GATE ONLINE CLASSES
ON
DATA STRUCTURES

Presented by

Chandra Sekhar Sanaboina

Assistant Professor

Department of CSE

University College of Engineering Kakinada

Jawaharlal Nehru Technological University Kakinada

Day – 10 Lecture Notes

on

DATA STRUCTURES

GATE ONLINE CLASSES

GRAPH REPRESENTATIONS

Graph Representations

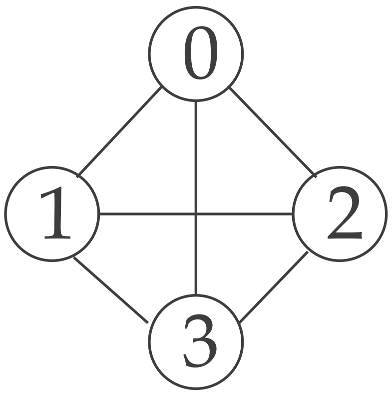
- Adjacency Matrix
- Adjacency Lists

ADJACENCY MATRIX

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric
- The adjacency matrix for a digraph need not be symmetric

Examples on Adjacency Matrix



G

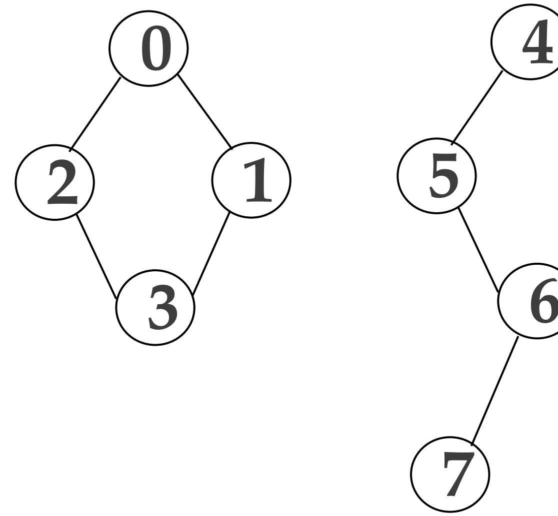
1

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_2

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Symmetric



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For undirected graph, the degree of a vertex is sum of columns
- For a digraph, the row sum is the out-degree, while the column sum is the in-degree

$$outd(vi) = \sum_{j=0}^{n-1} A[i, j]$$

$$ind(vi) = \sum_{j=0}^{n-1} A[j, i]$$

FEATURES OF ADJACENCY MATRIX

- Storage complexity: $O(|V|^2)$

- Undirected graph: symmetric along main diagonal
  - A^T transpose of A
  - Undirected: $A=A^T$

- Directed Graph:
  - In-degree of X: Sum along column X $O(|V|)$
  - Out-degree of X: Sum along row X $O(|V|)$

- Very simple, good for small graphs

DEMERITS OF ADJACENCY MATRIX

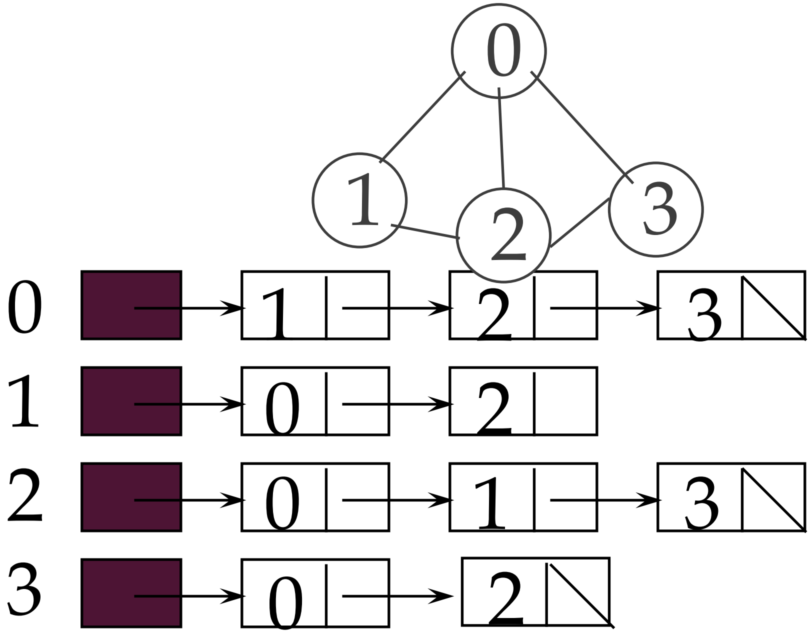
- Many graphs in practical problems are sparse
  - Not many edges --- not all pairs x,y have edge x→y

- Matrix representation demands too much memory

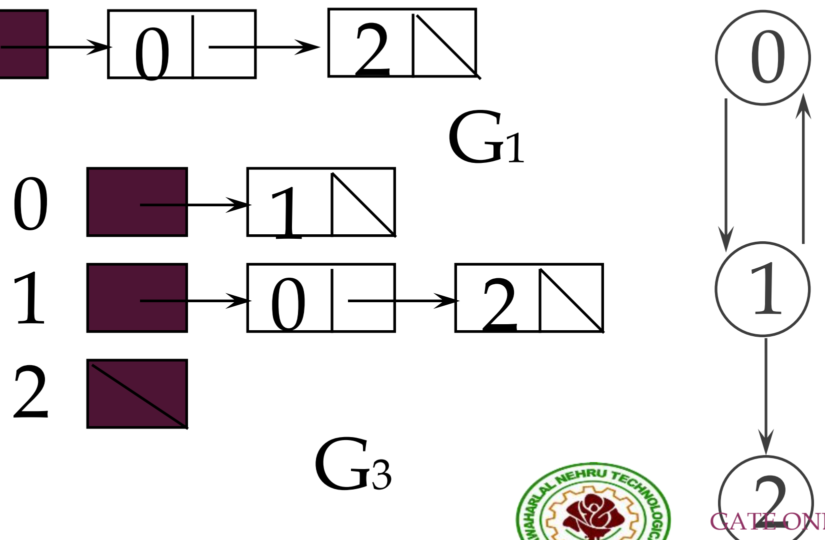
- We want to reduce memory footprint
  - Use sparse matrix techniques

ADJACENCY LISTS

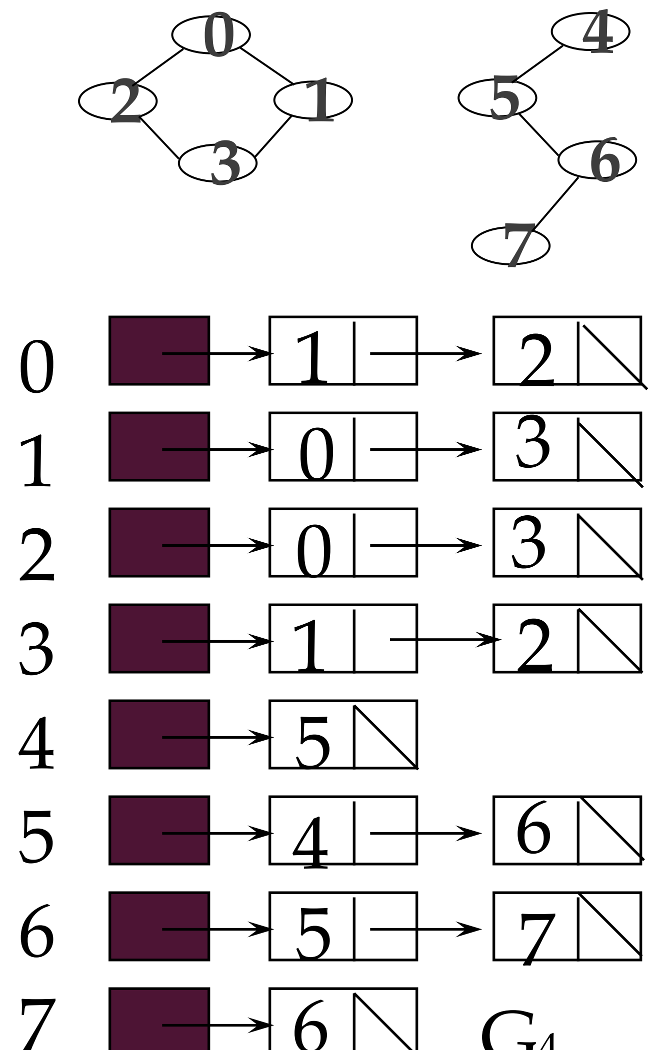
Adjacency List



G_1



G_3



G_4

An undirected graph with n vertices and e edges ==> n head nodes and 2e list nodes

INTERESTING OPERATIONS

- degree of a vertex in an undirected graph
  - –# of nodes in adjacency list
- out-degree of a vertex in a directed graph
  - –# of nodes in its adjacency list
- in-degree of a vertex in a directed graph
  - –traverse the whole data structure

FEATURES OF ADJACENCY LIST FEATURES

- Storage Complexity:
  - $O(|V| + |E|)$
  - In undirected graph: $O(|V|+2*|E|) = O(|V|+|E|)$

- Degree of node X:
  - Out degree: Length of Adj[X] $O(|V|)$ calculation
  - In degree: Check all Adj[] lists $O(|V|+|E|)$

GRAPH TRAVERSALS

Graph Traversal / Grah Searching Techniques

- A traversal (search):
  - An algorithm for systematically exploring a graph
  - Visiting (all) vertices
  - Until finding a goal vertex or until no more vertices

- Two types of Graph Traversal Techniques
  - Depth First Search (DFS)
    preorder tree traversal
  - Breadth First Search (BFS)
    level order tree traversal

BREADTH FIRST SEARCH

BREADTH-FIRST SEARCH

- One of the simplest algorithms
- Also one of the most important
  - It forms the basis for MANY graph algorithms

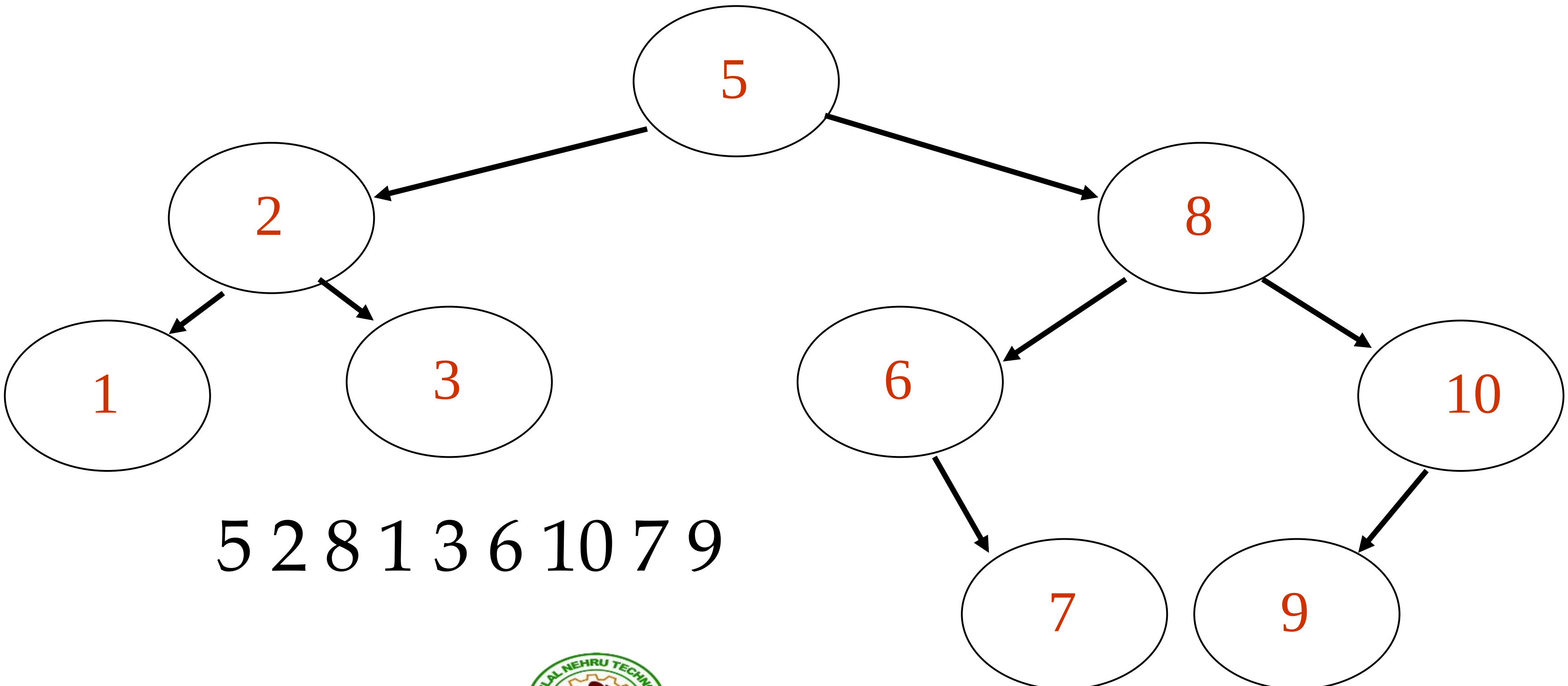
BFS: LEVEL-BY-LEVEL TRAVERSAL

- Given a starting vertex s
- Visit all vertices at increasing distance from s
  - Visit all vertices at distance k from s
  - Then visit all vertices at distance k+1 from s
  - Then ….

APPLICATIONS OF BREADTH FIRST TRAVERSAL

- Shortest Path and Minimum Spanning Tree for unweighted graph
- Peer to Peer Networks
- Crawlers in Search Engines
- Social Networking Websites
- GPS Navigation systems
- Broadcasting in Network
- In Garbage Collection
- Cycle detection in undirected graph
- To test if a graph is Bipartite
- Path Finding
- Finding all nodes within one connected component
- Prim's Minimum Spanning Tree
- Dijkstra's Single Source Shortest Path

BFS IN A BINARY TREE

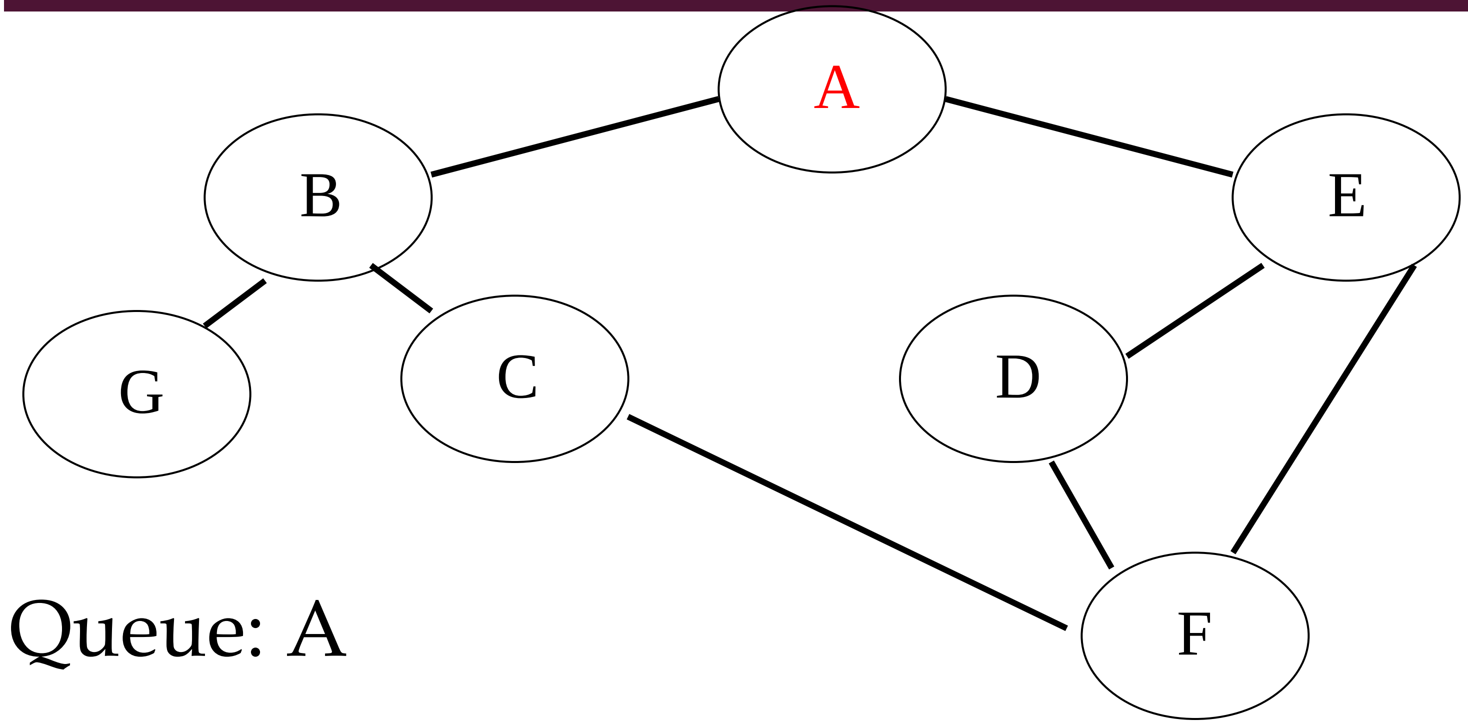


5 2 8 1 3 6 10 7 9

ALGORITHM FOR BFS USING QUEUES

1. unmark all vertices in G
2. q ← new queue
3. mark s
4. enqueue(q, s)
5. while (not empty(q))
6. curr ← dequeue(q)
7. visit curr // e.g., print its data
8. for each edge <curr, V>
9. if V is unmarked
10. mark V
11. enqueue(q, V)

BFS Using Queues

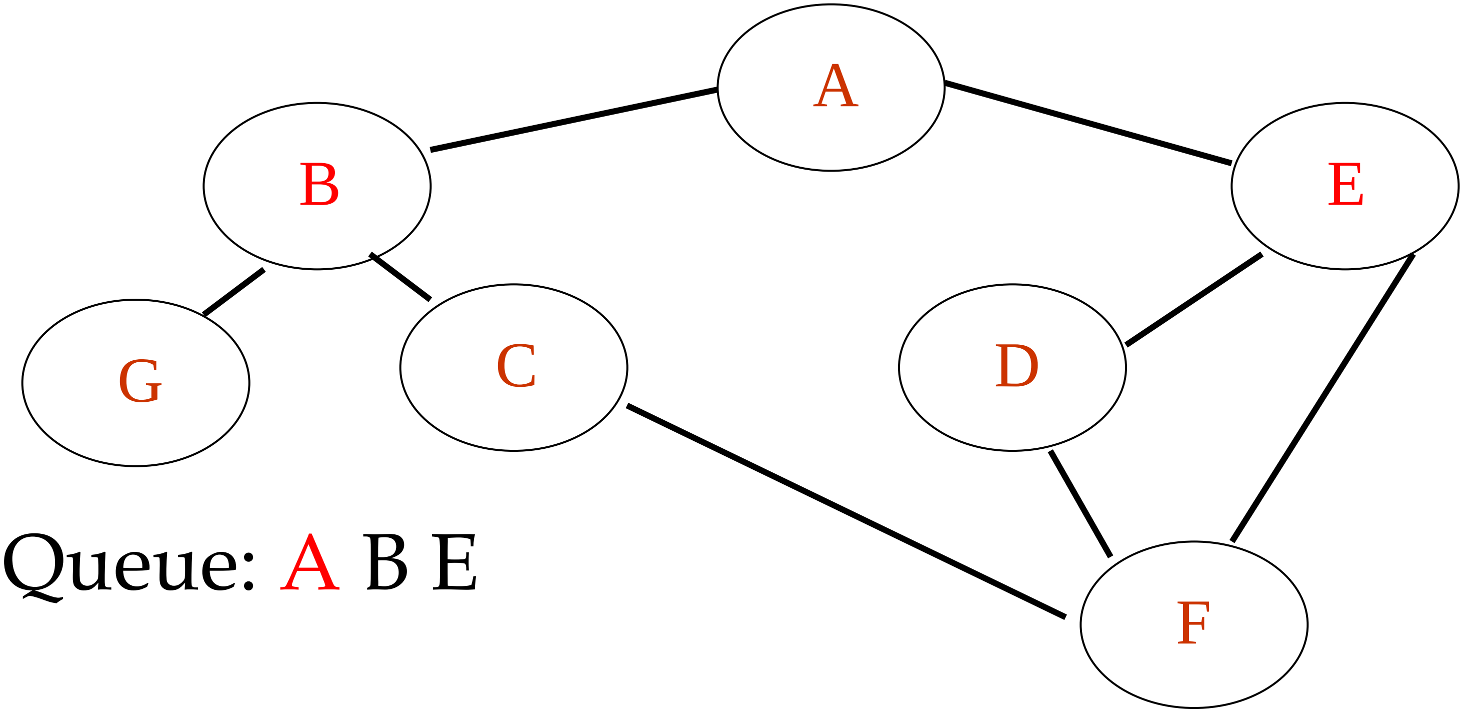


Queue: A

Start with A. Put in the queue (marked red)

GATE ONLINE CLASSES

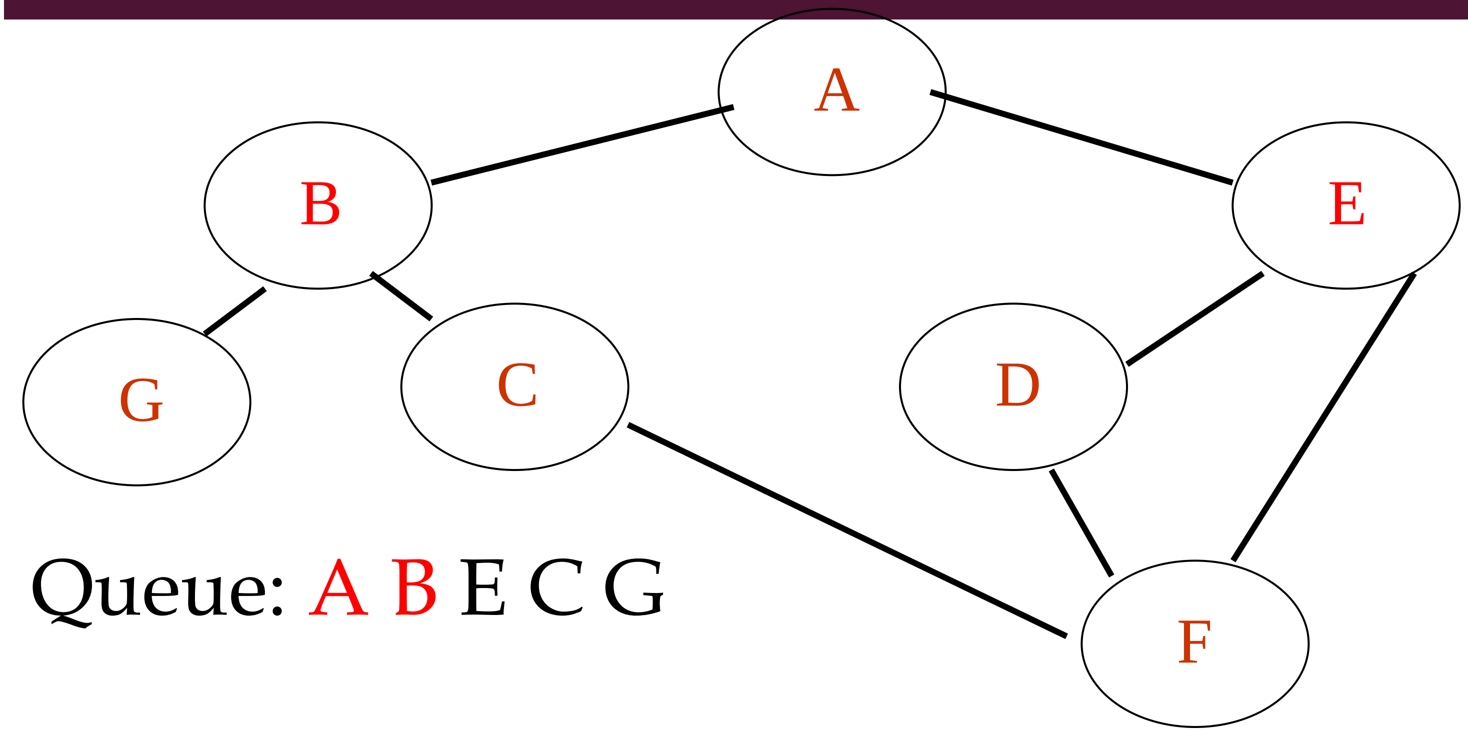
BFS Using Queues



Queue: A B E

When we go to A, we put B and E in the queue

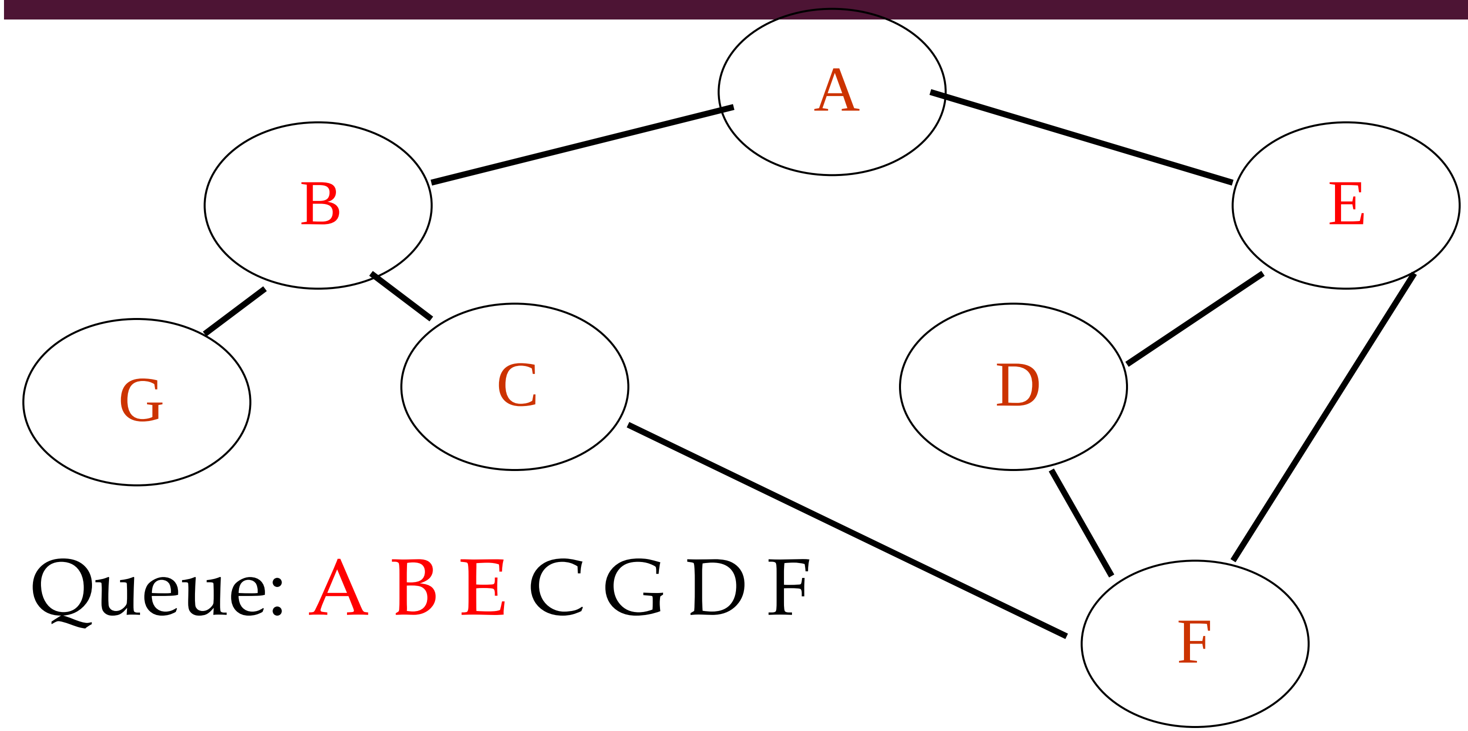
BFS Using Queues



Queue: A B E C G

When we go to B, we put G and C in the queue

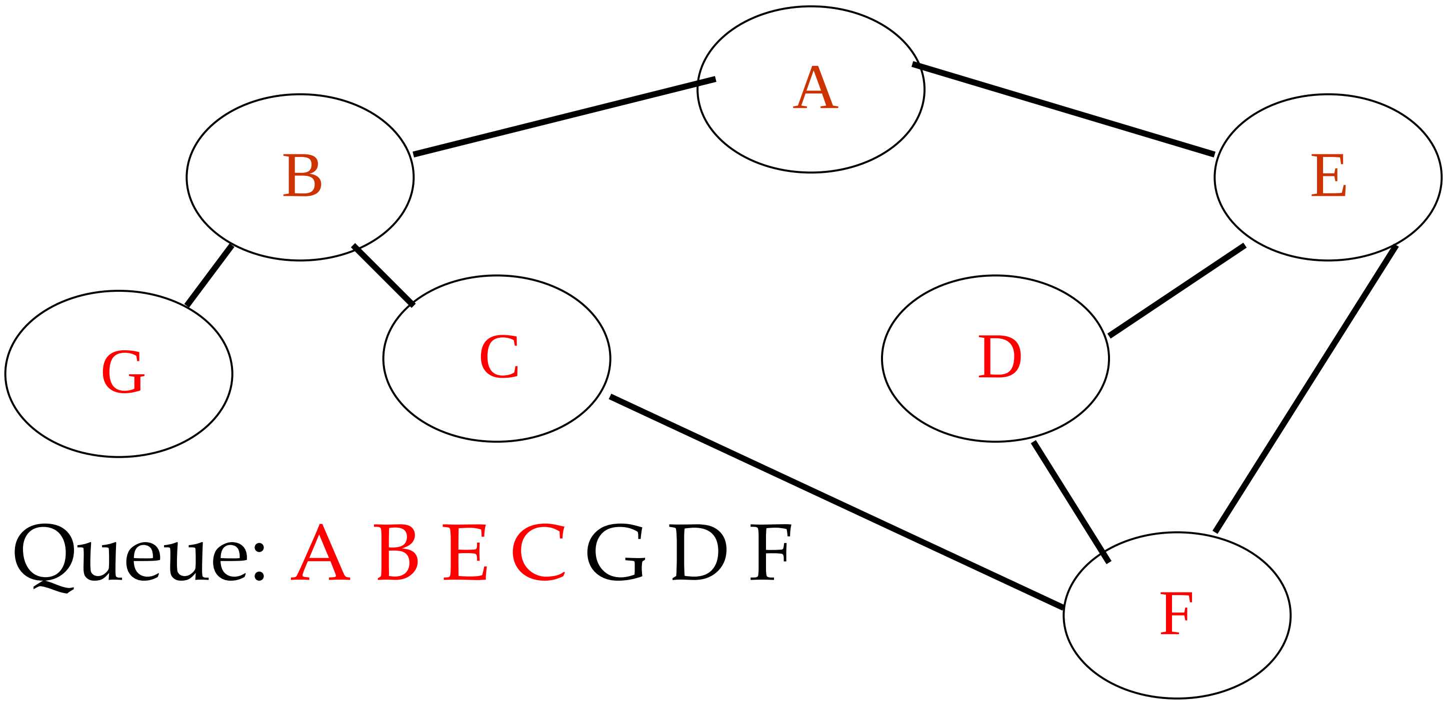
BFS Using Queues



Queue: A B E C G D F

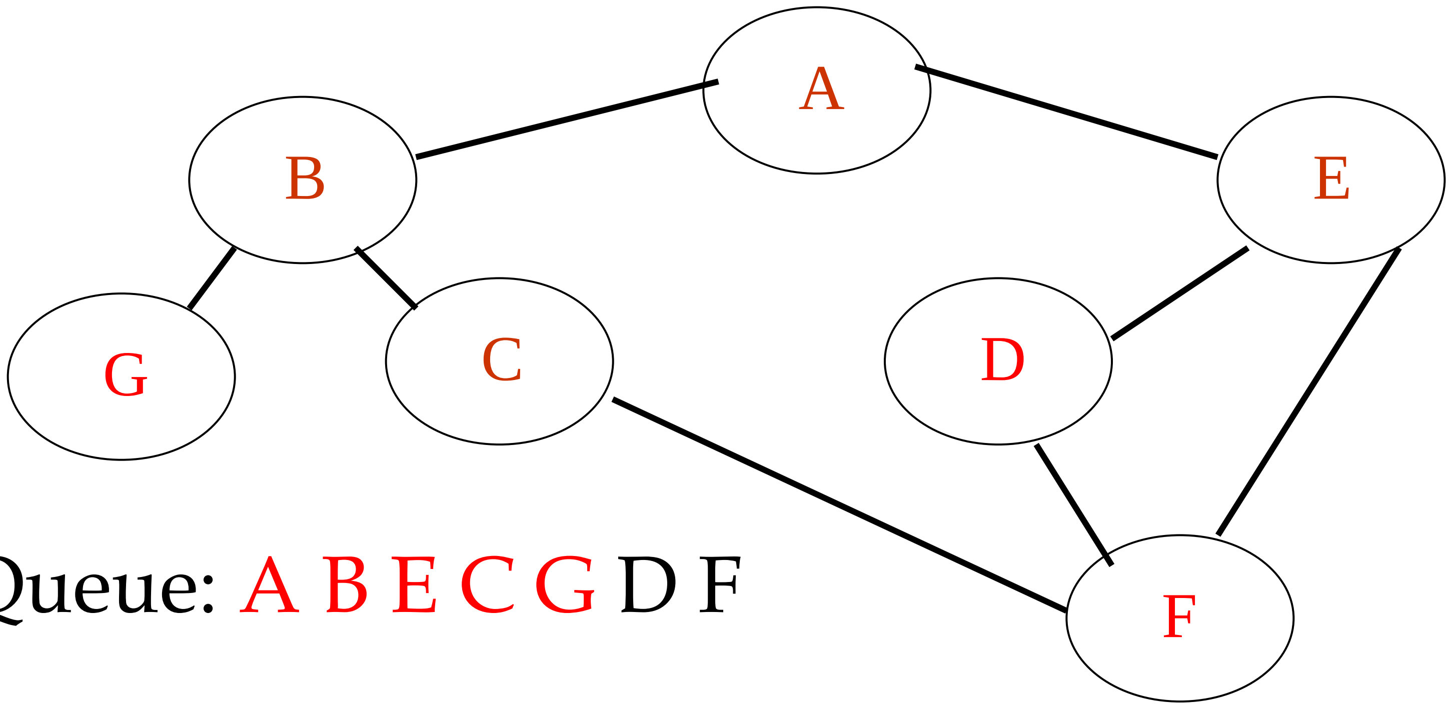
When we go to E, we put D and F in the queue

BFS Using Queues



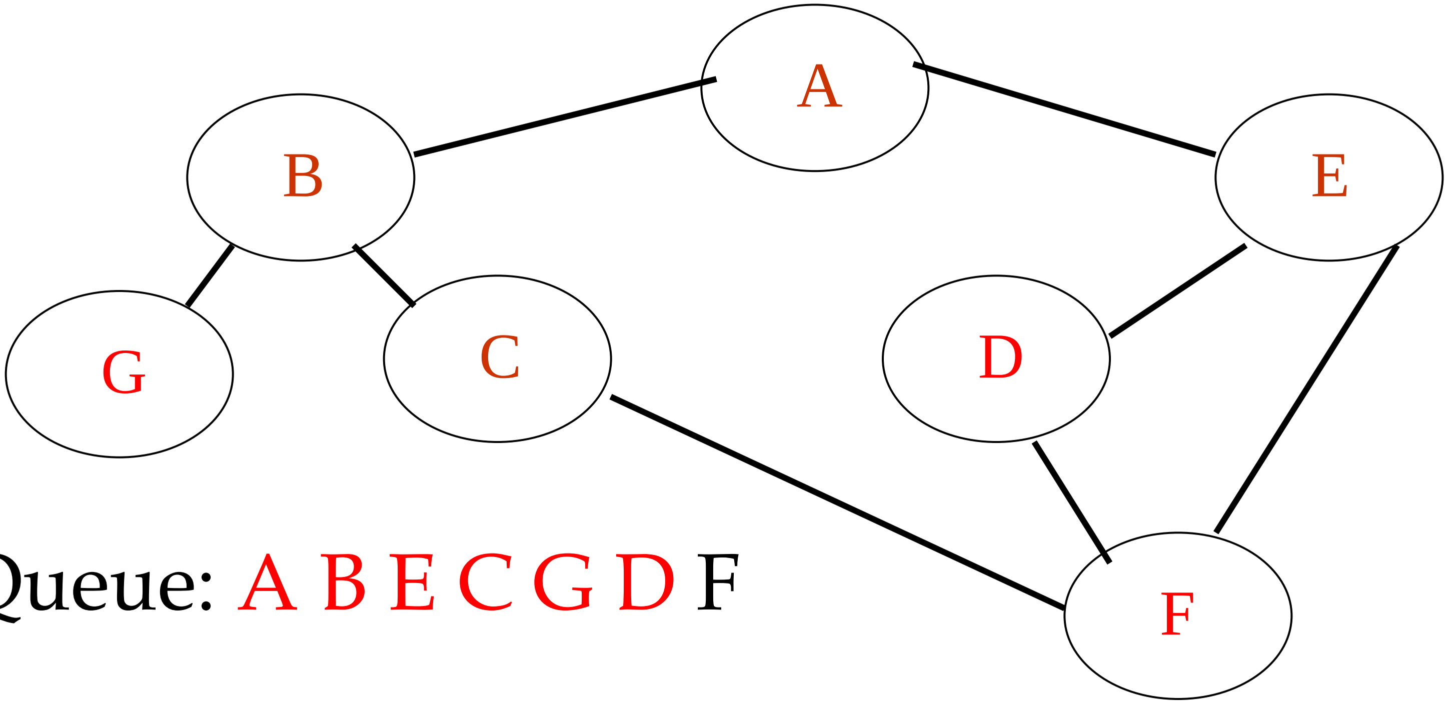
Queue: A B E C G D F

BFS Using Queues



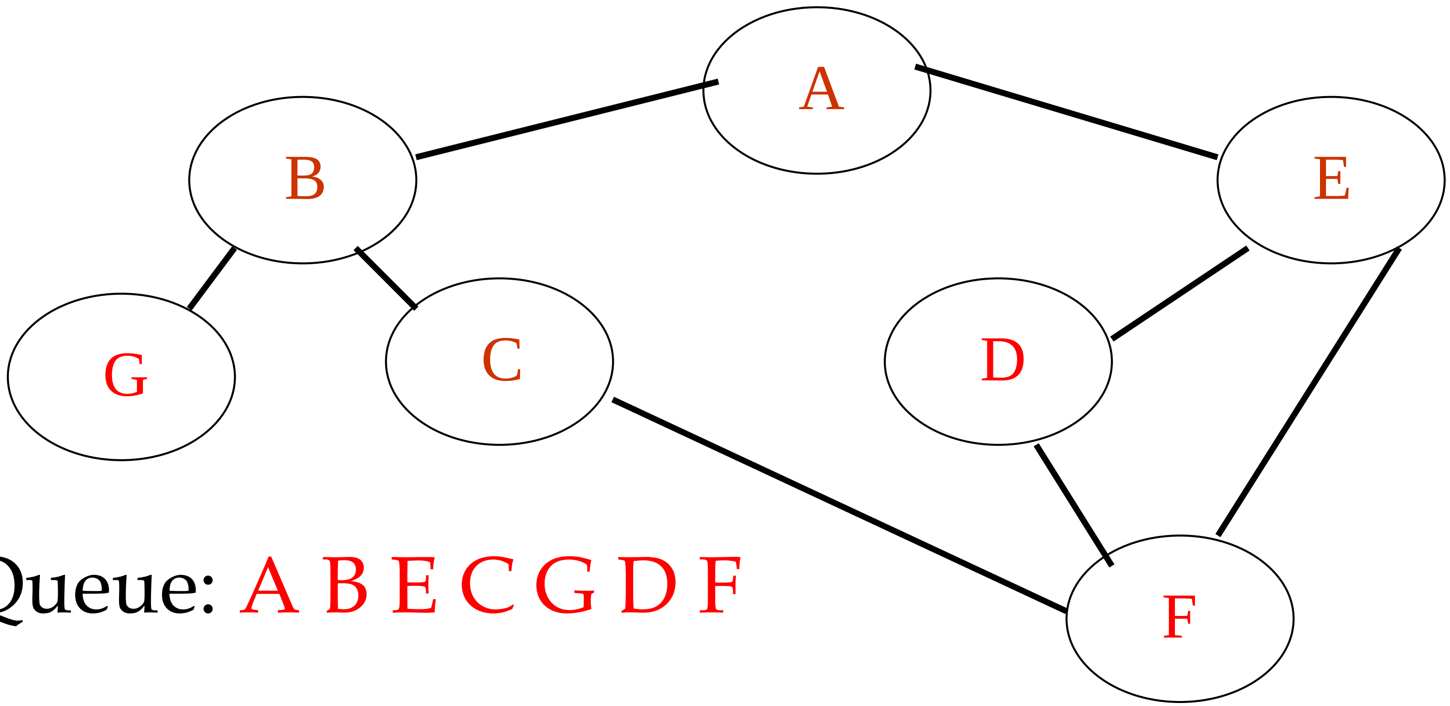
Queue: A B E C G D F

BFS Using Queues



Queue: A B E C G D F

BFS Using Queues



Queue: A B E C G D F

FEATURES OF BFS

- Complexity: $O(|V| + |E|)$
  - All vertices put on queue exactly once
  - For each vertex on queue, we expand its edges
  - In other words, we traverse all edges once

- BFS finds shortest path from s to each vertex
  - Shortest in terms of number of edges

DEPTH FIRST SEARCH

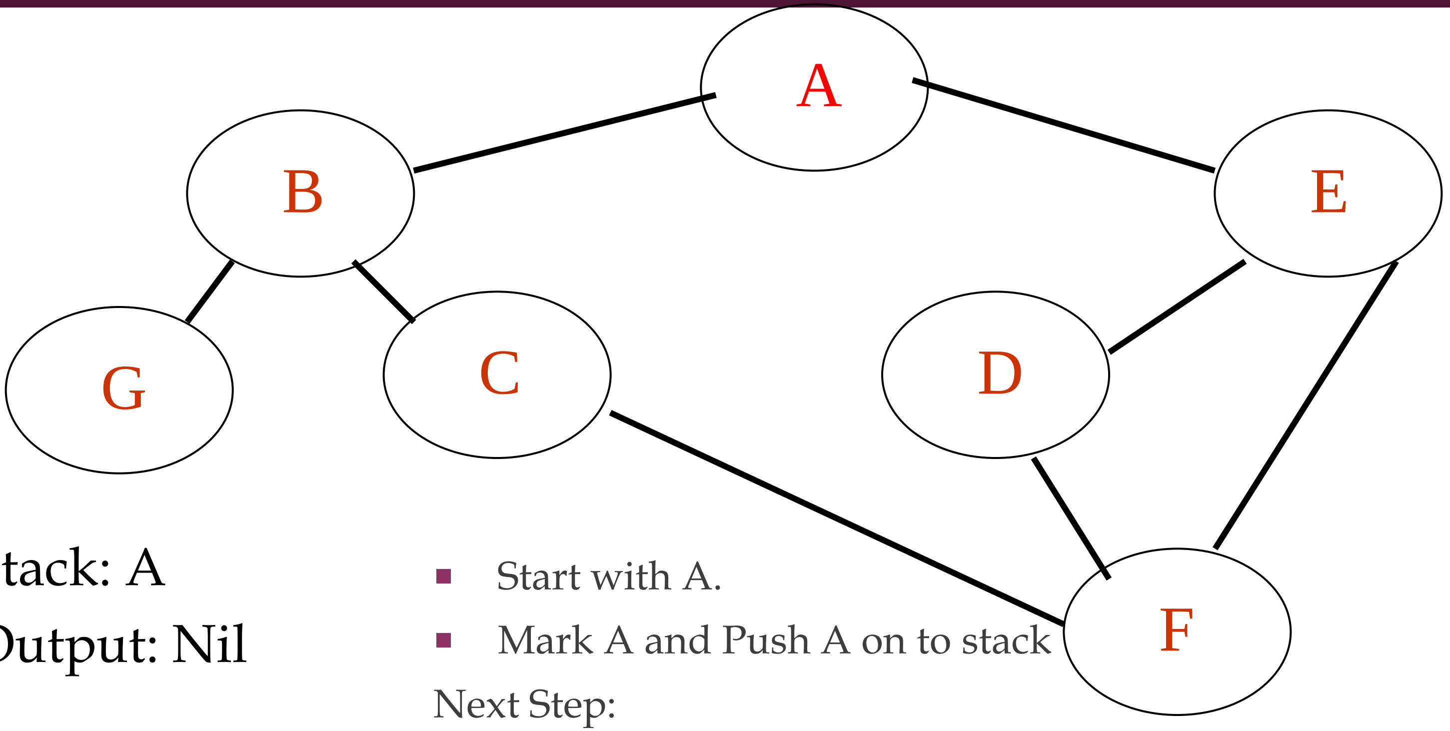
DEPTH-FIRST SEARCH

- Again, a simple and powerful algorithm
- Given a starting vertex s
- Pick an adjacent vertex, visit it.
  - Then visit one of its adjacent vertices
  - …..
  - Until impossible, then backtrack, visit another

APPLICATIONS OF DEPTH FIRST SEARCH

- Minimum spanning tree for undirected graphs
- Detecting cycle in a graph
- Path Finding
- Topological Sorting
- To test if a graph is bipartite
- Finding Strongly Connected Components of a graph
- Solving puzzles with only one solution

DFS Using Stacks



Stack: A

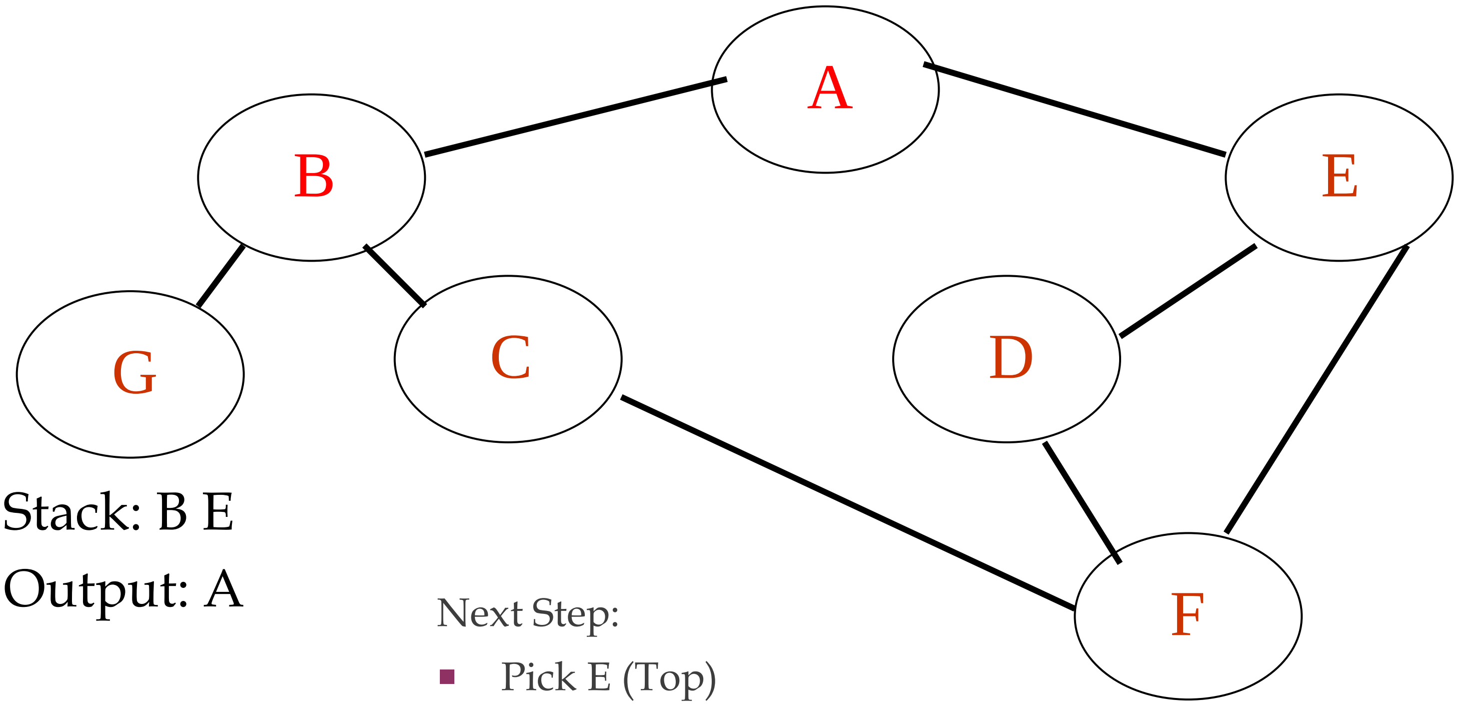
Output: Nil

- Start with A.
- Mark A and Push A on to stack

Next Step:

- Pop A and output
- Push the Adjacent Vertices of A (B, E) into Stack

DFS Using Stacks



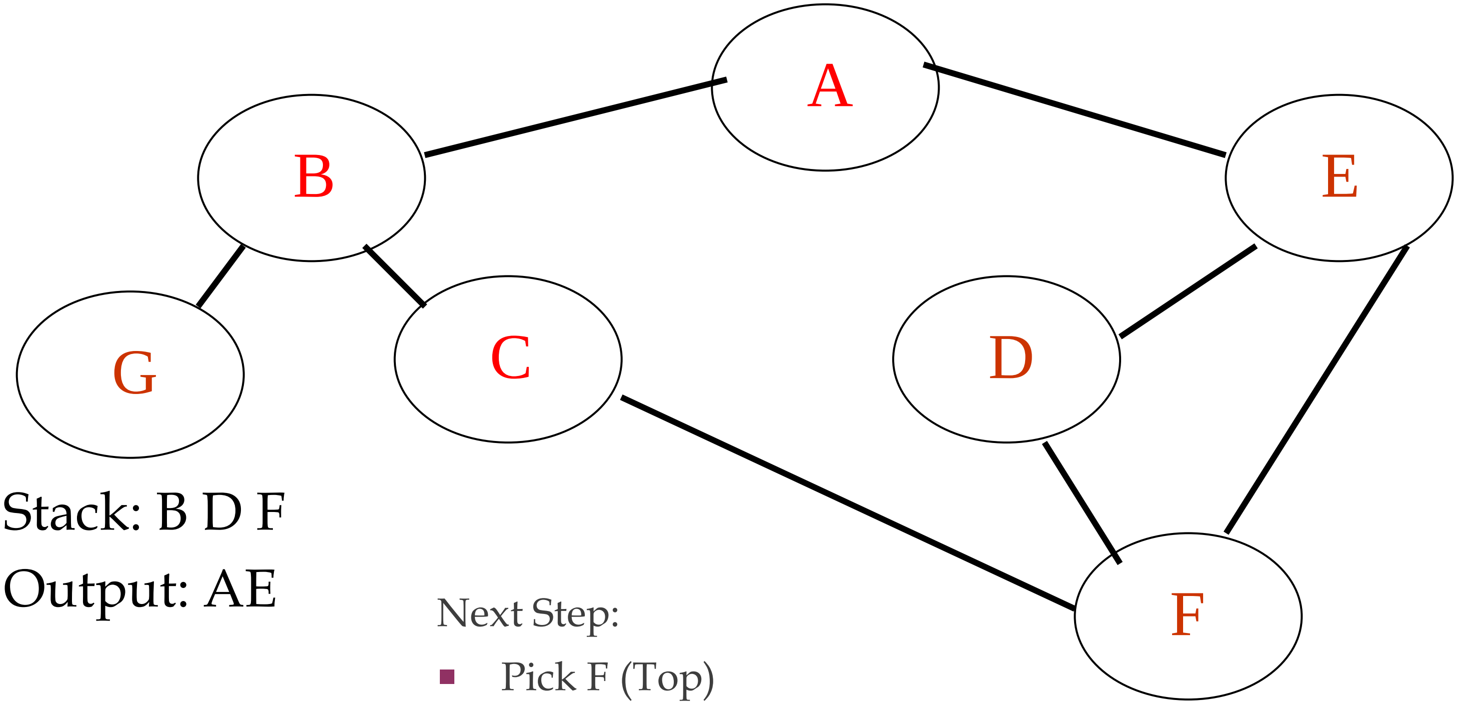
Stack: B E

Output: A

Next Step:

- Pick E (Top)
- Pop E and place it in output by pushing adjacent vertices of E (D, F) on to the stack

DFS Using Stacks



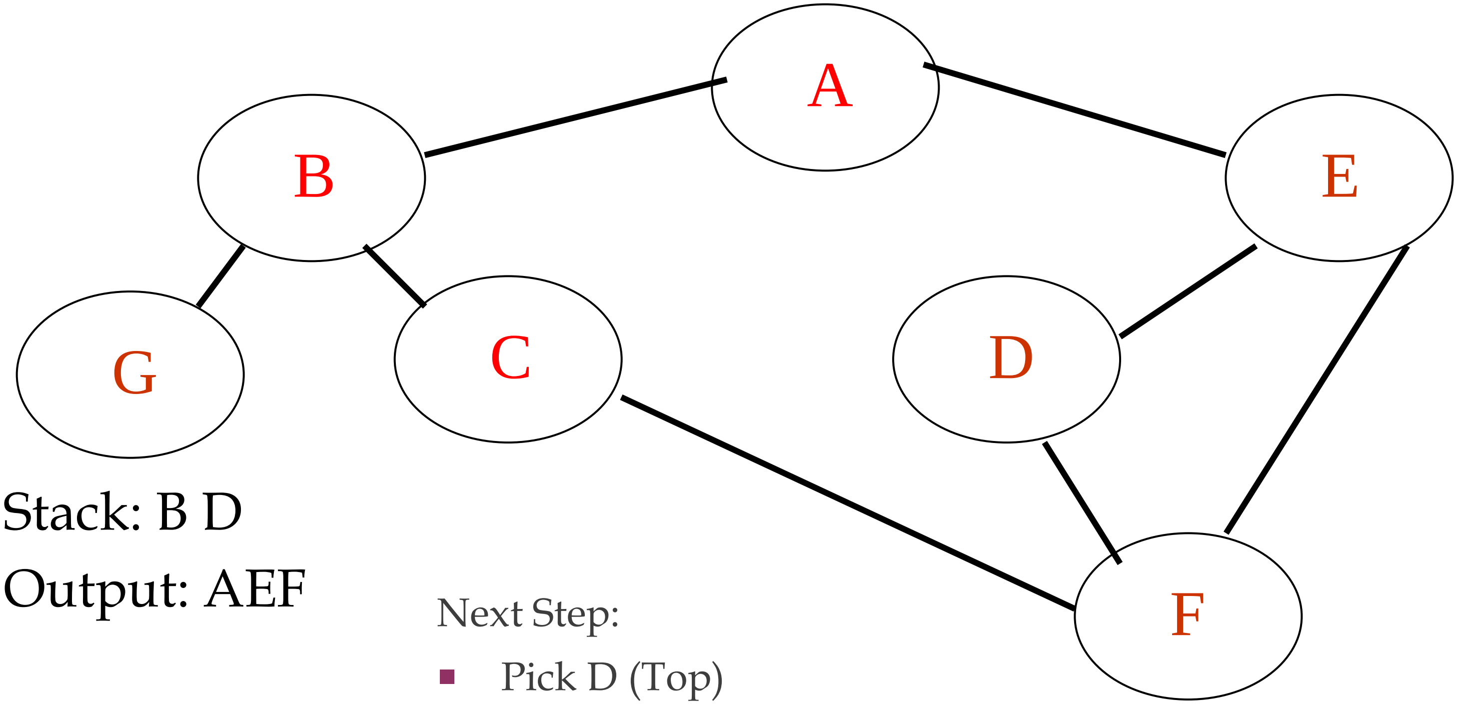
Stack: B D F

Output: AE

Next Step:

- Pick F (Top)
- Pop F and place it in output by pushing the unvisited adjacent vertices of F (NIL) on to the stack

DFS Using Stacks



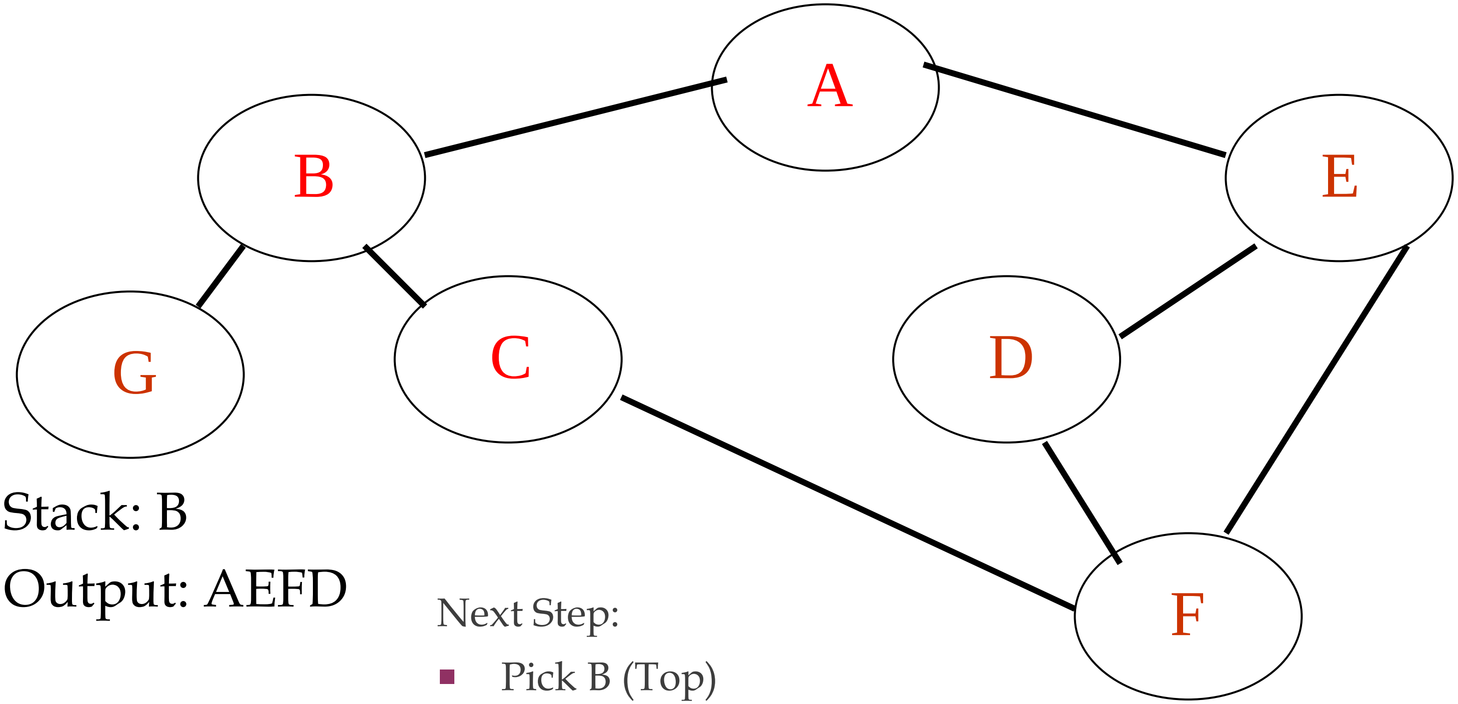
Stack: B D

Output: AEF

Next Step:

- Pick D (Top)
- Pop D and place it in output by pushing the unvisited adjacent vertices of D (NIL) on to the stack

DFS Using Stacks



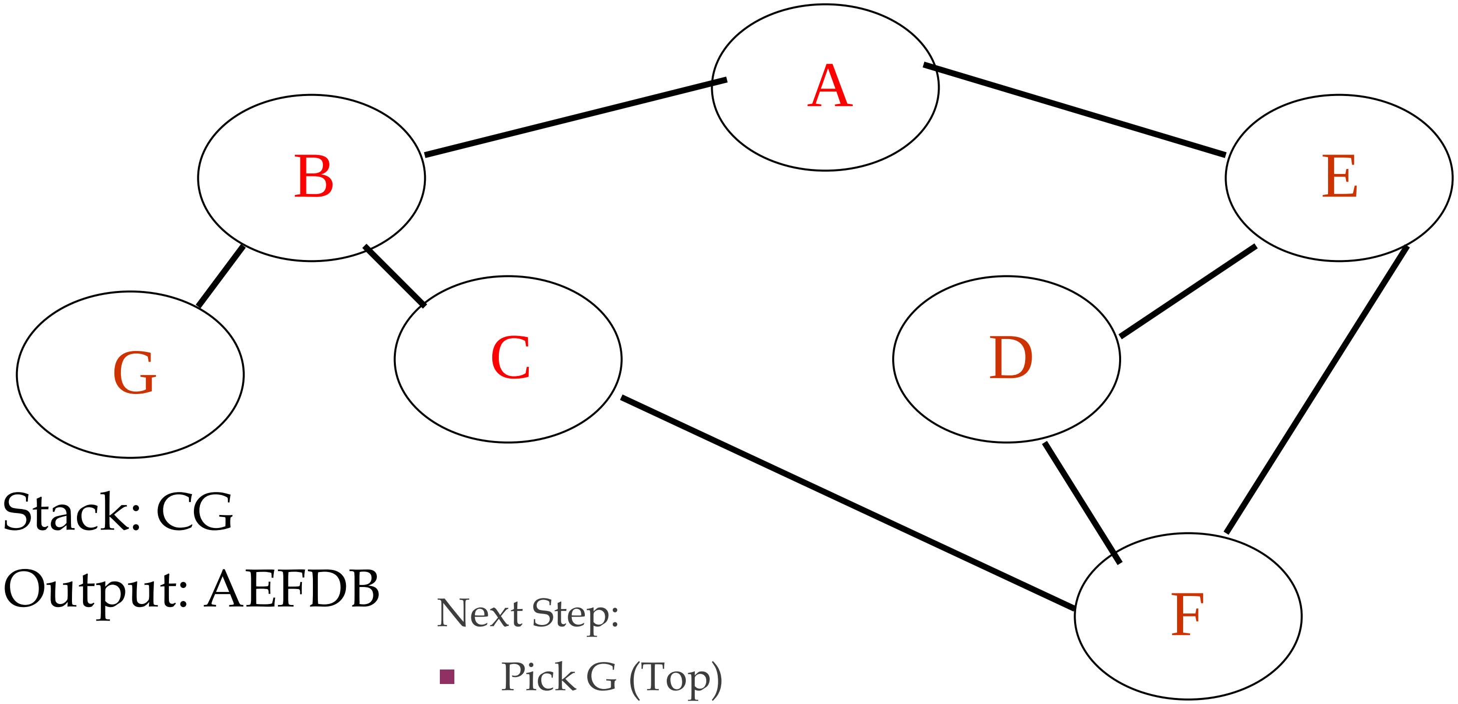
Stack: B

Output: AEFD

Next Step:

- Pick B (Top)
- Pop B and place it in output by pushing the unvisited adjacent vertices of B (C, G) on to the stack

DFS Using Stacks



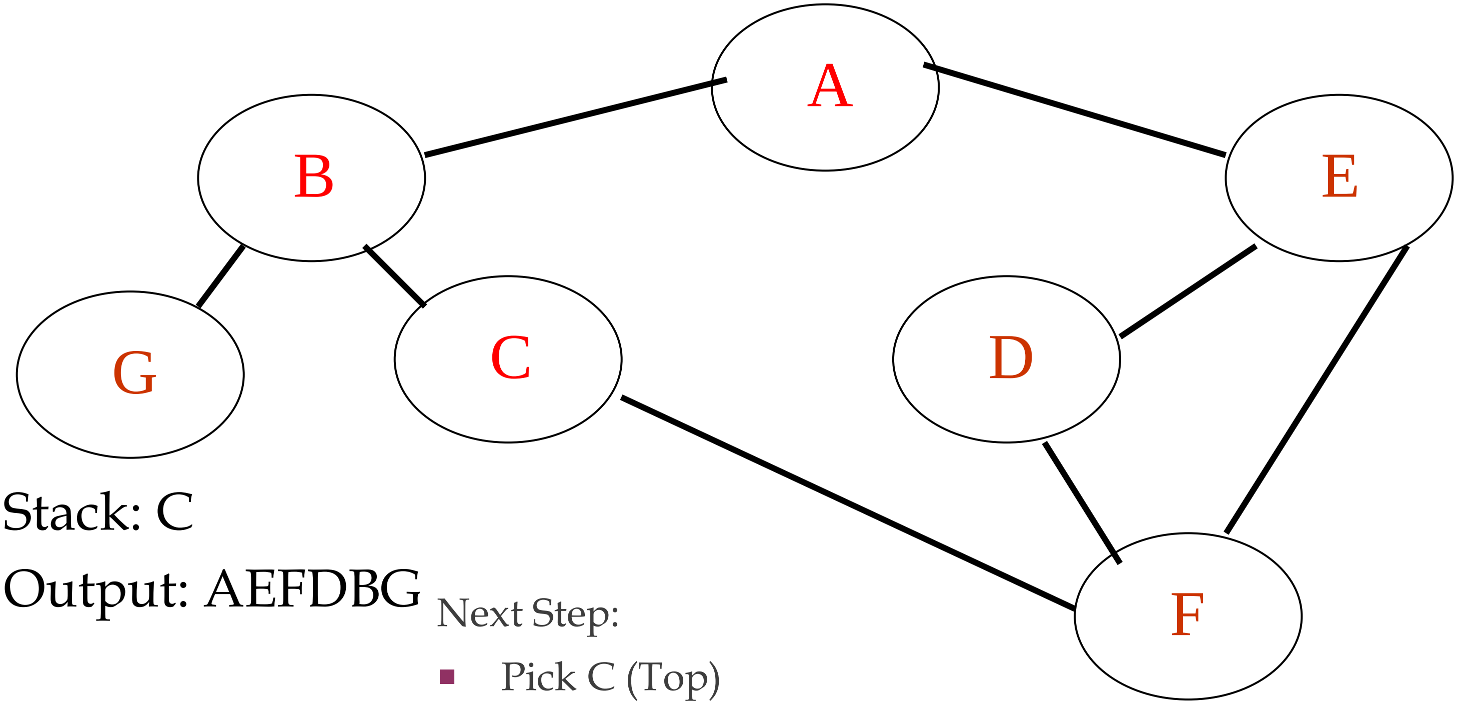
Stack: CG

Output: AEFDB

Next Step:

- Pick G (Top)
- Pop G and place it in output by pushing the unvisited adjacent vertices of G (NIL) on to the stack

DFS Using Stacks



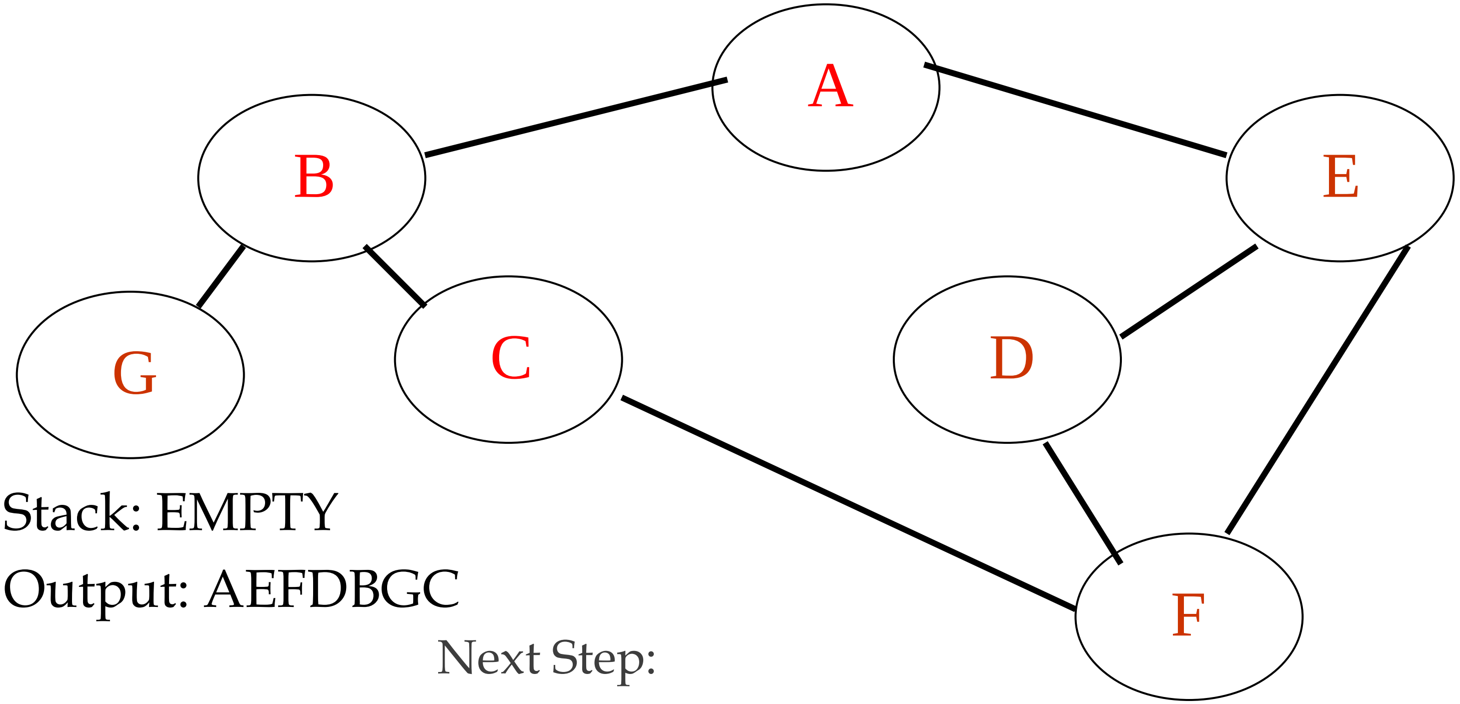
Stack: C

Output: AEFDBG

Next Step:

- Pick C (Top)
- Pop C and place it in output by pushing the unvisited adjacent vertices of C (NIL) on to the stack

DFS Using Stacks



Stack: EMPTY

Output: AEFDBGC

Next Step:

- Stack Empty -> No elements to process -> Final Outpu

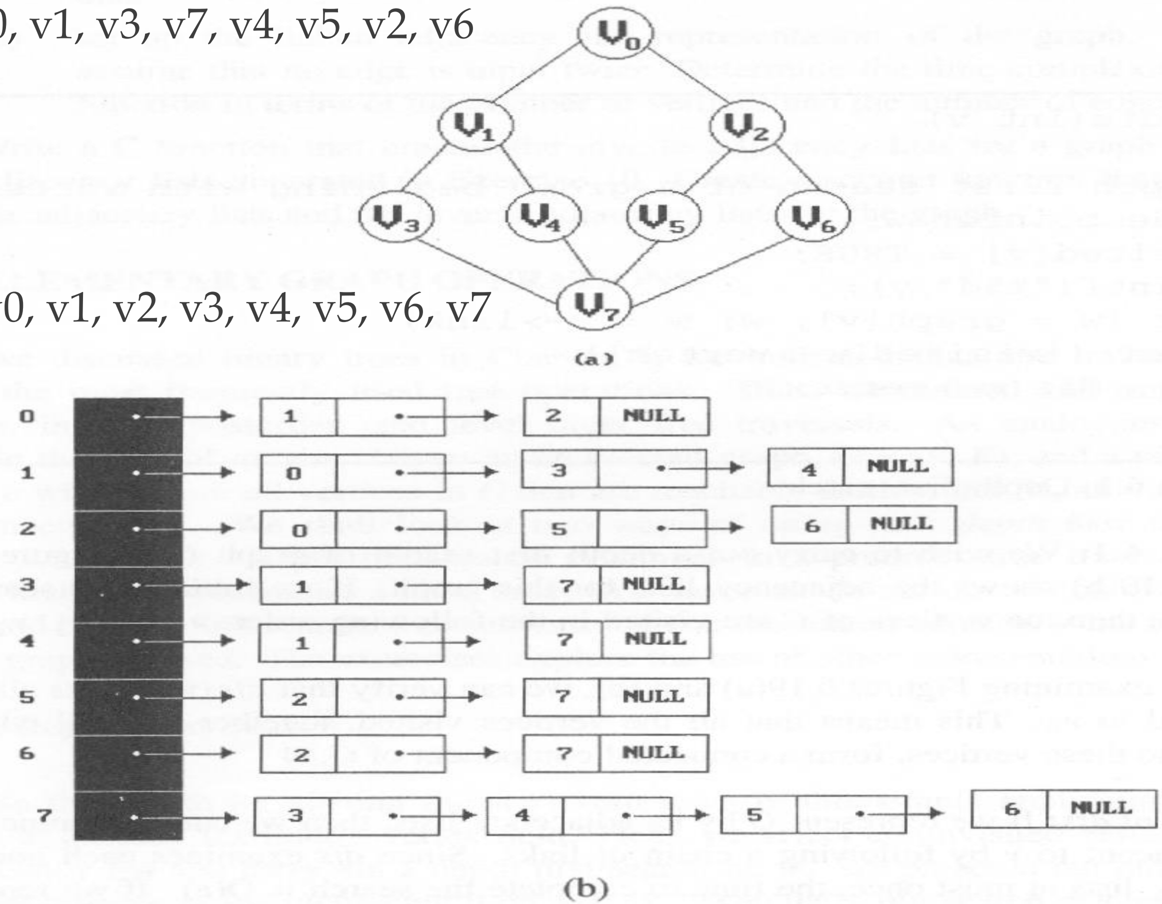
INTERESTING FEATURES OF DFS

- Complexity: O(|V| + |E|

DFS and BFS of a graph from given Linked List

Depth First Search: v0, v1, v3, v7, v4, v5, v2, v6

Breadth First Search: v0, v1, v2, v3, v4, v5, v6, v7



GATE ONLINE CLASSES

PART – 9
HASHING

- Agenda:
  - Introduction to Hashing
  - Steps Involved in Hashing
  - Components of Hashing
  - Hash Functions
    - Requirements for Good Hash Functions
    - Types of Hash Functions
    - Need for Good Hash Function
  - Hash Tables
  - Collisions
  - Collision Resolution Techniques
    - Linear Probing
    - Quadratic Probing
    - Separate Chaining

Hashing

INTRODUCTION TO HASHING

- **Hashing**
  - Hashing is a technique used for storing and retrieving information as quickly as possible.
  - Used to perform Optimal Searches
  - Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects
    - E.g: Student Registration Number in Universities, Books in Libraries etc.,
  - An object is assigned a key to it to make searching easy
  - To store the key/value pair, you can use a simple **array** like a data structure where keys (integers) can be used directly as an index to store values
  - Keys are large and cannot be used directly as an index you should use *hashing*

Steps Involved in Hasing

STEPS INVOLVED IN HASHING

- Hashing is implemented in two steps:
  - Step 1:
    - An element is converted into an integer by using a hash function. This integer value can be used to calculate the index which is used to store the original element
    - hash = hashfunc(key)
  - Step 2:
    - The element is stored in the hash table where it can be quickly retrieved using hashed key
    - index = hash % array_size

Components of Hashing

COMPONENTS OF HASHING

- Hashing has four key components
  - Hash Functions
  - Hash Table
  - Collisions
  - Collision Resolution Techniques

Hash Functions

HASH FUNCTION

- **Hash Function**
  - The hash function is used to transform the key into the index
  - A hash function should map each possible key to a unique slot index (Which is impossible in real practice)
  - The values returned by a hash function are called **hash values**, hash codes, **hash sums,** or simply **hashes**

Requirements for Good Hash Function

REQUIREMENTS OF GOOD HASH FUNCTION

- Basic Requirements of Good Hash Function:
  - **Easy to compute**:
    - It should be easy to compute and must not become an algorithm in itself
  - **Uniform distribution**:
    - It should provide a uniform distribution across the hash table and should not result in clustering
  - **Less collisions**:
    - Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided

Types of Hash Functions

TYPES OF HASH FUNCTIONS

- Division –
  - easiest method to create a hash function
  - the first order of business for a hash function is to compute an integer value
  - if we expect the hash function to produce a valid index for our chosen table size, that integer will probably be out of range and that is easily remedied by ***modding*** the integer by the table size
    - h(k) = k mod n
  - it is better if the table size is a prime, or at least has no small prime factors as that makes sure the keys are distributed with more uniformity
  - k=1501 n=10 h(1501) = 1501 mod 10 = 1
  - Disadvantage:
    - Consecutive keys map to consecutive hash values – which leads to poor performance

TYPES OF HASH FUNCTIONS CONTD…

- Folding –
  - begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value
  - portions of the key are often recombined, or folded together
    - shift folding: 123-45-6789 -> 123 + 456 + 789
    - boundary folding: 123-45-6789 -> 123 + 654 + 789
  - can be efficiently performed using bitwise operations

TYPES OF HASH FUNCTIONS CONTD…

- **Mid-Square Function –**
  - square the key, then use the middle part as the result
  - e.g., 3121 -> 9740641 -> 406 (with a table size of 1000)
  - a string would first be transformed into a number using ASCII values

TYPES OF HASH FUNCTIONS CONTD…

- **Extraction –**
  - use only part of the key to compute the result
  - The ISBN starting digits are the same for a publisher, so they should be exclude if the hash table is for only one publisher.

TYPES OF HASH FUNCTIONS CONTD…

- **Radix Transformation –**
  - change the base-of-representation of the numeric key, mod by table size
  - Example:
    - Key = 345, change to base 9 = 423 % TSize

Need for Good Hash Function

NEED FOR GOOD HASH FUNCTION

- Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }
- **Hash Function1: h1 =** The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599
  - The hash function will compute the same index for all the strings

NEED FOR GOOD HASH FUNCTION

- **Hash Function2: h2 =** The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number)

NEED FOR GOOD HASH FUNCTION

String	Hash function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23
cdefab	(991 + 1002 + 1013 + 1024 + 975 + 986)%2069	14
defabc	(1001 + 1012 + 1023 + 974 + 985 + 996)%2069	11

Hash Tables

HASH TABLES

- **Hash table**
  - A hash table is a data structure that is used to store keys/value pairs
  - It uses a hash function to compute an index into an array in which an element will be inserted or searched

HASH TABLES

- Provides virtually direct access to objects based on a key (a unique String or Integer)
  - key could be your SID, your telephone number, social security number, account number, …
  - Must have unique keys
  - Each key is associated with–mapped to–a value

HASH TABLES

- **Load Factor of the Hash Table:**
- It is the denoted by the symbol λ
- λ = (number of items in the table) / tablesize
- Example: Assume that the tablesize is 10 and it consists of 6 items, then the load factor of the table is λ = 6/10 = 0.6

Collisons

COLLISIONS

- A good hash method
  - executes quickly
  - distributes keys equitably
  - Has less collisions

COLLISIONS CONTD

- But you still have to handle collisions when two keys have the same hash value
  - the hash method is not guaranteed to return a unique integer for each key
    - example: simple hash method with "baab" and "abba“
- There are several ways to handle collisions
  - Linear Probing
  - Quadratic Probing
  - Separate Chaining

Collision Handling Mechanisms

Or

Collision Resolution Techniques

COLLISION RESOLUTION

- It is the systematic method for placing the collided item in the hash table
- Two methods of Collision Resolution
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
  - Chaining

Linear Probing

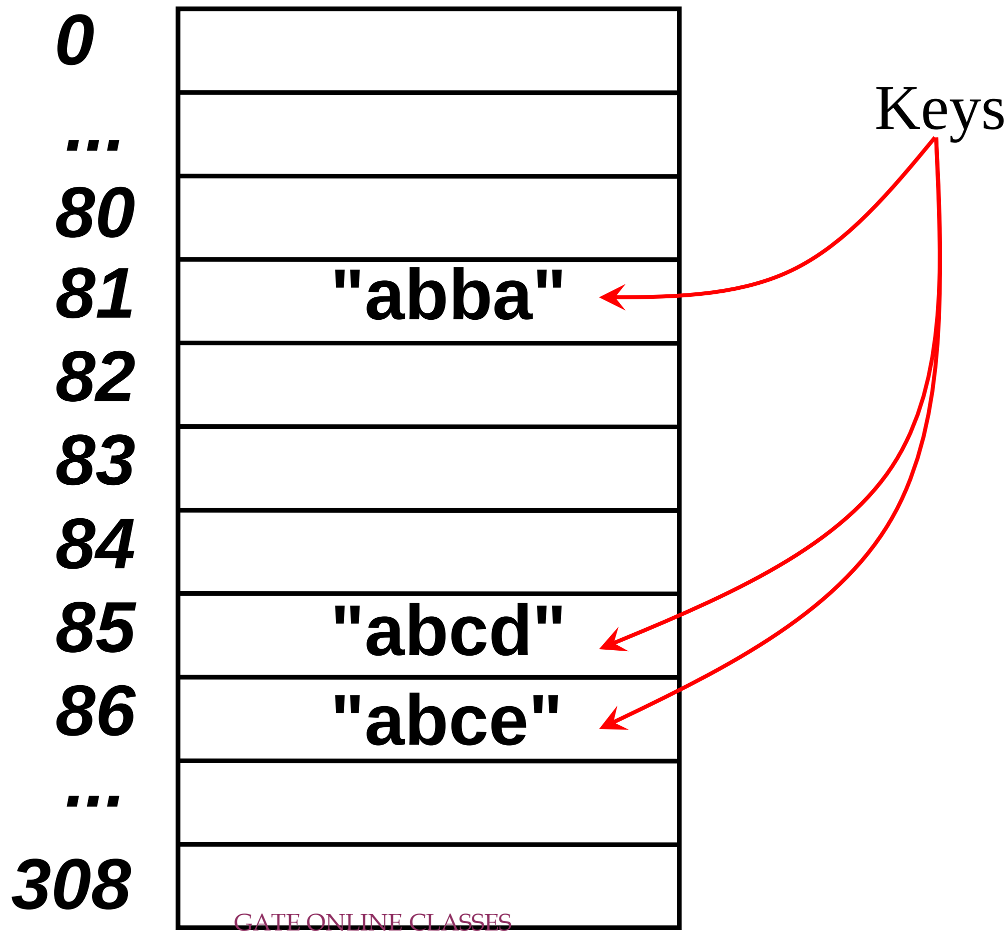
LINEAR PROBING

- *Linear Probing*:
  - search sequentially for an unoccupied position
  - uses a wraparound (circular) array

A HASH TABLE AFTER THREE INSERTIONS

insert objects with
these three keys:

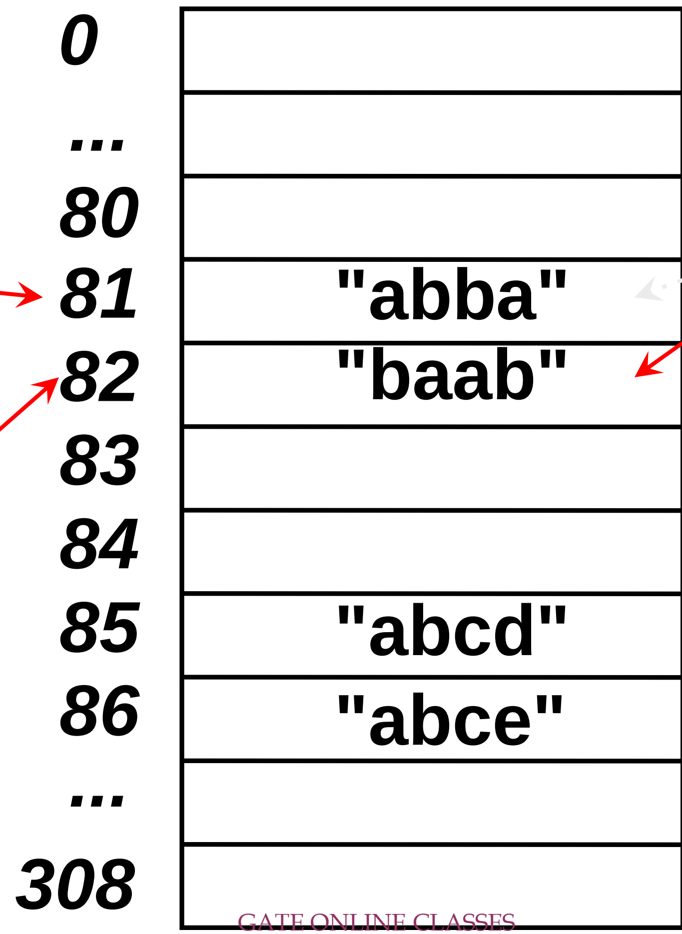
"abba"
"abcd"
"abce"



COLLISION OCCURS WHILE INSERTING “baab"

can't insert "baab" where it hashes to same slot as "abba"

Linear probe forward by 1, inserting it at the next available slot



"baab"
Try [81]
Put in [82]

WRAP AROUND WHEN COLLISION OCCURS AT END

Insert "KLMP" a
"IKLT"
both of which have
hash value of 308

0	"IKLT"
...	
80	
81	"abba"
82	"baab"
83	
84	
85	"abcd"
86	"abce"
...	
308	"KLMP"

FIND OBJECT WITH KEY “baab"

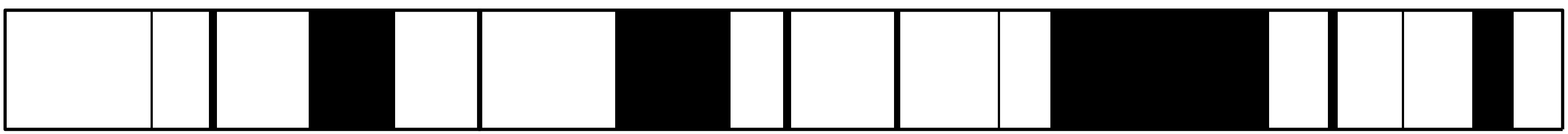
"baab" still hashes to 81, but since [81] is occupied, linear probe to [82]

At this point, you could return a reference or remove baab

0	"IKLT"
...	
80	
81	"abba"
82	"baab"
83	
84	
85	"abcd"
86	"abce"
...	
308	"KLMP"

LINEAR PROBING HAS CLUSTERING PROBLEM

- Used slots tend to cluster with linear probing



Black areas represent slots in use; white areas are empty slots

Quadratic Probing

QUADRATIC PROBING

- Quadratic probing eliminates the primary clustering problem

- Assume hVal is the value of the hash function

  - Instead of linear probing which searches for an open slot in a linear fashion like this

    hVal + 1, hVal + 2, hVal + 3, hVal + 4, ...

  - add index values in increments of powers of 2

    hVal + 1*1, hVal + 2*2, hVal + 3*3, hVal + 4*4, ...

DOES IT WORK?

- Quadratic probing works well if
  - Table size is prime
    - studies show the prime numbered table size removes some of the non-randomness of hash functions

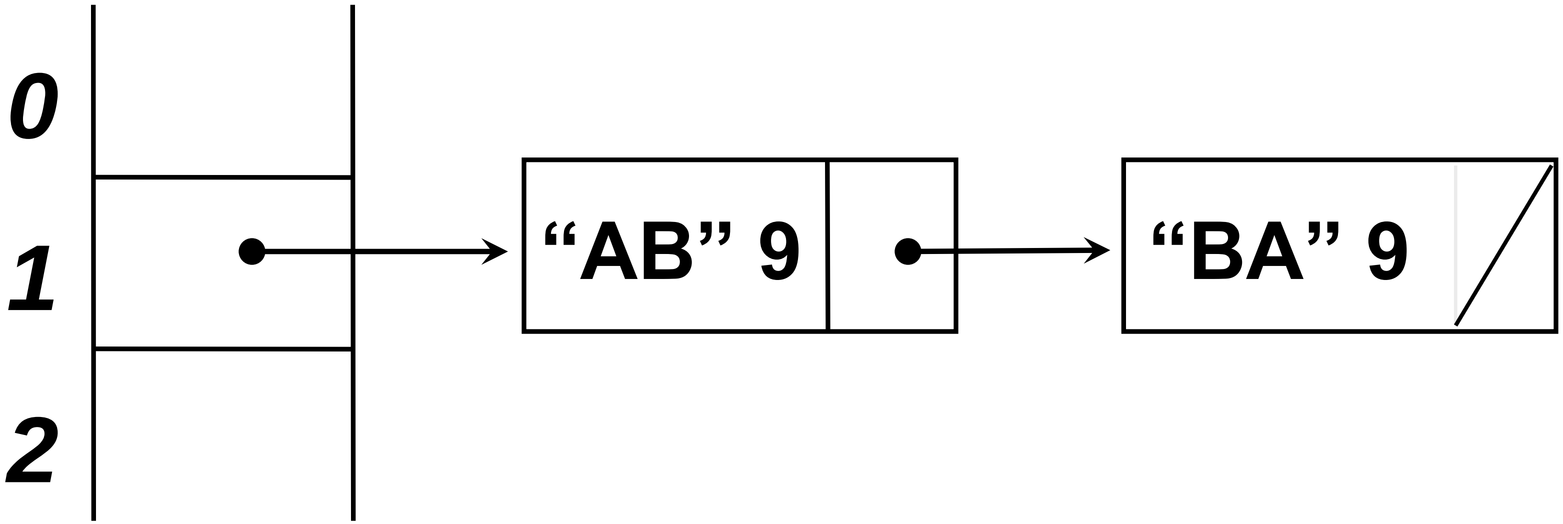
Separate Chaining

SEPARATE CHAINING

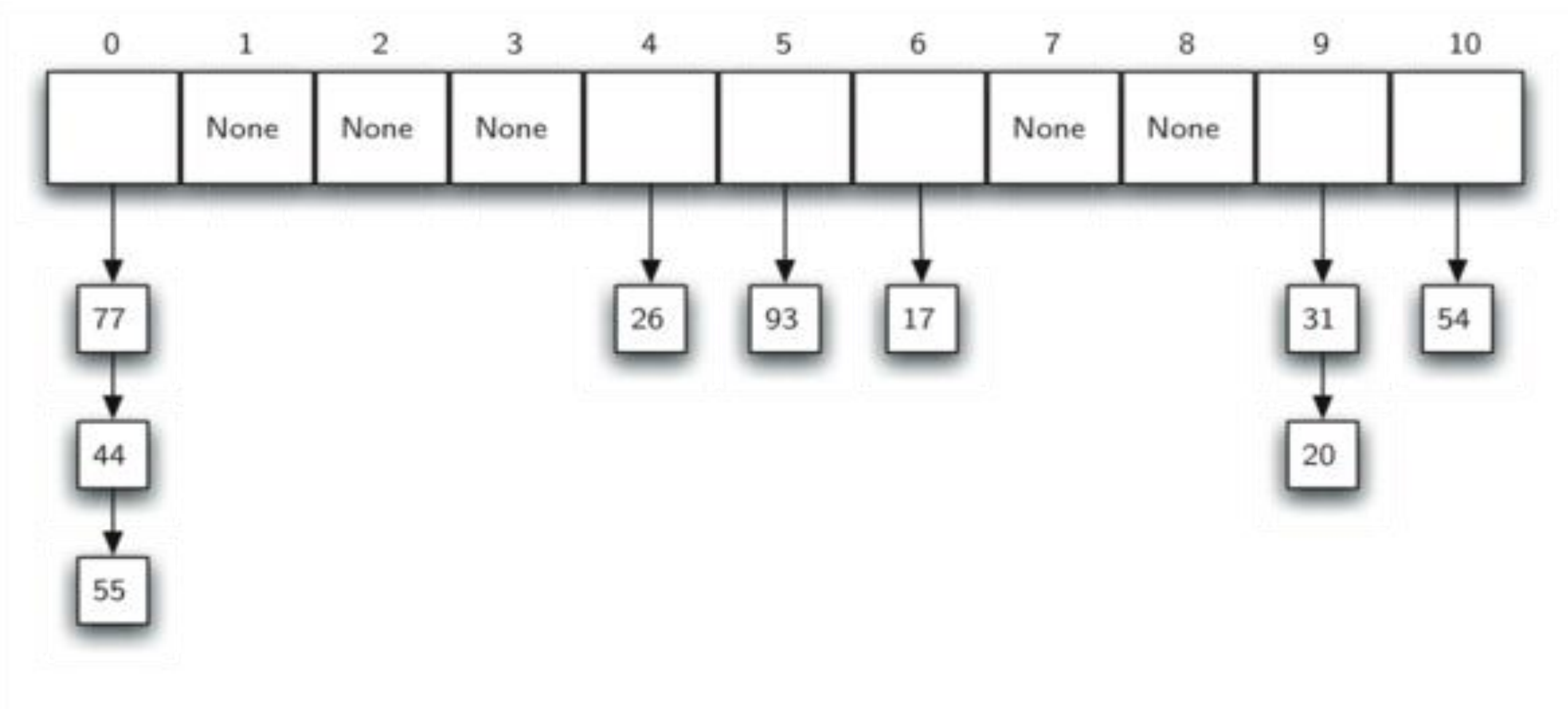
- Separate Chaining is an alternative to probing
- How?
  - Maintain an array of lists
- Hash to the same place always and insert at the beginning (or end) of the linked list.

ARRAY OF LINKEDLISTS DATA STRUCTURE

- Each array element is a List



SEPARATE CHAINING



HASHING SUMMARY

- Hashing involves transforming a key to produce an integer in a fixed range (0..TABLE_SIZE-1)
- The function that transforms the key into an array index is known as the hash function
- When two data values produce the same hash value, you get a collision
- Collision resolution may be done by searching for the next open slot at or after the position given by the hash function, wrapping around to the front of the table when you run off the end (known as linear probing)

HASHING SUMMARY

- Another common collision resolution technique is to store the table as an array of linked lists and to keep at each array index the list of values that yield that hash value *known as separate chaining*

- Most often the data stored in a hash table includes both a key field and a data field (e.g., social security number and student information).

- The key field determines where to store the value

- A lookup on that key will then return the value associated with that key (if it is mapped in the table)


The End

End of

Day – 10 Lecture Notes

on

DATA STRUCTURES