CSE-203

5.2.20

Book – Schaum's series → Seymour ~~tipscutz~~ Lipschutz
Data Structures with C

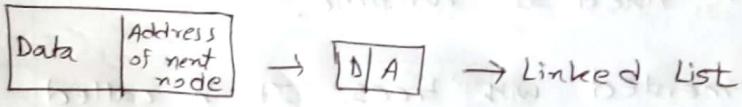
DS are used in : ① OS, ② Compiler ③ AI, ④ Image Processing

*) Entity → Represents the class of certain objects

*) Attribute → Represents particular property of an entity.

*) Field → Single elementary unit of information.

Node:

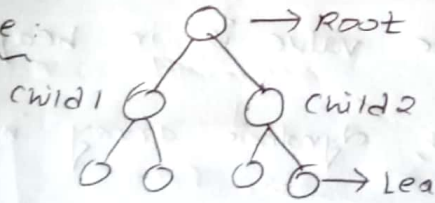


→ Linked List

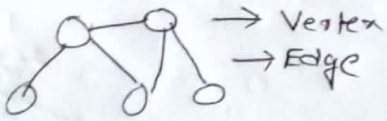
Stack : LIFO

Queue : FIFO

Tree:



Graph:



① Graph can have multiple edges but trees will have only two edges.

② No cycles ~~can~~ → is possible in tree, possible only in graph

③ Graph edges can have weight.

②

*) Memory allocation is divided into 4 parts :-

① Heap →
② Stack → Function calls and local variables
③ Static / Global → Global / static variables
④ Code → Instruction

Stack Overflow → Unavailability of space in stack.

Size of stack is fixed at compile time.

Size of heap is fixed at runtime.

Heap is implemented with trees. It is called free pool / store of memory.

```
p = (int *) malloc (sizeof (int));
*p = 15 ;
```

→ Pointer is stored at stack, but the value is in heap.

→ Static ~~variable~~ array is stored in stack, dynamic array is stored in heap.

For C++,

```
*p = new int;
*p = 20;
delete p;
```

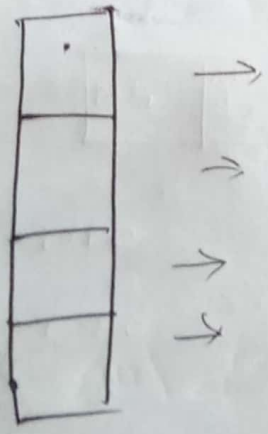
For 2D Allocation : `int *a = (int *) malloc(r * c * sizeof(int));`

To represent 2D array in 1D way : `*(arr + i*c + j) = ++count;`

To represent 2D in 2D way, we need double pointers.

```
int **a = (int **) malloc (r * sizeof (int *));
for (i=0; i<r; i++)
    a[i] = (int *) malloc (c * sizeof (int));
```

**ann = 4x int



CSE-203

6.2.20

Traversing a 2D array :->

arr [i*c + j] → Row major traverse

arr [i + j*r + i] → Column major traverse

Freeing dynamic 2D array :

```
for (i=0; i<r; i++)
    free (arr[i]);
free (arr);
```

*) Memory Manager :- ① Keeps track of free space
↓ ② Allocates space on request of program
Part of OS

*) Defragmentation :-

Removes Bad sector of hard disk. All the memory ~~has be~~ is taken at a side to remove wasted space.

*) Linked List :

It is a linear data structure in which each node has two parts where one part contains value and the second ~~node~~ part contains pointers to next node.

```
struct node
{
    int item;
    node *next;
};
```

⑤

CW

CSE 203

12.7.20

Linked List

Note :

① A node member is referenced with "→" member.

p→item

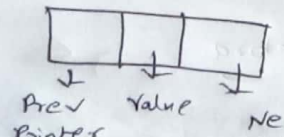
② Traversing next node : cur = cur→next;

~~*) Deleting the last node:~~

*) Lookup (search a node) ~~(int x)~~ (int * L, int x)

```
{
    if (L->next == NULL)
        return FALSE;
    else if (x == L->value)
        return TRUE;
    else
        return lookup (x, L->next)
```

*) Doubly Linked List :-

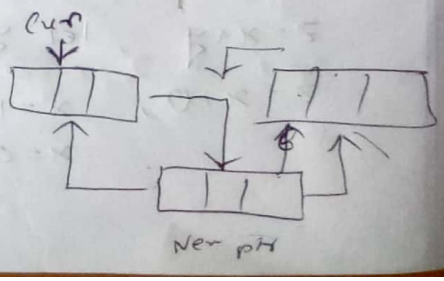


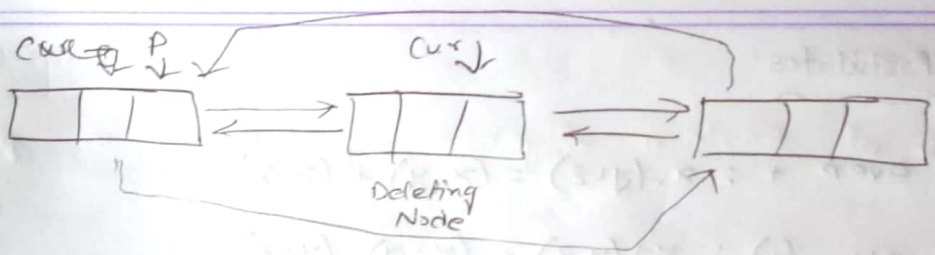

```
struct node
{
    int item;
    node *prev;
    node *next;
}
```

newptr→next = cur→next;

Assignment → Create a Double linked list. Traverse both ways

```
newptr->next = cur->next;
newptr->prev = cur;
cur->next = newptr;
newptr->next->prev = newptr;
cur->next->next->prev = newptr
```






```
p->next = cur->next ;
cur->next->prev = p ;
free(cur) ;
```

Ct syllabus → Pointer, dma, linked list, double linked list

(2)

CW

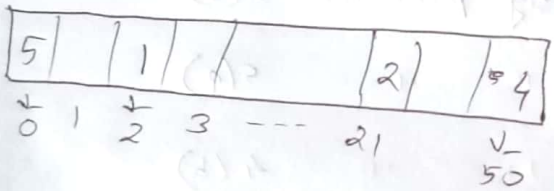
CSE-203

13.2.2

Polynomial : $4x^{50} + 2x^{21} + x^2 + 5$

Coefficient, Power

int arr[50];



Array index = Power
Array value = Coefficient

Circular Linked List:

Last node points to first node.

Array → Accessed by index, Best case : Constant O(1)
Linked list → " " traversing " : O(1), Worst : O(n)

(8)

CSE-203

CN

19.2.20

Stack

Basic Operation : Pushing, Popping

Abstract data type

→ Stack can be implemented with array or linked list

→ LIFO

```
struct node
{
   double data;
   node *next;
};
```

Note: No stack overflow in ~~dy~~ dynamic stack.

⑨

CSE-203

CW

20.2.20

Stack

Application of stacks:

① Line Editing [Use backspace to correct error of input]
② Bracket matching
③ Postfix calculation
④ Function call stack

*) Infix Notation:

Operators in the middle, numbers on both sides. (operands)

*) Prefix:

++a, +ab [a+b]

*) Postfix:

a++, ab+

*) Parenthesis:

~~Brackets Off~~

→ Brackets have more precedence than operators.

(2+3)*5 = 25

2+(3*5) = 12

*) +~~2*35~~ + 2 * 3 5 = + 2 15 = 17

*) * + 235 = * + 2 3 5 = * 5 5 = 25

* Note: Parenthesis required in infix notation, not in prefix or postfix notation.

*) 2 3 5 * + = 2 3 5 * + = 2 15 + = 17

*) 2 3 + 5 * = 2 3 + 5 * = 5 5 * = 25

*) Fully Parenthesized Expression:

A FPE has exactly one set of parenthesis enclosing each operator.

((A * B) + C)

Infix to Prefix:

((A+B) * (C+D))

= (+AB * +CD)

= * +AB +CD (Ans)

Infix to Postfix:

*) (((A+B) * C) - ((D+E)/F))

= ((AB+ * C) - ((D+E)/F))

= (AB+ C * - (DE+ / F))

= (AB+C * - DE+F /)

= AB+C* DE+F/ - (Ans)

(11)

Infix to Postfix:

For programming,

if it is operand : append to output

" " " operator or '(' : push onto stack

" " " ')' : pop till we get a '(' and output

Note: Infix should be fully parenthesized.

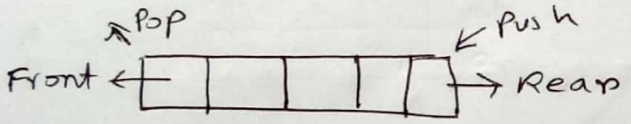
CSE-203

27.2.20

Queue

→ First In, First Out

→ Enqueue, Dequeue



→ For static queue, overflow can occur even if the queue is not full.

→ Destructor destroys the instance/object members.

Assignment → Memory Allocation in OOP.

→ Use circular queue to avoid "shifting" and improve efficiency.

(13)

CSE-203

CN

4.3.20

Circular Queue

Rear pointer is shifted using mod

rear = (rear + 1) % maxsize ; [Enqueue]

For dequeueing

*) Priority Queue:

A priority queue is a type of queue in which each element is assigned a priority.

*) DEQue :

Double Ended Queue . Also known as Dech.

→ Elements can be inserted or deleted from both sides.

① Input restricted Deque [Input 1 side, output both sides]
② output " " [Input both sides, output 1 side]

CSE-203

CW

5.3.20

Time Complexity

① **Best Case:**
Minimum time for an algorithm. [Lower Bound time]

② **Worst Case:**
Maximum time for an algorithm. [Upper Bound time]

③ **Average Case:**
Anything between best and worst case

$O \rightarrow$ Upper Bound
$\Omega \rightarrow$ Lower Bound
$\Theta \rightarrow$ Average case line

① `for (i=1; i<=n; i++)` → Loop iterates ~~(n+1) time~~ n times
Loop condition check (n+1) times
`{ statement; }` → statement executes n times

if $n = 5$,
condition check: $i = 1, 2, 3, 4, 5, 6 \rightarrow$ Comes out of loop
∴ Total time $= (n+1) + n = (2n+1)$ ~~time~~ units
Order: $O(n+1)$ [this O is order, not big O]
$\approx O(n)$

Time complexity is **order of n**

Note: Any constant added or coefficient will be omitted when we calculate order. Order will only depend on power.

② for (i = n; i >= 1; i--)
{
    statement;
}

Order: O(n)

③ for (i=1; i<=n; i = i+2) → $\left(\frac{n}{2}+1\right)$ times
{
    statement; → $\frac{n}{2}$ times
}

Order: O(n)

if n = 10, i = 1, 3, 5, 7, 9, 11

∴ Time = $\left(\frac{n}{2}+\frac{n}{2}+1\right) = (n+1)$

④ for (i=1; i<=n; i++) → (n+1) times
{
    for (j=1; j<=n; j++) → n(n+1) times
        statement; → ~~n times~~ ~~n(n+1)~~ times
                n × n
}

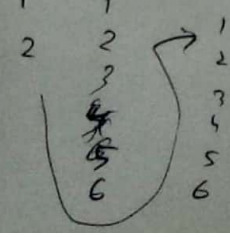
Total: ~~$(n+1) + n(n+1) + n(n+1)$~~ = ~~$(n+1)(n+1)+n$~~ = ~~$n + (n^2+1)^2$~~

Total time = $(n+1) + n(n+1) + n \times n$

$= n+1+2n^2+n+2n^2$

$= 2n^2+2n+1$

∴ Order: $O(n^2)$



⑤
```
for (i=1; i<=n; i++)  → (n+1) times
{
    for(j=1; j<=i; j++)  → n(n+1)/2
    {
        statement;
    }
}
```

Order: (n^2)

i	j	No. of times
1	1 X	1
2	1	2
	2 X	
3	1	3
	2	
	3 X	
4	1	4
	2	
	3	
	4 X	

⑥
```
For (i=0; i<=n; i++)
{
    for(j=0; j<i; j++)
    {
        statement;
    }
}
```

i	j	No. of times
0	0	0
1	0	1
	1 X	
2	0	2
	1	
	2 X	

∴ Order: (n^2)

(x) `p = 0;`

```
for (i=1; p<=n; i++)
{
    p = p+i;
}
```

i	p	No. of times
1	0+1=1	
2	1+2=3	
3	3+3=6 [1+2+3]	
4	10 [1+2+3+4]	
↓	↓	
k	[1+2+3+...+k]	

Run until k

$k(k+1)$ / 2 times

Loop stops at k.

$p > n$

$\therefore \frac{k(k+1)}{2} > n$

$\therefore k^2 > n$

$\therefore k > \sqrt{n}$

$\therefore$ Order of code: $O(\sqrt{n})$

CSE-203

11.3.20

Time Complexity

*)
```
for (i=1; i<n; i=i*2)
{
  statement;
}
```

1, 2, 4, 8, 16 10 n=10

i
1 = 2^0
2 = 2^1
4 = 2^2
8 = 2^3
16 = 2^4

Complexity : $O(\log_2 n)$

Assume at 2^k, $i >= n$

$\therefore 2^k \geq n$

~~Let~~ Let, $2^k = n$

or, $k = \log_2 n$

For $i = i*3$, Complexity : $O(\log_3 n)$

If $n = 8$, then loop will run 4 times. → i = 1, 2, 4, 8 (I comes out of loop)

Complexity : $O(\lceil \log n \rceil)$ [ceil]

$O(\lfloor \log n \rfloor)$ ~~[Roof]~~ [Floor]

*)
```
for (i=n; i>0; i=i/2)
{
  --
}
```
Complexity : $O(\log_2 n)$

*)
```
for (i=0; i<n; i++)
{
}
```
-- n

```
for (j=0; j<n; j++)
{
}
```
--- n

$O(2n) \approx O(n)$

*) `p=0; for (i=1; i<n; i=i*2) { p++; }`

$\log_2 n$

i	p
1	0
2	1
4	2
8	3

[Here, the two loops are dependent]

```
for (j=1; j<p; j=j*2)
{
  ---
}
```

~~$\log_2(\log_2 n)$~~

~~$\log(\log n)$~~

$\log_2 p = \log_2(\log_2 n)$

$\log n + \log(\log n)$

$\therefore O(\log n)$

i	
1	
2	
4	
8	

p	
0	
1	
2	
3	
4	

*) `for (i=0; i<n; i++)` —— n

```
{
  for(j=1; j<n; j=j*2)
  {
  }
}
```

~~log n~~

— n log n

$n + n\log n$

$\therefore O(n\log n)$

① `for(i=0; i<n; i++)` → $O(n)$

② `for (i=0; i<n; i=i+2)` → $\frac{n}{2}$ → $O(n)$

③ `for (i=1; i<n; i=i*2)` → $O(\log n)$

`i=i*3` → $O(\log_3 n)$

CT syllabus → Stack, Queue, Time Complexity

*) Analysis of while loop:-

Cases: 1, $\log\log n$, $\log n$, $\sqrt{n}$, n, $n\log n$, n^2, n^3

Complexity decrease ← (w) (Ω)

Complexity increase → (O)

*)
```
while (i<n)
{
   i++;          → O(n)
}
```

*)
```
a=1
while (a<n)
{
   a=a*2;        → O(log n)
}
```

*)
```
i=1;
k=1;
while (k<n)
{
   k=k+i;        → O(√n)
   i++;
}
```

k	i
1	1
2	2
4	3
7	4
11	5
16	6
22	7
29	8

if n=8

k	i
1	1
2	2
4	3
7	4
11	5
k	m

n=8

$3 < 8 < 9$

$= 2 < \sqrt{8} < 3$

$n = 16$

$\log 2^n$

*)
```
while (m!=n)
{
  if (m>n)
     m=m-n;
  else
     n=n-m;
}
```

m	n
6	3 [Let]
3	3 [Ends]

2 times

$k > n$

$\frac{m(m+1)}{2} > n$

Again,

$\Rightarrow m^2 > n$

$\Rightarrow m > \sqrt{n}$

m	n
5	5

→ 1 time

∴ Best case : O(1)

Worst case : O(n)

m	n
16	2
14	2
12	2
10	2
8	2
6	2
4	2
2	2

→ 8 times

$m/2$ times

(21)

CW

CSE-203

12.3.20

Graph

Heirarchical : 1 to many

Graph : Many to many

Graph is a set of vertices and edges. $G(V, E)$

Edge is a collection of pairs of vertices.

$V = \{a, b, c, d, e\}$

$E = \{(a,b), (a,c), (a,d), (b,c), (c,d)\}$

Note : A tree is a special type of graph

① Simple Graph :- Undirected, Unweighted, No loops or multiple edges

② Non simple Graph

Ordered pair (u,v) → u → starting node, v → ending node

Digraph → Directed graph

→ u, v are end vertices (endpoints) of edge a.

→ edges a, d, b are incident on v

→ Degree of a vertex : Total edges incident on a vertex.

Path :-

Sequence of alternating vertices and edges.

Begins and ends with ~~vertice~~ vertex.

Simple Path :- No vertex, edge repeated

Not simple : Repeated

*) Cycle:

start and end vertex same

*) Dense Graph:

$|E| \approx |V|^2$

*) Sparse Graph:

$|E| \approx |V|$

Complete Graph is a graph that has maximum no. of edges → No parallel edges

For undirected graph with n vertices, Max = $\frac{n(n-1)}{2}$

" directed " , Max = $n(n-1)$

CSE-203

CW

Graph

13.5.20

→ A graph is a pair (V, E) where V = Vertices, E = Edges

→ A tree is a special type of graph.

Simple Graph → No multiple ~~too~~ edge or loop

Non-simple " → Has " " " "

Application :- ① Electronic Circuits
② Transportation Network
③ Computer Network
④ Database

Edge :- ① Directed → Directed Graph
② Undirected → Undirected Graph
③ Both → Mixed Graph

Degree : Total edges connected to a vertex

Outgoing edges → Out-degree of a vertex
Incoming edges → In-degree of a vertex

Path: Sequence of alternating vertices and edges

Simple Path → All vertices and edges are distinct

Cycle: A path whose start and end vertices are same.

Simple cycle → All vertices and edges are distinct except the start and end vertex.

Dense Graph : $|E| \approx |V|^2$

Sparse Graph : $|E| \approx |V|$ → Spanning Tree

Weighted Graph : Associates weight with either edge or vertex

Complete Graph : Graph with maximum number of edges

$\frac{n(n-1)}{2}$ for undirected graph

$n(n-1)$ for directed graph

CSE-203

CW

Graph 14.5.20

A subgraph G' of a Graph G such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Spanning Subgraph:

Contains all vertices of G.

$V(G') = V(G)$, $E(G') \subseteq E(G)$

Forest:

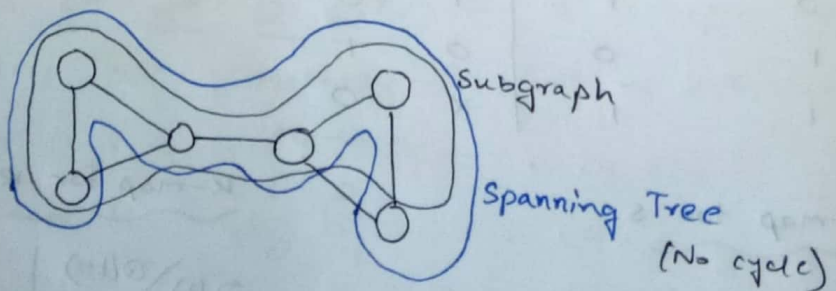
A Graph without cycle

Free Tree:

A connected forest, that is connected graph without cycles.

Spanning Tree:

Spanning subgraph that is a free tree



Graph G

CSE-203

CW

Graph 1.6.20

*) Components:

Each connected parts of graph are called components of a graph.

*) Path:

If two nodes are reachable between each other, then a path is said to be present between the two nodes.

Note: A directed graph is strongly connected if there is a directed path from v_i to v_j and vice-versa.

Properties:

If n is no. of vertices and m is no. of edges

① $\sum_v deg(v) = 2m$ for undirected graph

② $\sum_v indeg(v) = \sum_v outdeg(v) = m$ for directed graph

③ $m \leq \frac{n(n-1)}{2}$ for undirected graph

$m \leq n(n-1)$ for directed graph

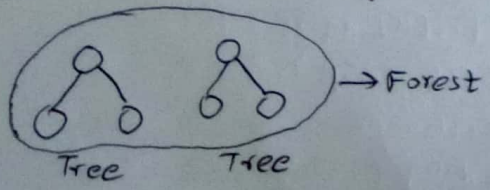
④ For a graph G,

G is connected if $m \geq n-1$

G is tree if $m = n-1$

G is forest if $m \leq n-1$

Note: Forest is actually a collection of non-connected trees.



CSE-203

CW

& 2.6.20

Graph

*) Graph Representation:

① Adjacency matrix
② Adjacency list

An adjacency matrix represents the graph as a nxn matrix.

A[i, j] = 1 if edge (i, j) ∈ E
= 0 if " " ∉ E

Array List = Linked List + Array

For adjacency matrix:

Pros:

① Simple to implement
② Edge can be traced in O(1)

Cons:

① Matrix takes $O(n^2)$ memory

Adjacency list:-

A graph is represented by one-dimensional array L of linked lists
→ L[i] is the linked list containing all the nodes adjacent to node i.

Pros:

① Saves memory. Takes O(|V|+|E|)
② Good for large, sparse graph

Cons:

① Searching an edge can take O(n) time

CSE-203

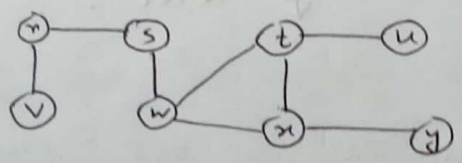
2.6.20

Graph Searching Techniques

*) Breadth First Search (BFS)

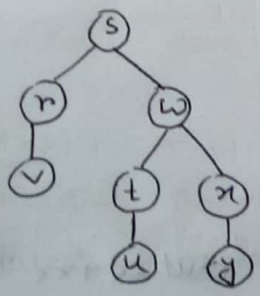
Goal: Methodically explore every vertex and edge

→ Ultimately builds a tree
→ Pick a vertex as the root
→ Choose certain edges to produce a tree
→ Might build a forest if graph is not connected
→ "Queue" is required for BFS



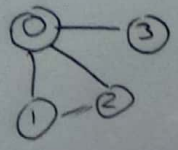
Start from s

Traversal: s, w, r, t, x, v, u, y



Tree Building in BFS

Graph Representation: Use adjacency list/matrix (2D array)
Data structure: Queue



List:

0 → 1 → 2 → 3 → ∅
1 → 0 → 2 → ∅
2 → 0 → 1 → ∅
3 → 0 → ∅

Matrix:

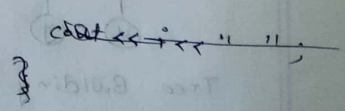
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Code of BFS:

Note: For using std queue, use header
#include <queue>

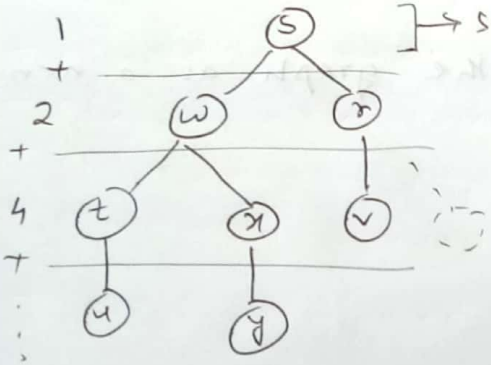
File open: freopen ("in.txt", "r", stdin)

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int vertices, edges;
    cout << "Enter total vertices:";
    cin >> vertices;
    cout << "Enter total edges:";
    cin >> edges;
    int graph[vertices][vertices]={0}; int visited[vertices]={0};
    for(int i=1; i<=edges; i++) {
        cout<<"Enter the two nodes for edge "<<i<<":";
        int u,v;
        cin>>u>>v;
        graph[u][v]=1;
        graph[v][u]=1;
    }
    int start;
    cout<<"Enter starting vertex:";
    cin>>start;
    queue<int>q;
    q.push(start);
    visited[start]=1;
    cout<<start<<" ";
    while(!q.empty()){
        int x=q.front();
        q.pop();
        for(int i=0; i<vertices; i++) {
            if(graph[x][i])
                if(!visited[i]){
                    visited[i]=1;
                    q.push(graph[x][i]);
                    cout<<i<<" "; }}}
```



Time Complexity: $O(V+E)$ → Worst Case

Note: Time complexity can be calculated through the tree formed.



→ search Breadth-wise

$O(b^d) \approx O(V+E)$

Note: With adjacency matrix, time complexity can become $O(n^2)$. The efficiency is less. Using adjacency list or "vector" can be a good way to increase efficiency.

Uses:

① Finding shortest path

② Prim's minimum spanning tree and Djikstra algorithm use the concept of bfs.

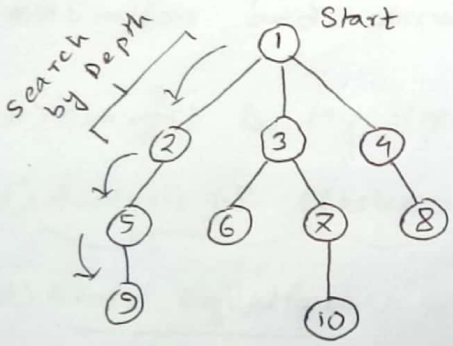
CSE-203

3.6.20

Graph Searching Techniques

*) Depth First Search (DFS):

→ Explore "deeper" in the graph whenever possible



Traversal: 1, 2, 5, 9, 3, 6, 7, 10, 4, 8

Data structure → stack, Recursive approach don't require stack

*) Code of DFS:

```
#include<bits/stdc++.h>
using namespace std;
int adj [10][10];
bool visited [10];
void dfs (int node);
int main () {
    int node, edge
    cin>>node >>edge;
    for (int i=1; i<= edge; i++) {
        int u,v;
        cin >>u>>v;
        adj[u][v] = 1;
        adj [v][u]=1;
    }
    for (int i=0; i<node; i++)
        dfs(i);
}
```

```
void dfs (int node)
{
    if (visited [node])
        return;
    else {
        visited [node] = true;
        cout << " " << node << " ";
        for (int i = 0; i < node; i++)
            dfs (adj [node] [i]);
}
```

Time Complexity : O(V+E)

Uses of DFS:

① Used for topological sorting
② Used for finding out components of graph