMAN(C)

1. $n \leftarrow |C|$
2. $Q \leftarrow C$ ← $O(n)$
3. **for** $i \leftarrow 1$ **to** $n - 1$
4. **do** allocate a new node **z**
5. $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN(Q)
6. $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN(Q)
7. $f[z] \leftarrow f[x] + f[y]$
8. INSERT (Q, **z**)
9. **return** EXTRACT-MIN(Q)

} $O(n \lg n)$

- Example

Calculate the Huffman code for the following characters

Character	Frequency
'a'	12
'b'	2
'c'	7
'd'	13
'e'	14
'f'	85

Example

Letter	Code
'a'	001
'b'	0000
'c'	0001
'd'	010
'e'	011
'f'	1

Decode the bits 1100010000010010

The string is ffcbdd

innovate achieve lead

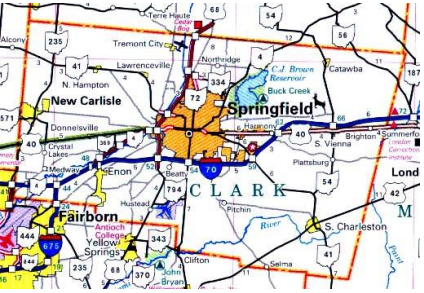
BITS Pilani

Pilani Campus

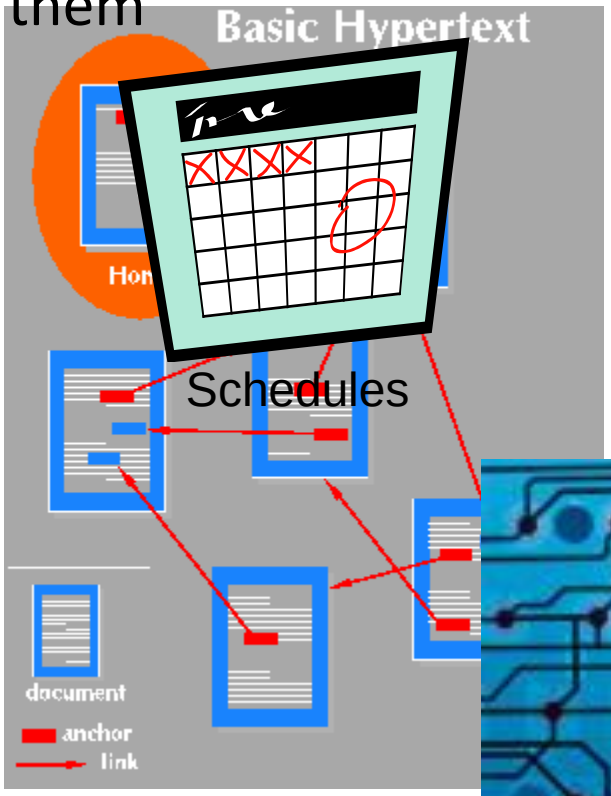
- **Graphs**

Graphs

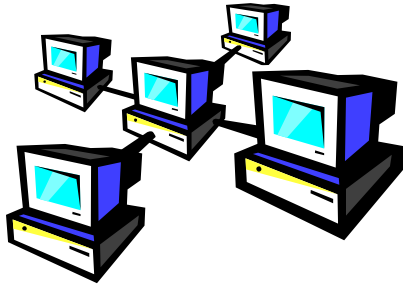
- Applications that involve not only a set of items, but also the connections between them



Maps

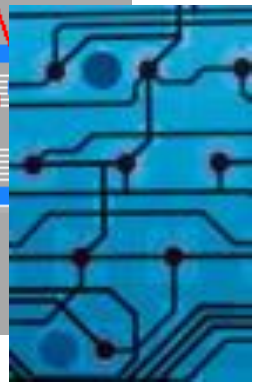


Schedules



Computer networks

Hypertext



Circuits

Graphs - An Introduction to Graphs

Graph theory is considered to have begun in the year 1736.

Graphs are used for modeling a wide variety of real life applications.

Some of the applications of graph theory are in

1. Scheduling problems- For Example: Project Scheduling

2. Computer networks, communication networks

More applications of graphs

3. Circuits

4. Hypertext

5. Maps

6. Games

7. Web's diameter- which is the maximum number of links one needs to follow to reach one web page from another by the most direct route between them.

What is a Graph ?

Example:

Four students- A, B, C, D have completed their course work in BITS, Pilani.

They are applying for their final project internship.

There are 4 different organisations: C1, C2, C3, C4 which offer projects.

The following are the preferences of the students.

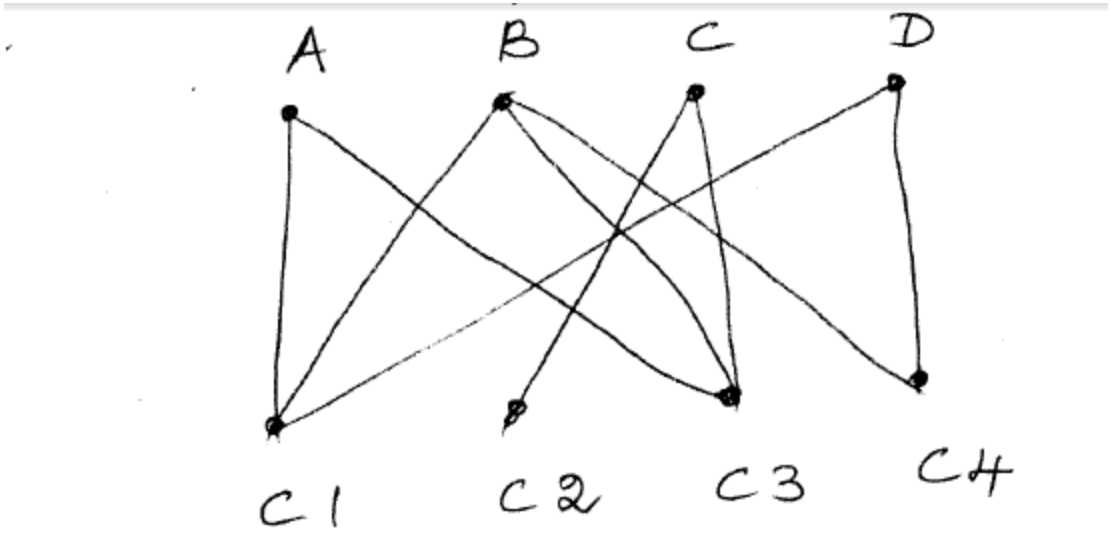
A prefers C1 and C3.

B prefers C1, C3, and C4.

C prefers C2 and C3.

D prefers C1 and C4.

The preferences of the students can be diagrammatically represented in the following manner.



The above diagram is called a Graph.

Graph:

A graph G is a finite non empty set V(G) of objects called **vertices (also called as nodes)** and a set E(G) of two element subsets of V(G) called **edges.**

V(G) is called the **vertex set** of graph G and

E(G) is called the **edge set of graph G.**

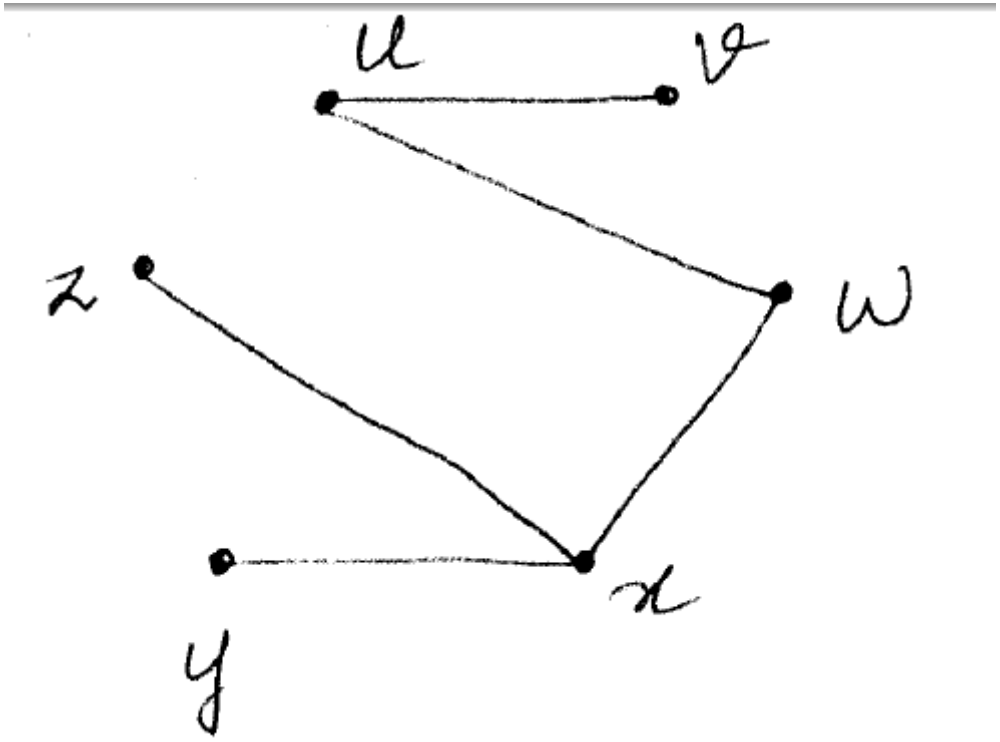
Let G be a graph and {u, v}, an edge of G. Since,{u, v} is a 2-element set, we may write {v, u} instead of {u, v}. Conveniently, we represent this edge by uv or vu.

Adjacent Vertices: If e = vu is an edge of a graph G, then we say that u and v are adjacent in G.

Example: A graph G is defined by the sets

V(G) = {u,v,w,x,y,z}

E(G) = {uv,uw,wx,xy,xz}

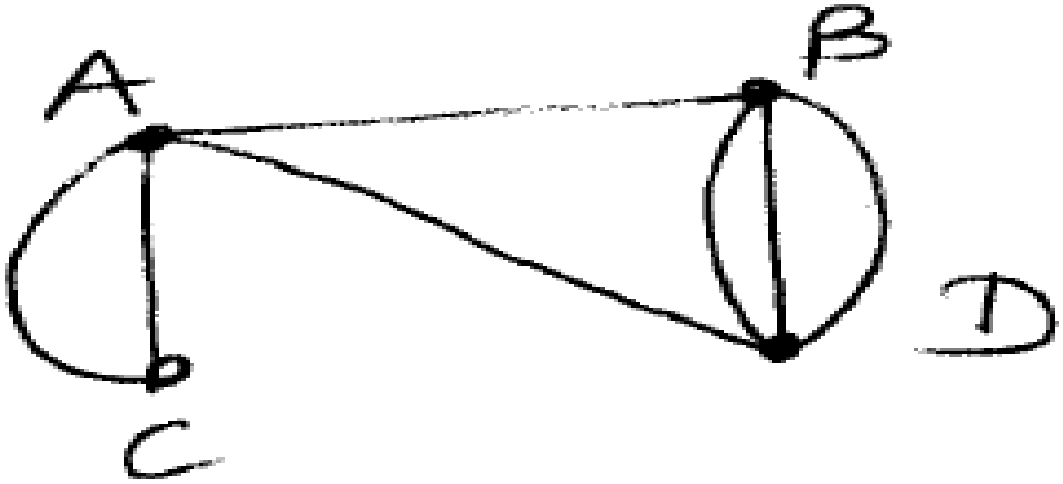


Every graph has a diagram associated with it. The diagrams are useful for understanding problems involving such a graph. The vertices are represented by means of points and by joining two points by means of a line segment is an edge.

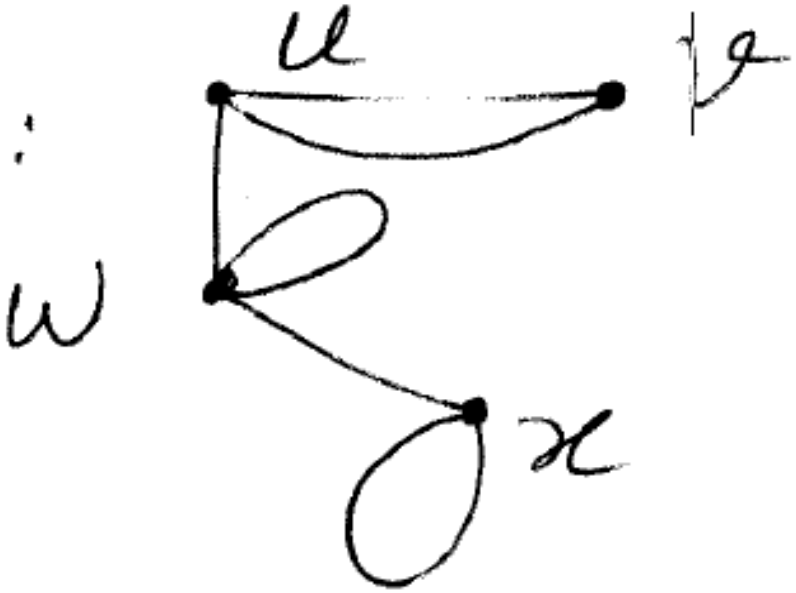
Parallel Edges: Two or more edges that join the same pair of vertices are called parallel edges.

Example: In a road network, more than one road may connect a pair of cities.

Multigraph: If in a graph, there are parallel edges, such a graph is called a multigraph.



Loop: An edge that joins a vertex to itself is called a loop.



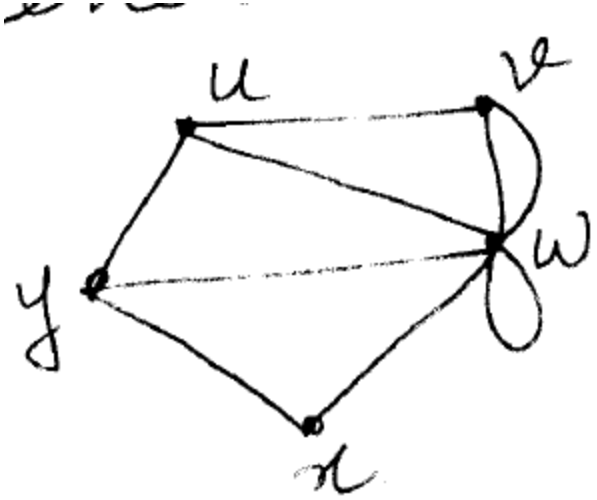
Psuedograph: A graph that contains both parallel edges and loops is called a psuedograph.

Order: The number of vertices in a graph G is called its order.

Size: The number of edges in a graph G is called its size.

Degree of a vertex: The number of edges incident on a vertex is called the degree of a vertex.

Example:



$\deg(u) = 3$

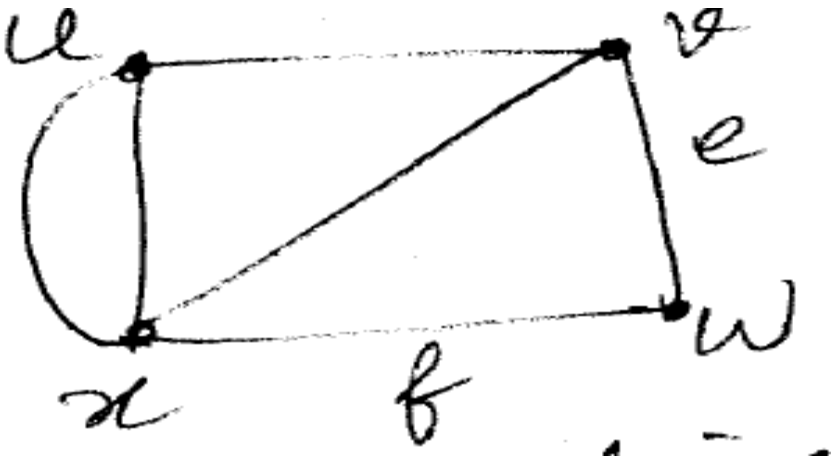
$\deg(v) = 3$

$\deg(w) = 4$

$\deg(x) = 2$

$\deg(y) = 3.$

Adjacent Edges: If e and f are distinct edges that are incident with a common vertex, then e and f are adjacent edges.



Order of graph = 4
Size of graph = 6

e and f are adjacent edges and u and v are adjacent vertices.

Isolated vertex : A vertex of degree 0 is called an isolated vertex.

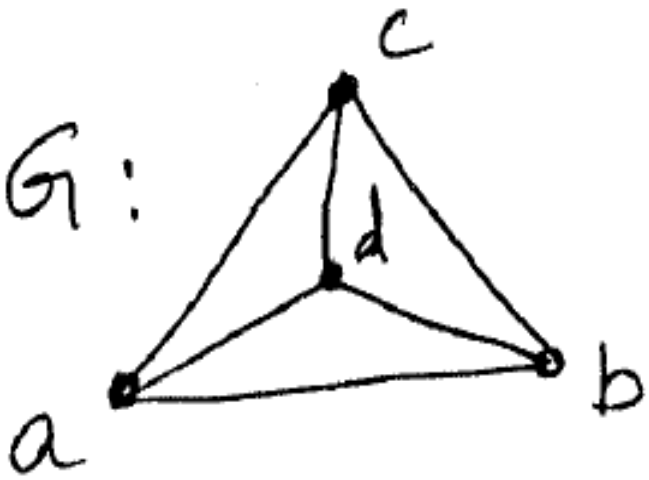
End vertex: A vertex of degree 1 is called an end vertex.

Even vertex: A vertex is called even if the degree of the vertex is even.

Odd vertex: A vertex is called odd if the degree of the vertex is odd.

Regular Graph: A graph G is r-regular or regular of degree r, If every vertex of G has degree r.

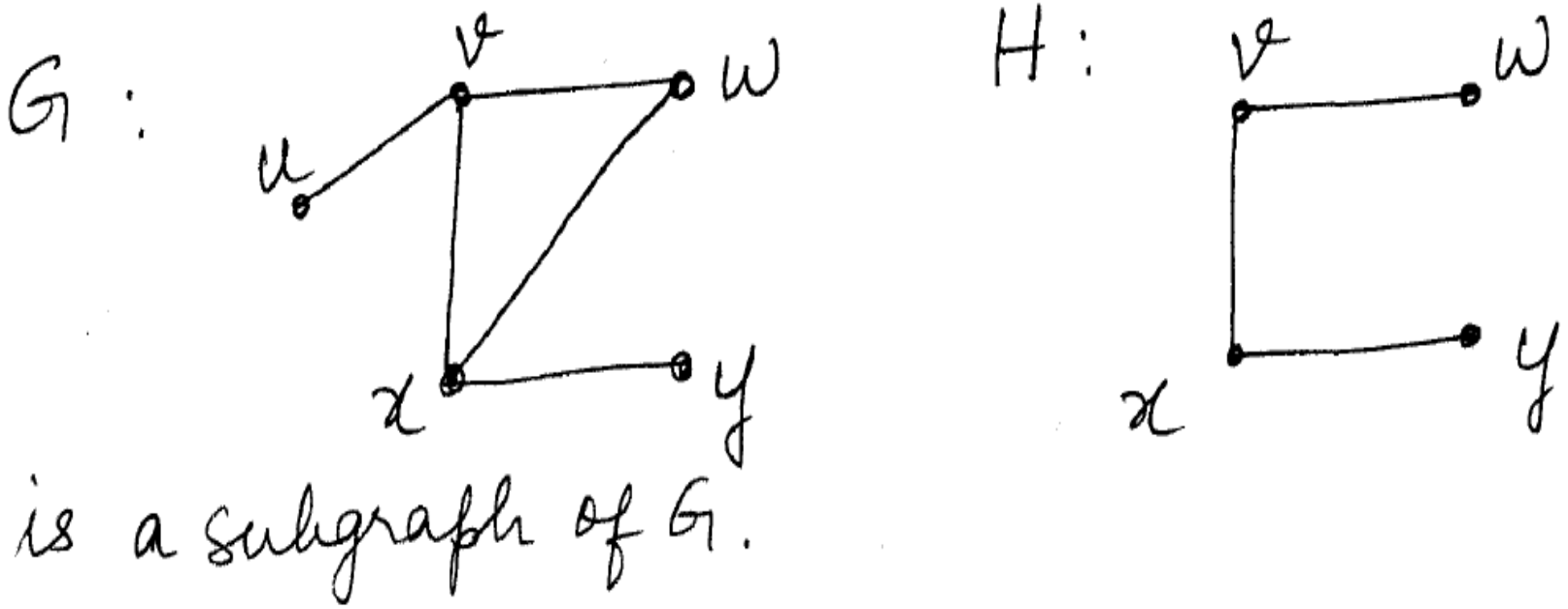
Example:



G is a regular graph of degree 3.

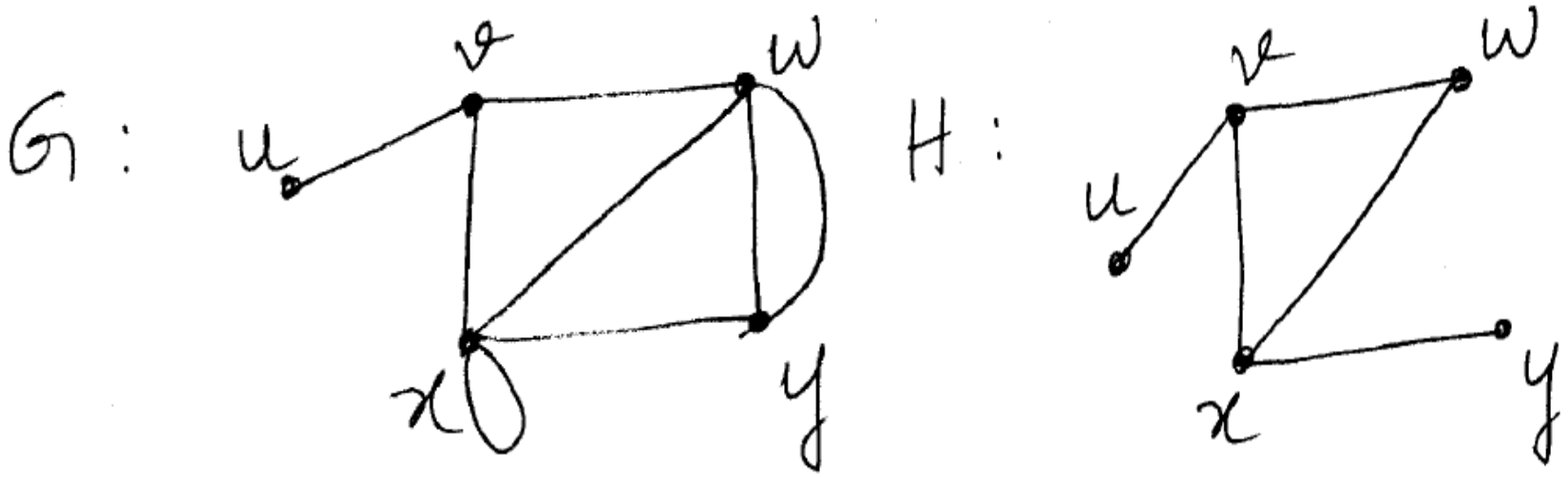
Subgraph: A graph H is a subgraph of a graph G if

$$V(H) \subseteq V(G) \text{ and } E(H) \subseteq E(G).$$



H is a subgraph of G.

Spanning subgraph: A subgraph H of a graph G is a spanning subgraph of G if V(H) = V(G).



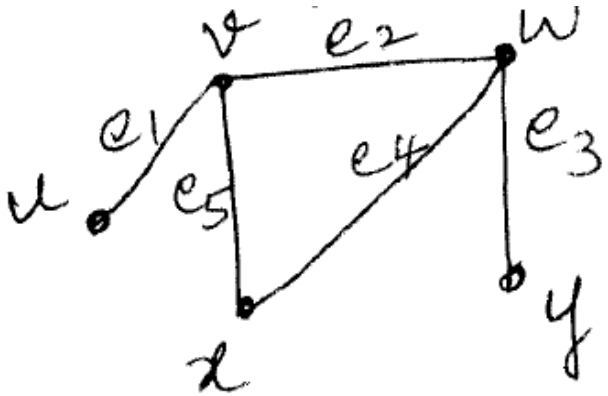
H is a spanning subgraph of G.

Walk: A walk in a graph G is an alternating sequence of vertices and edges.

$$W : v_0, e_1, v_1, e_2, v_2 \ldots\ldots e_n, v_n$$

Example :

G :



W : $u, e_1, v, e_5, x, e_4, w$ is a walk

Length of a walk: A walk is of length n if it has n edges.
In the previous example, walk

$W: u, e_1, v, e_5, x, e_4, w$

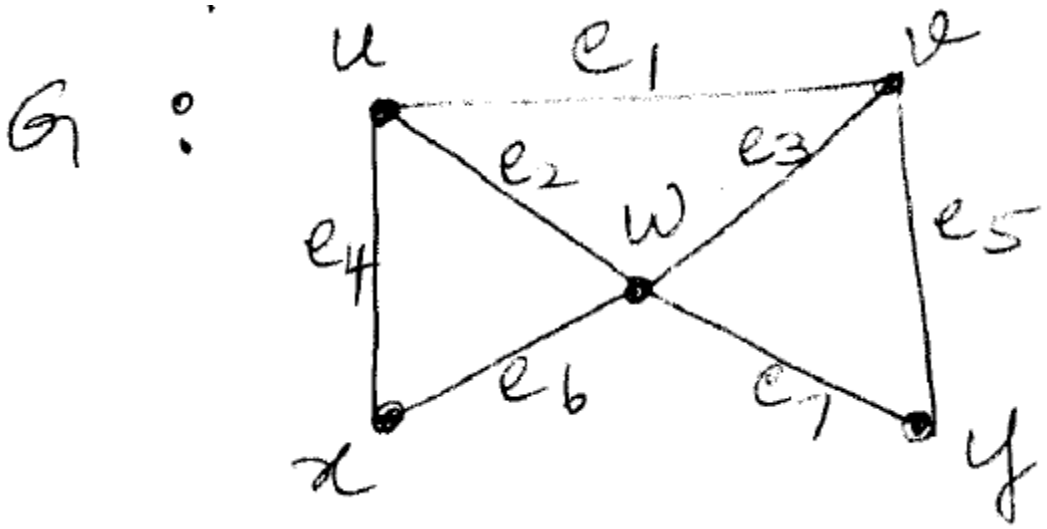
is a walk of length 3.

Trivial walk: A walk of length zero is called a trivial walk.

Trail: A trail is a walk in which no edge is repeated.

Path: A path is a walk in which no vertex is repeated.

Example:



In G, the walk x, w, v, u, w, y is a trail that is not a path.

u x w v y is a path.

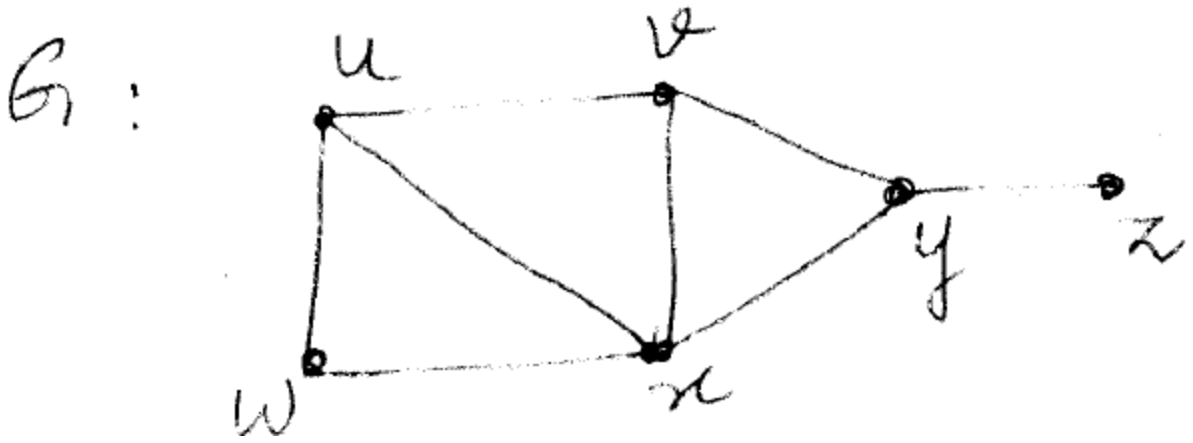
Cycle or Circuits: A cycle is a walk $v_0, v_1, v_2 \ldots\ldots v_n$

in which n >= 3, $v_0 = v_n$ and the n vertices

$v_1, v_2 \ldots\ldots v_n$ are all distinct.

A cycle of length n is referred to as an n-cycle.

Example:

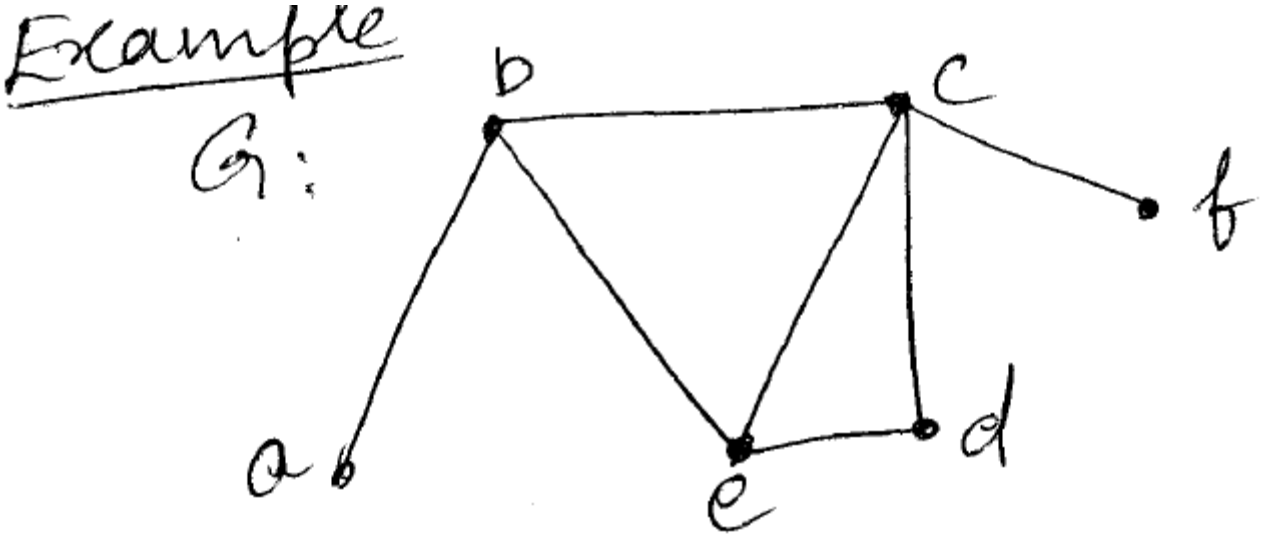


In G, u-v-x-w-u is a cycle of length 4.

Connected: Let u and v be vertices in a graph G. We say that u is connected to v if G contains a u-v path.

Connected graph: The graph G is connected if u is connected to v for every pair u,v of vertices of G.

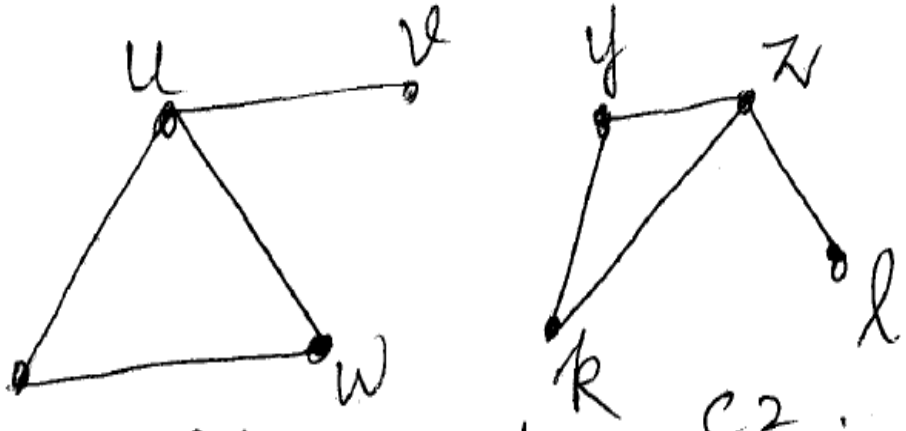
A graph that is not connected is called a disconnected graph.



G is a connected graph.

Example

G:



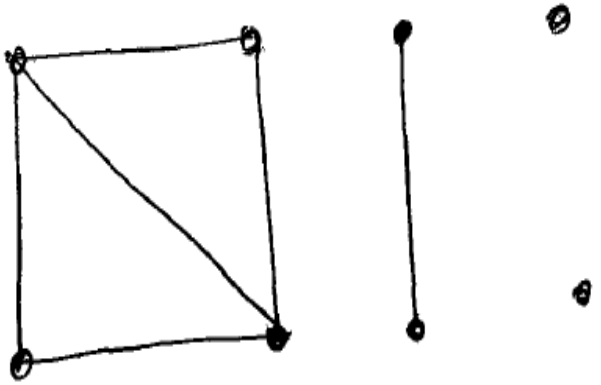
G is disconnected.

C1 and C2 are called components of G.

Components: The maximal connected subgraphs of a Disconnected graph G are called its components.

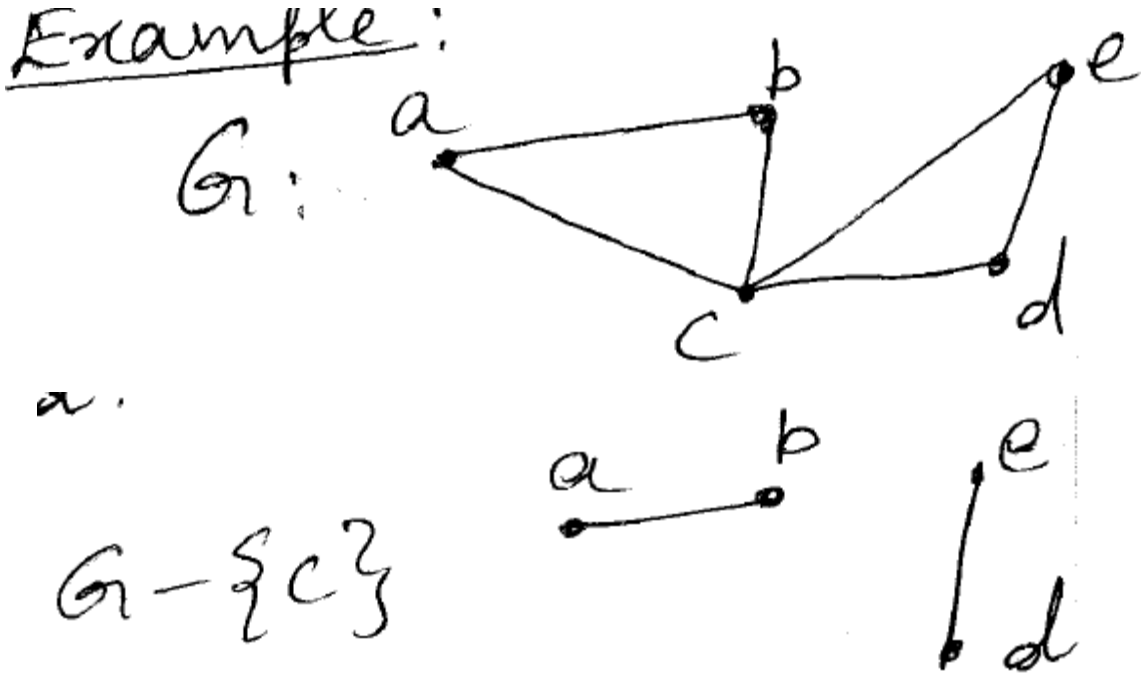
Example:

G:



G is a disconnected graph with 4 components

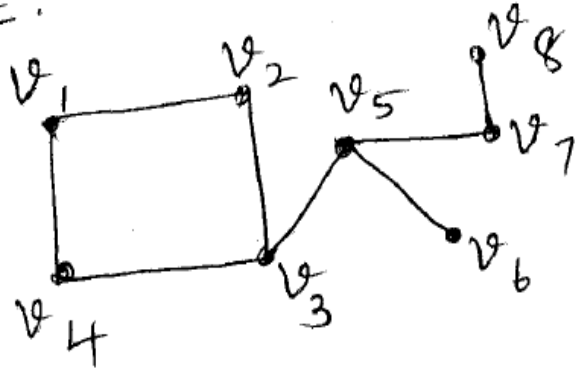
Cut vertex: A vertex v in a connected graph G is called a cut vertex if G – v is disconnected. In graph G, G-{c} is disconnected. Hence, c is a cut vertex.



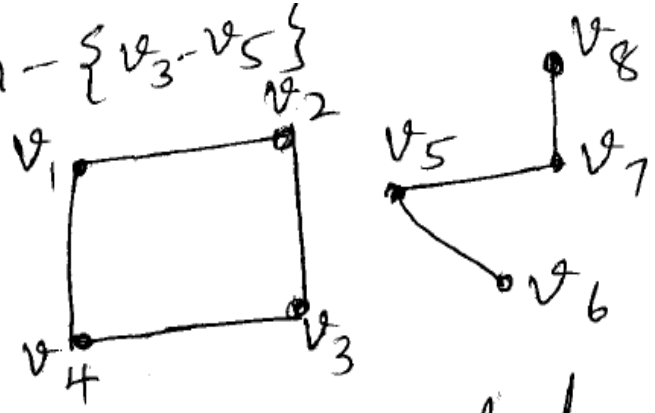
Bridge: An edge 'e' in a connected graph G is called a bridge if G – e is disconnected.

Example:

G:



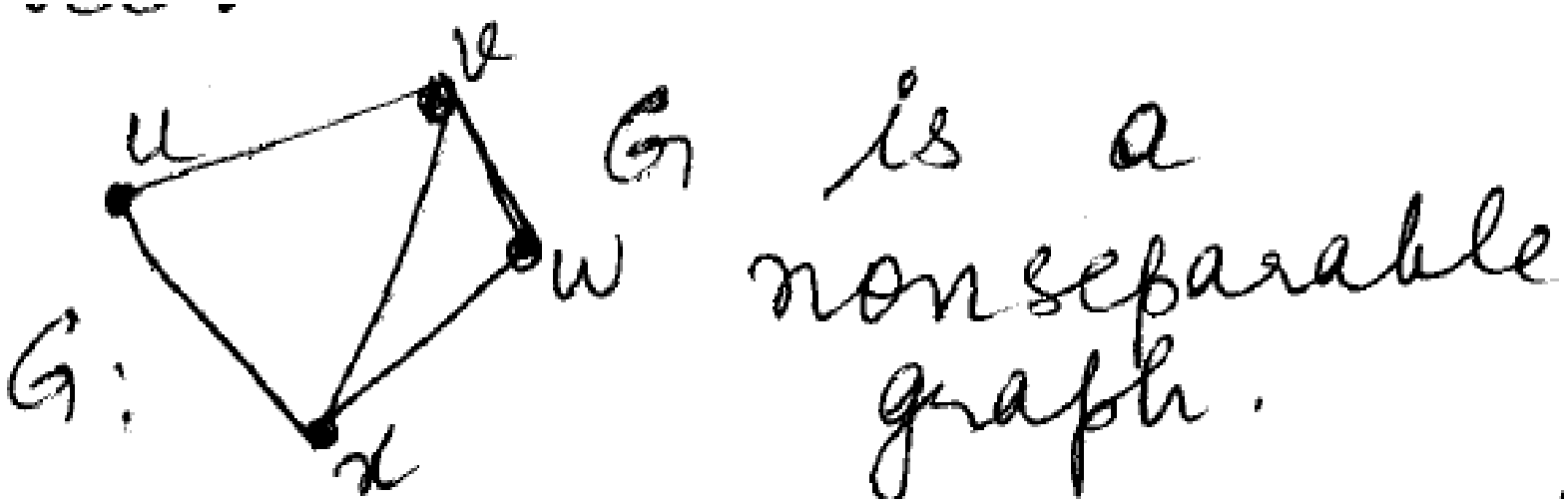
$G - \{v_3 - v_5\}$



In graph G, $G - \{v_3 - v_5\}$ is disconnected

$\therefore \{v_3 - v_5\}$ edge is a bridge.

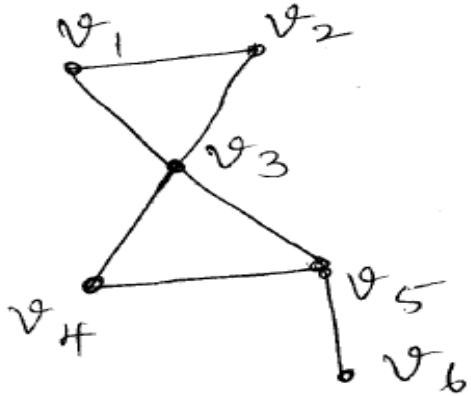
Non-separable graph: A nontrivial connected graph without a cut vertex is called a non-separable graph.



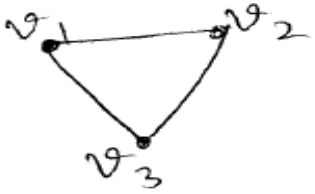
Blocks: Let G be a non trivial connected graph. A block of G is a subgraph of G that is itself a maximal non separable graph.

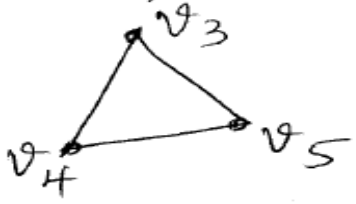
Example:

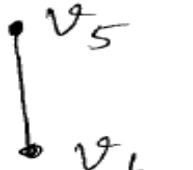
G:



Graph G has 3 blocks namely $\langle\{v_1, v_2, v_3\}\rangle$, $\langle\{v_3, v_4, v_5\}\rangle$ and $\langle\{v_5, v_6\}\rangle$







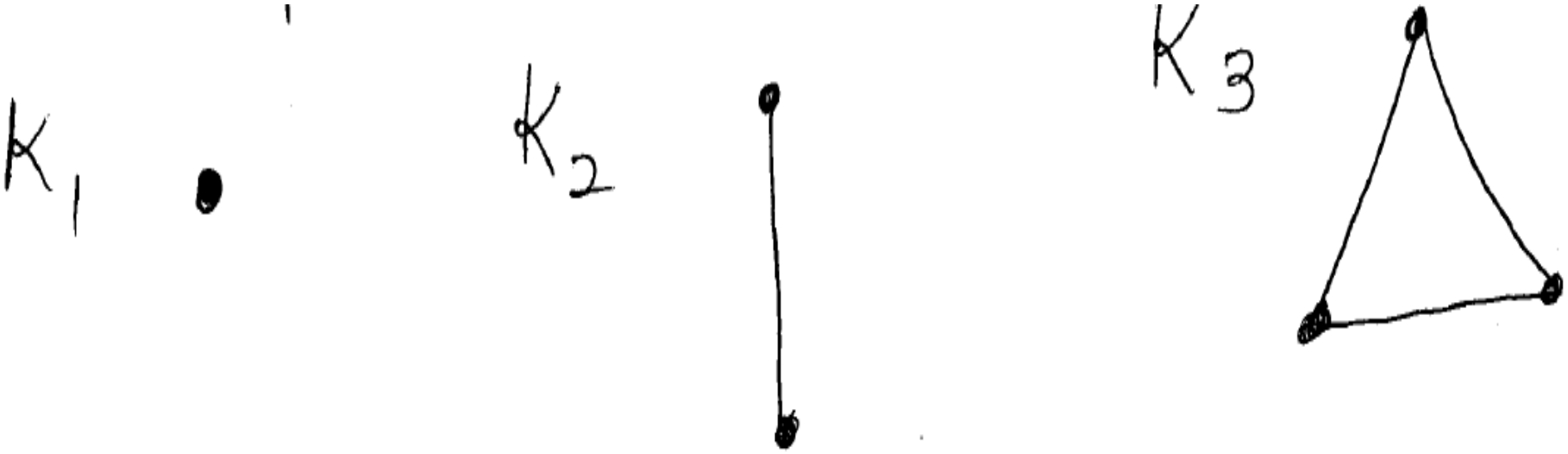
Note:

1. The blocks of a graph produce a partition of the edge set of the graph.

2. Every two blocks have at most one vertex in common.

3. If two blocks share a vertex, then the vertex is a cut vertex.

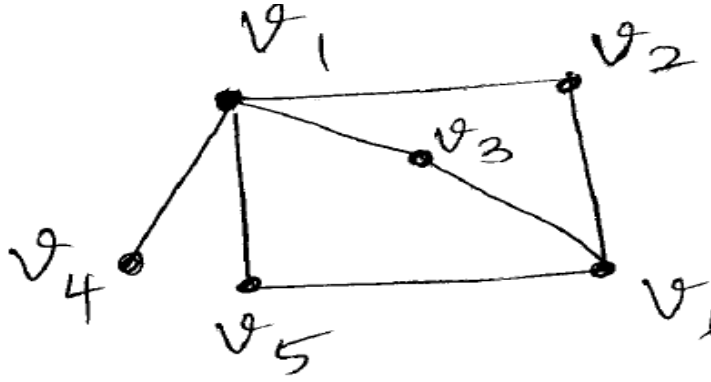
Complete graphs: A graph in which every distinct pair of vertices are adjacent is called a complete graph.
K_n denotes the complete graph on n vertices.



Bipartite graph: A graph G is called bipartite if the vertex set V(G) of G can be partitioned into two non empty subsets V_1 and V_2 such that every edge of G joins a vertex of V_1 and a vertex of V_2

Example:

G_1 :



G_1 is bipartite $V_1 = \{v_1, v_6\}$

$V_2 = \{v_2, v_3, v_4, v_5\}$

Representation of Graphs

Adjacency Matrix: Let G be a (p,q) graph with p vertices and q edges. $V(G) = \{ v_1, v_2, \ldots v_p \}$

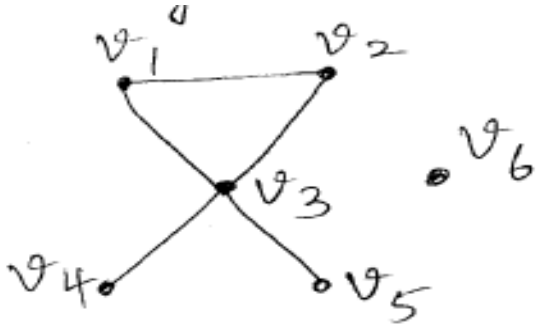
The adjacency matrix $A = [a_{ij}]$ of G is the p X p matrix defined by

$$a_{ij} = \begin{cases} 1, & \text{if } v_i v_j \in E(G) \\ 0, & \text{otherwise} \end{cases}$$

Thus A is a symmetric matrix in which every entry on the main diagonal is 0.

Example:

G :



$V(G) = \{ v_1, v_2, v_3, v_4, v_5, v_6 \}$

$E(G) = \{ v_1v_2, v_1v_3, v_2v_3, v_3v_4, v_3v_5 \}$

$A =$

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	1	0	0	0
v_2	1	0	1	0	0	0
v_3	1	1	0	1	1	0
v_4	0	0	1	0	0	0
v_5	0	0	1	0	0	0
v_6	0	0	0	0	0	0

6×6

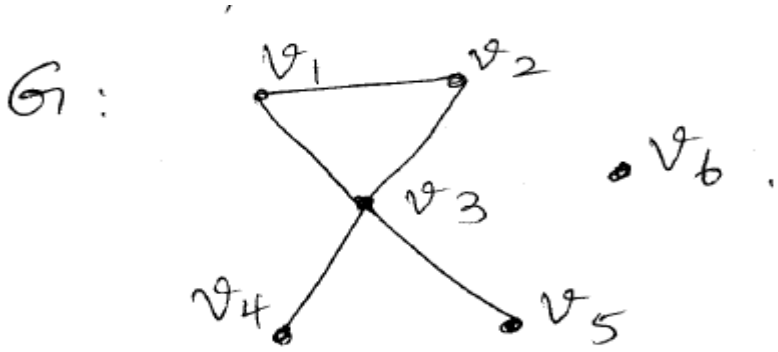
Since, A is a $p \times p$ matrix, p^2 memory locations need to be allocated for its entries.

Note:

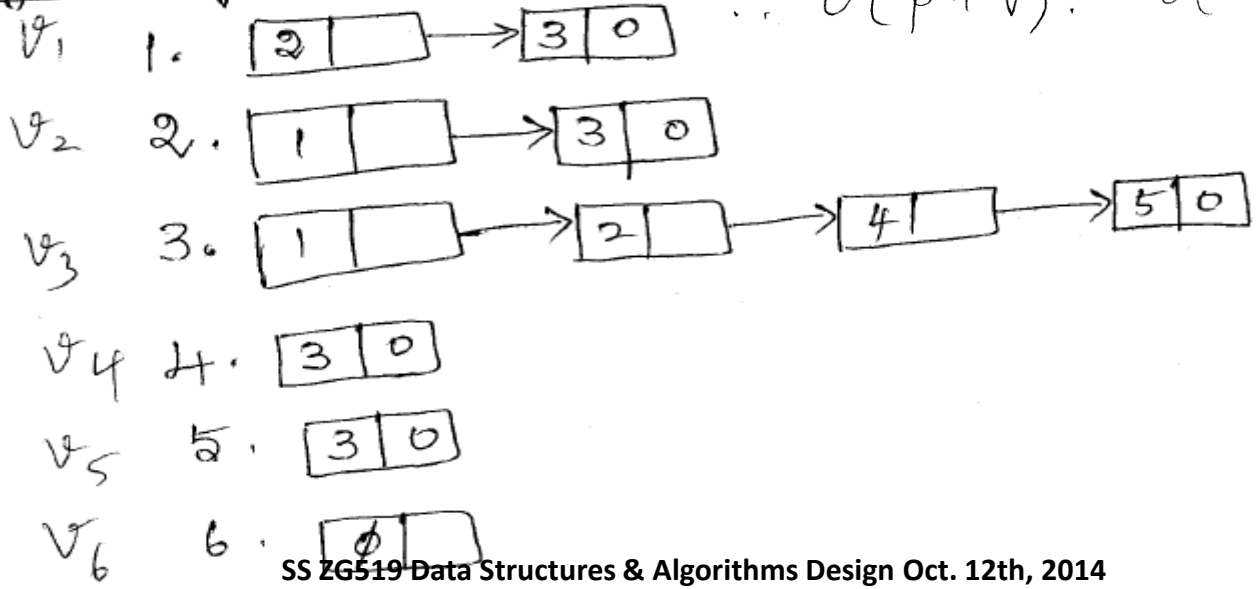
If G is a graph with relatively few edges, then many locations of its adjacency matrix contain 0. Thus, unusually Large amount of memory space is required for relatively few edges.

Adjacency Lists: Let G be a graph with vertex set $V(G) = \{ v_1, v_2, \ldots v_p \}$. The adjacency list representation of G associates with each vertex a list of its adjacent vertices.

Example:



Adjacency Lists



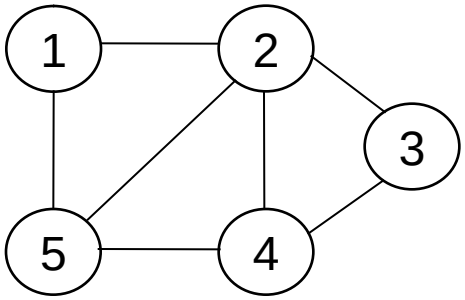
Space complexity is $p+2q$ locations

$\therefore O(p+q) = O(|V|+|E|)$

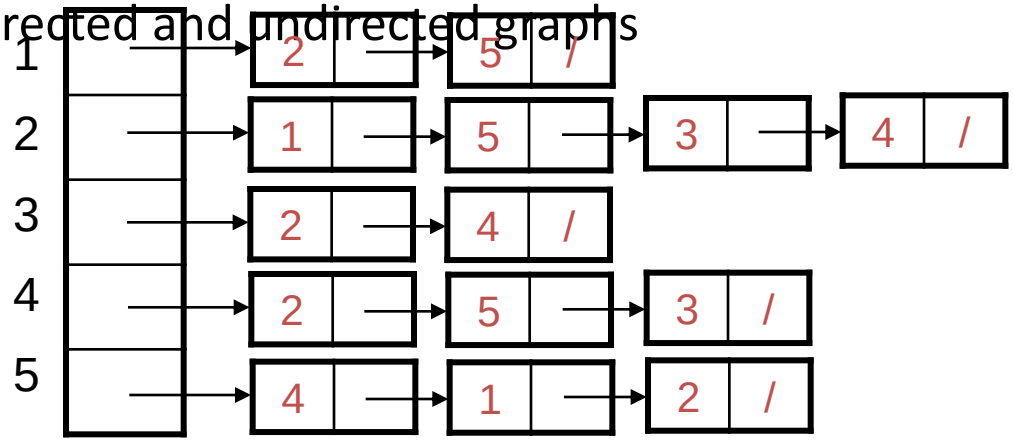
Graph Representation

Adjacency list representation of G = (V, E)

- An array of $|V|$ lists, one for each vertex in V
- Each list Adj[u] contains all the vertices v such that there is an edge between u and v
  - Adj[u] contains the vertices adjacent to u (in arbitrary order)
- Can be used for both directed and undirected graphs



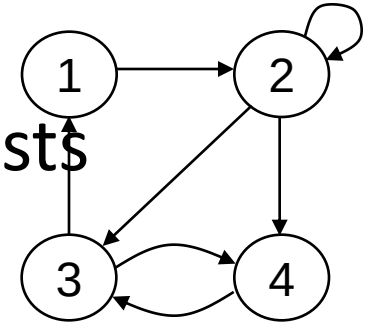
Undirected graph



Properties of Adjacency-List Representation

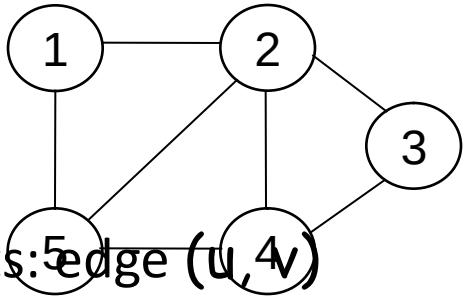
Sum of the lengths of all the adjacency lists $|E|$

- Directed graph:
  - Edge (u, v) appears only once in u's list
- Undirected graph:
  - u and v appear in each other's adjacency lists: edge (u, v) appears twice



Directed graph

$2|E|$



Undirected graph

Properties of Adjacency-List Representation

Memory required

- $\Theta(V + E)$

Preferred when

- the graph is sparse: $|E| << |V|^2$

Disadvantage

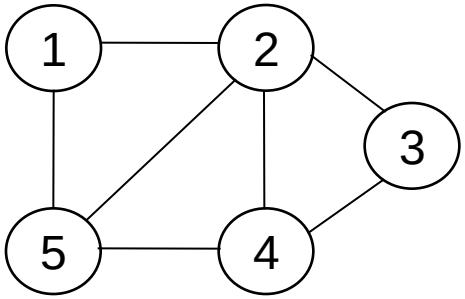
- no quick way to determine whether there is an edge between node u and v

Time to list all vertices adjacent to u:

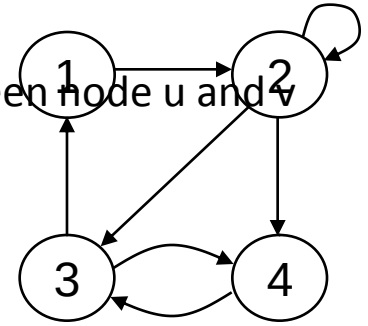
- Θ(degree(u))

Time to determine if $(u, v) \in E$:

- Θ(degree(u))



Undirected graph

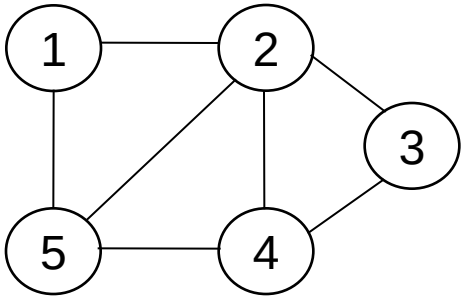


Directed graph

Graph Representation

Adjacency matrix representation of G = (V, E)

- Assume vertices are numbered 1, 2, … |V|
- The representation consists of a matrix $A_{|V| \times |V|}$:
- $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

For undirected graphs matrix A is symmetric:

$a_{ij} = a_{ji}$

$A = A^T$

Properties of Adjacency Matrix Representation

innovate achieve lead

Memory required

- $\Theta(V^2)$, independent on the number of edges in G

Preferred when

- The graph is dense $|E|$ is close to $|V|^2$
- We need to quickly determine if there is an edge between two vertices

Time to list all vertices adjacent to u:

- $\Theta(V)$

Time to determine if $(u, v) \in E$:

- $\Theta(1)$

Weighted Graphs

- **Weighted graphs** = graphs for which each edge has an associated weight w(u, v)

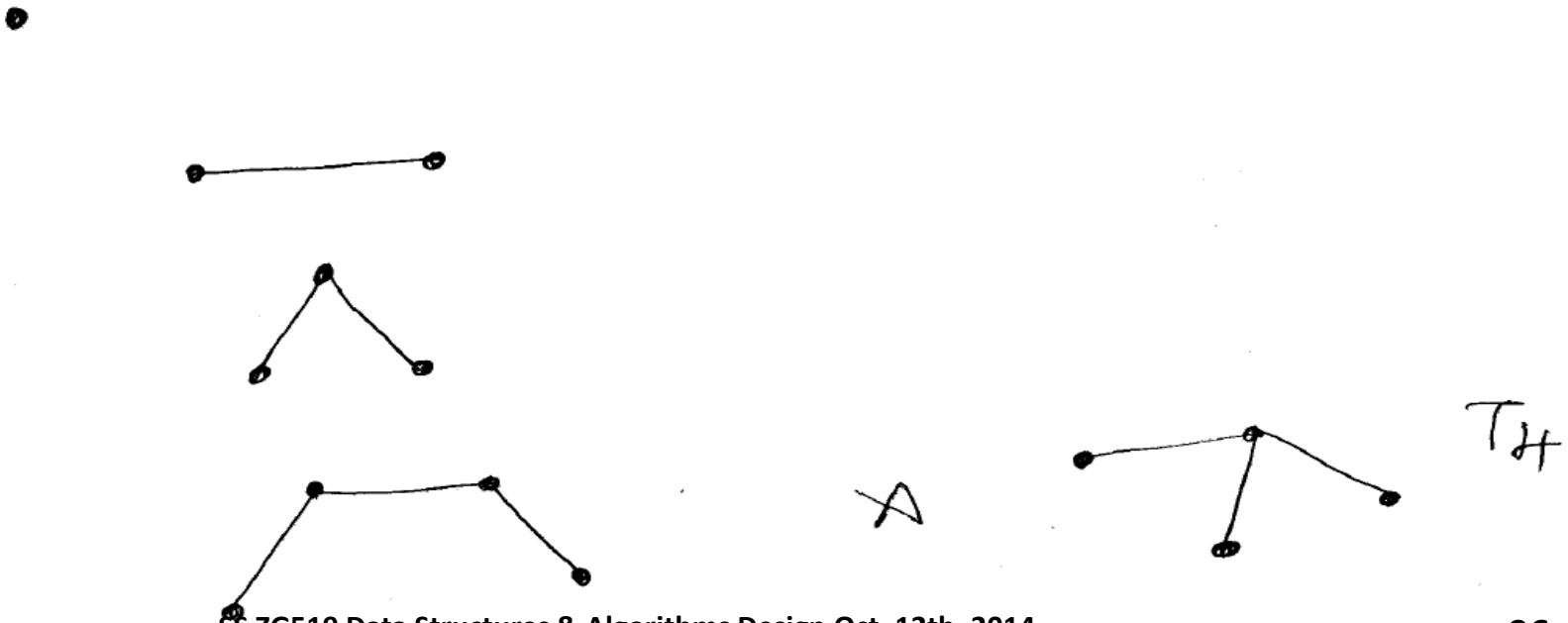
$$w: E \rightarrow R, \text{ weight function}$$

- Storing the weights of a graph
  - Adjacency list:
    - Store w(u,v) along with vertex v in u's adjacency list
  - Adjacency matrix:
    - Store w(u, v) at location (u, v) in the matrix

Trees: Tree is a connected graph without cycles.

Example:

Tree of order 1, T_1



Remarks:

1. A tree on n vertices has n-1 edges.

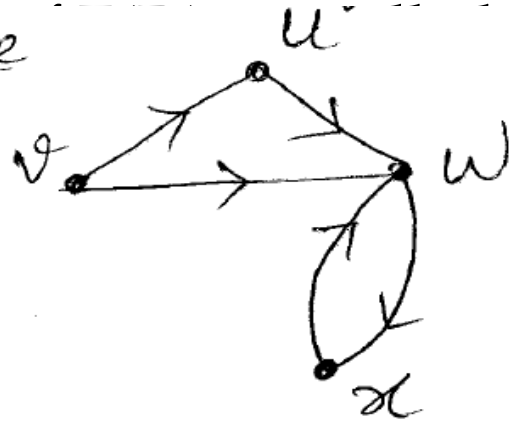
2. Every non trivial tree contains atleast two end vertices.

3. If u and v are distinct vertices of a tree T, then T contains exactly one u-v path.

Digraphs: A digraph (or directed graph) D is a finite nonempty Set V(D) of vertices and a set E(D) of ordered pairs of distinct vertices.

Example

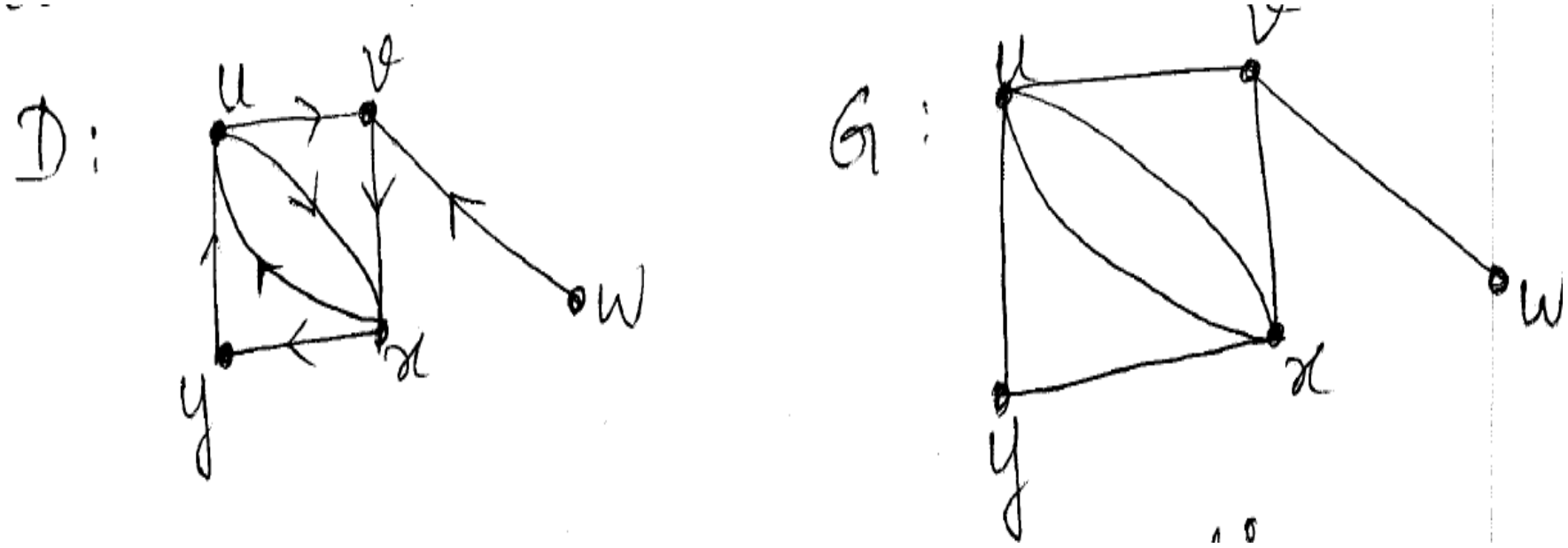
D :



$V(D) = \{u, v, w, x\}$

$E(D) = \{(u,w), (v,u), (v,w), (x,w), (w,x)\}$

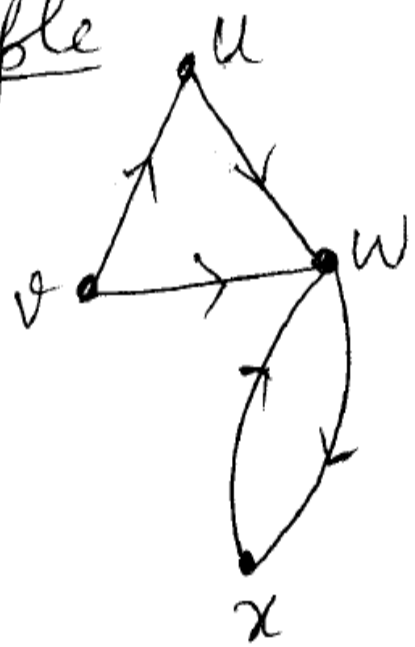
Underlying graph of a digraph D is that graph G obtained from D by replacing all arcs(u,v) or (v,u) by the edge uv.



Out degree is the number of vertices adjacent from a vertex v.

In degree of a vertex v is the number of vertices adjacent to v.

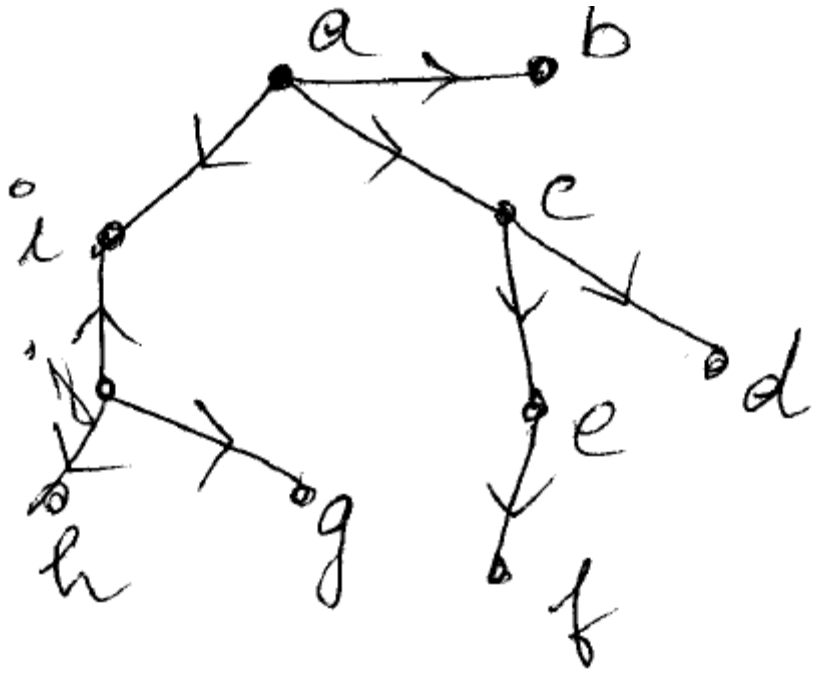
Example



vertex	outdegree	Indegree
u	1	1
v	2	0
w	1	3
x	1	1

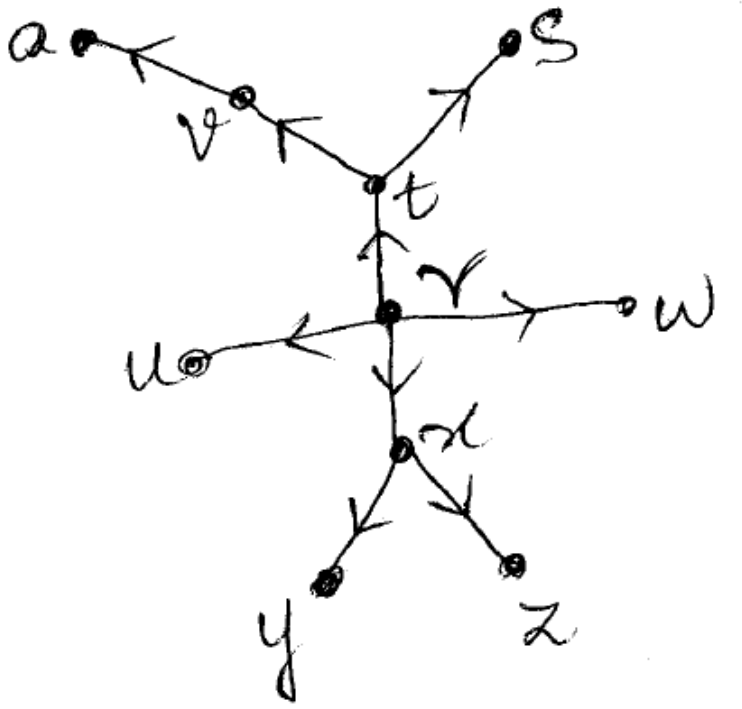
Directed Tree: A directed tree is an asymmetric digraph whose underlying graph is a tree.

Example:



Rooted Tree: A directed tree T is called a rooted tree if there exists a vertex r of T called the root such that for every vertex v of T, there is a directed r-v path in T.

Example:



Note:

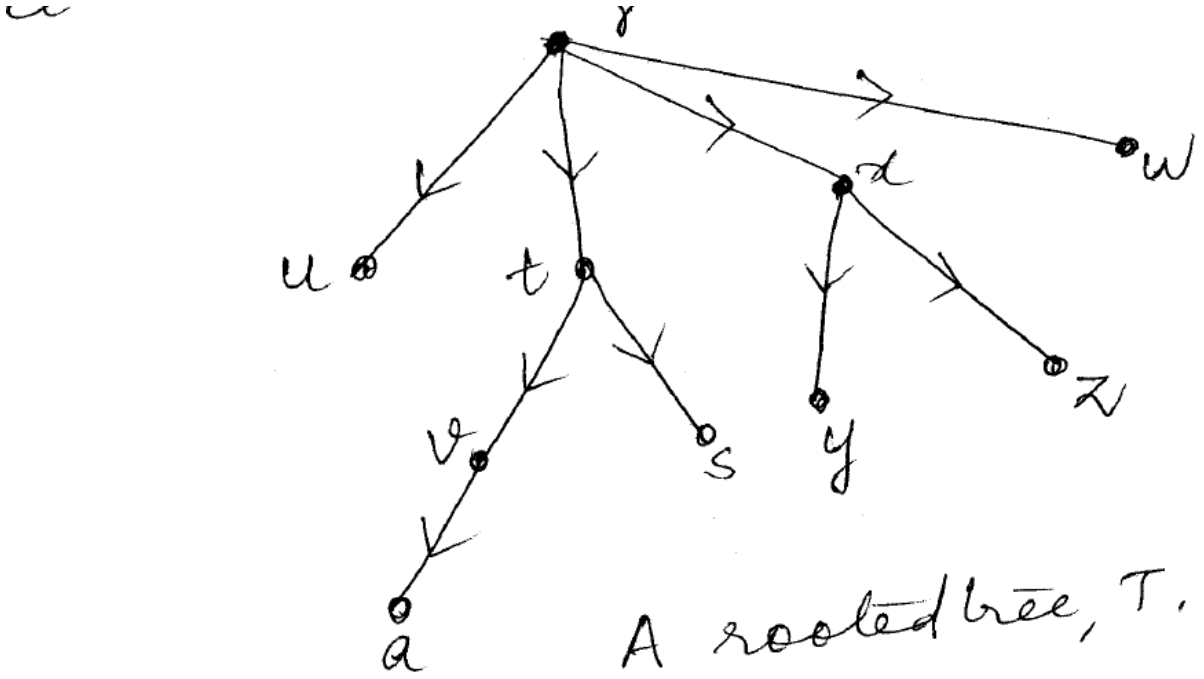
•If T is a rooted tree, then it is customary to draw T with root r at the top at level.

• The vertices adjacent from r are placed one level below at level 1.

•Any vertex adjacent from a vertex at level 1 is at level 2, and so on.

•In general, every vertex at level i > 0 is adjacent from exactly one vertex, namely one at level i-1.

Height: The largest integer h for which there is a vertex at level h in a rooted tree is called its height.



A rooted tree, T.

Height of T is 3.

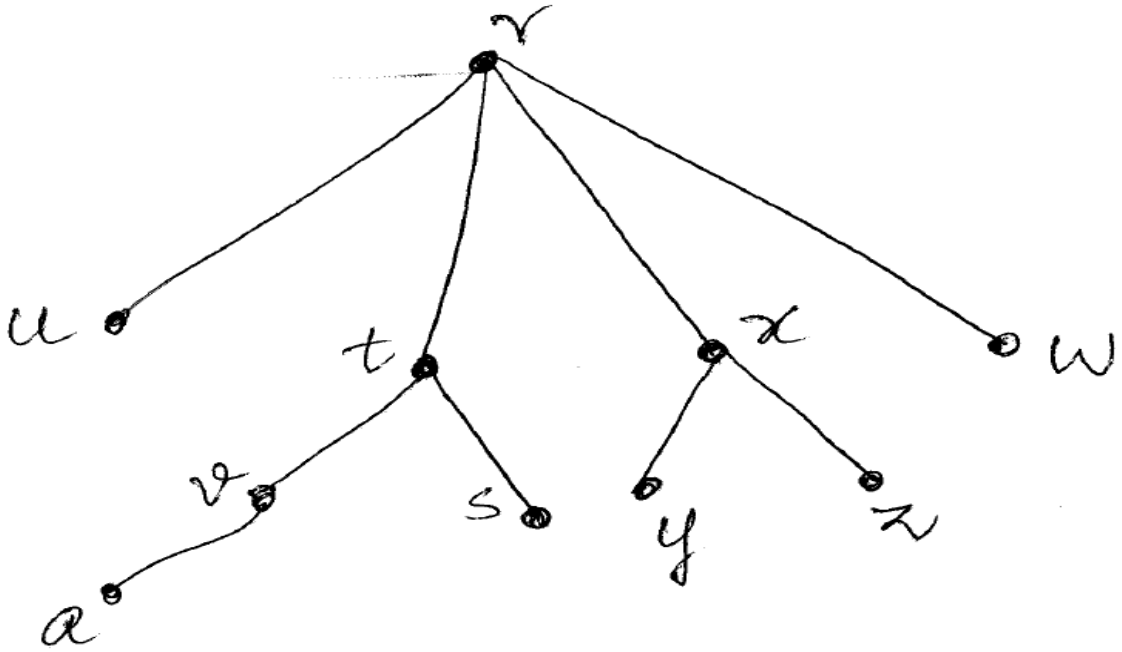
Let T be a rooted tree. If a vertex v of T is adjacent to u and u lies in the level below v, then u is called a **child** of v and v is the **parent** of u.

A vertex w is a **descendant** of v and v is an **ancestor** of w if the v-w path in T lies below v.

The vertex z is a child of x, and x is the parent of both y and z. a is a descendant of t since the t-a path t,v,a in T lies below t. But y is not a descendant of t since the t-y path t,r,x,y inT contains vertices that are not below t.

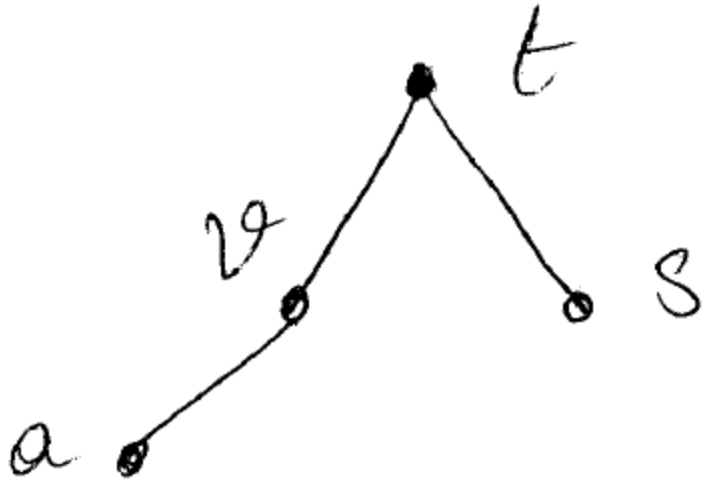
Example:

T:



Maximal Subtree: The subtree of a rooted tree T induced by a vertex v and all of its descendants is also a rooted tree with root v. This subtree is called the maximal subtree of T rooted at v.

Example



From the tree T in the previous example, this is a maximal subtree rooted at t.

Note: In a rooted tree , only the root has no parent, while every other vertex has exactly one parent.

Leaf: A vertex with no children is called a leaf; all other vertices are called **internal vertices.**

m-ary tree: A rooted tree is called m-ary if every vertex has atmost m children.

A **binary tree** is a 2-ary tree in which each child is designated as a left chid or a right child.

A rooted tree T is called a **complete m-ary tree** if every vertex of T has m children or no children.

Thus in a **complete binary tree**, every vertex has two children or no children.

A rooted tree of height h is **balanced** if every leaf is at level h or level h – 1.