Date :
Page No.

Stack and Queues

Introduction to stack

Application → ① Recursion
② Infix to postfix conversion
③ Parsing
→ matching parenthesis
→ match tags < >
④ Browsing
⑤ Editors
⑥ Tree traversals and graph traversals.

Implementation of stack (LIFO, FILO)

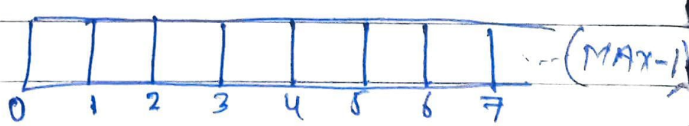
i) Using array.

Array have fixed size so less flexible but take less time and in linked list we have dynamic size so more flexibility and more time.

We can use array with dynamic m/m allocation also but if you are using dynamic m/m then go with structure is better (linked list).

Ex
```
int stack [MAX];
int top = -1;
void push (int item) {
if (top == MAX-1)
  Pf ("Overflow");
else {
   top++;
   stack [top] = item; }
return;
```

```
# define MAX 100
# we assume array start with 0.
```




```
# stack [++top] = item;
```

Pop()

```
int pop(){ int temp;
 if (top==-1) {
   pf ("Underflow"); return -1; }
 else {
    temp = stack[top]; }
    top--; }
 return temp;
}
```

→ To show there is an error we can return any other value.

temp = stack[top--];

temp.

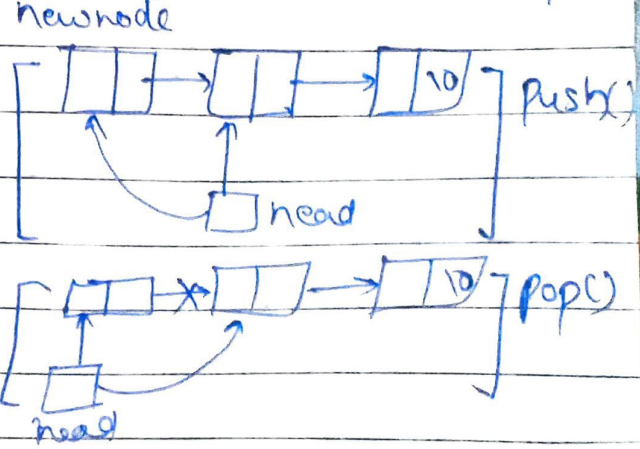
Popping → It doesn't mean we are erasing the element the element will exist there but top will decrement to previous and top behaves like there is no value further. and when we push() the value by top++ then that will get overwrite.

Time Complexity → In both the cases either push() or pop() time complexity will be O(1) :- [order of (1)].

Using Linked List —:

```
Struct node {
   Int i;
   Struct node * link; };
Struct node * head;
push (int item) {
Struct node *p = (struct node *) malloc
                 (sizeof (struct node));
 If (p==NULL){
   pf ("error of malloc");
   retur; }
```

head is always pointed at the top.



[If heap have space to allot then show error of malloc]

Date :
Page No.

```
p->data = item;
p->link = NULL;      ] we can neglect
p->link = head;
head = p;
```

No need to check overflow conditions; but may check for underflow

Time Complexity → O(1)

Pop() :-

```
int pop(){
int item; struct node *p;
if (head == NULL)
  { Pf("Underflow"); return -1; }
  item = head->data(i);
    p = head;
  head = head->next;
    item = p->i;
    free(p); }
return item;
```

Time Complexity → O(1)

Implementation of Queue (FIFO, LILO)

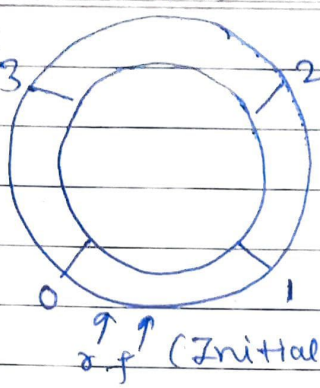
Using Circular array.] Normal array but visualise as circular

Queue is also a linear data structure.

We use circular array bcoz because lot of free space available remain unused normally, so using circular array we can reuse that free space

Any (Element) can remain empty.
→ space

Initial empty space is at '0' but it may change according to process



size is n then we use only 'n-1'. leaving 1 blank space.

1 blank space to ensure 1 side is beggining and other is ending.

```
enqueue (item) {
  rear = (rear+1) mod n;
  if (front == rear) {
     pf("Q is full")
  if (rear == 0) rear = n-1;
    else rear = rear-1;
    return; }
  else {
      q[rear] = item;
       return; }
    }
int dequeue() {
  if (front == rear) {
    pf("Q is empty");
      return -1; }
  else {
   front = (front+1) mod n;
     item = q[front];
     return item;
  }
}
```

f = r++] Overflow

Instant
front = rear = -1
After first enque:
rear++, front++
and after that all enque operation will increase only rear and all dequeue opn increase only front.

] f = r [Underflow]

Date :
Page No.

Gate 2012 Q. (Gate Previous Yr).

Queue is empty (front == rear)

full ~~(front == (rear+1) mod n)~~

↳ ((rear+1) mod n = front)

Gate 2004 Q (A circular linked - - - - - constant time)?

9mb Gate-1994 (Permutation possible with stack)

I/p sequence 1 2 3 4 5

o/p

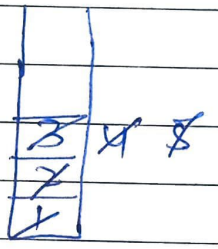
3 4 5 2 1 ✓

a) 3 4 5 1 2

b) 3 4 5 2 1

c) 1, 5, 2, 3, 4

d) 5, 4, 3 1 2

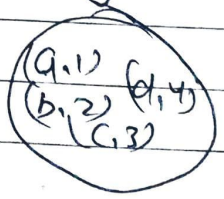


We have to generate the o/p sequence using input sequence. So, we verify to all option that they can be possible by input sequence or not.

Gate 1997 - (Implementing stack using priority queue

Insert (Q, C, K)

K → keys alloted to characters.



Gate-03 → Average lifetime of an array in stack

Date :
Page No.

for each push() or pop() operation → X
time btw each push() or pop() ——— → Y

n	2(n-1)X + 2(n-1)Y + Y
⋮	⋮
3	→ Y (lifetime)
2	→ 2(X+Y) + Y (lifetime)
1	→ 4(X+Y)+Y (lftme

~~Avg (for n) = 2(n-1)(X+Y) + Y~~

$$2\left(\sum_{i=1}^{n-1} i\right)Y + nY + \left(2\sum_{i=1}^{n-1} i\right)X$$

$$2 * \frac{n(n-1)}{2} Y + nY + \frac{2n(n-1)X}{2} \Big/ n$$

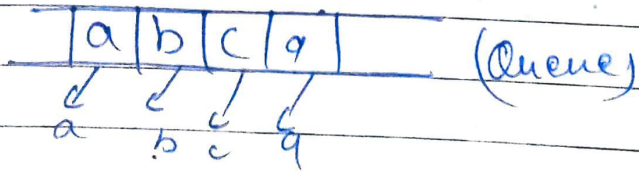
$$= n(n-1+1)Y + n(n-1)X \Big/ n$$

$$= \frac{nY + (n-1)X}{n(X+Y) - X}$$ Ans//

2006 Implementing a queue using two stacks.

Input and Output always have same order in queue.

a b c d



Date :
Page No.

Infix to postfix and Similar conversion

On Data Structure Class Notes.

Algorithm for postfix evaluation.

1. Scan the postfix string from left to right.
2. Initialize an empty stack.
3. Repeat step 4 & 5 till all the characters are scanned.
4. If the stack character is an operand, push it onto the stack.
5. If the scanned character is an operator, and if operator is unary, then pop an element from the stack. If the operation is binary, then pop two elements from stack. After popping the elements apply the operator to the popped elements. Push the result on the stack.
6. After all the elements are scanned, the result will be as last element in stack.

Time Complexity → O(n) because complete string is input as once. so all the n characters input once.

Space Complexity → O(n) no. of operand pushed is always less than total characters. So, O(n).

Graphs

Module (8)

Date :
Page No.

Introduction to Graphs

Traversal → we assume or we can reach upto each node after traversal.

Search → we may reach upto particular node but no need to traverse all the nodes.

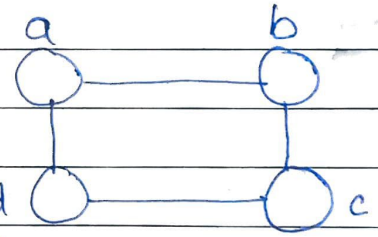
Using search we can't traversal every node.

Adjacent nodes → Nodes adjacent to particular nodes which are connect through edge.

Representations of Graph →

Two popular representation —

① Adjacency matrix (adj mat)

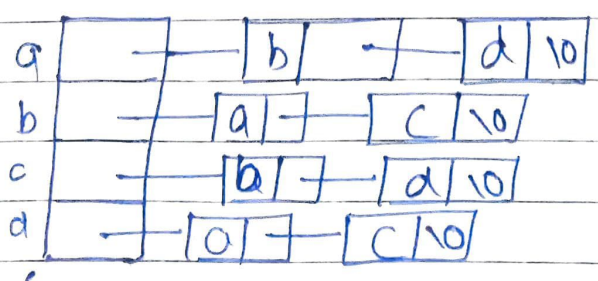


	a	b	c	d
a	0	1	0	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

for dense graphs
(If no. of Edges = $O(V^2) = E$)

space = $O(V^2)$

② Linked List



array of pointer pointed to linked list.

Sparse Graph
$E = O(V)$
Ex. facebook

space = $O(V+E)$

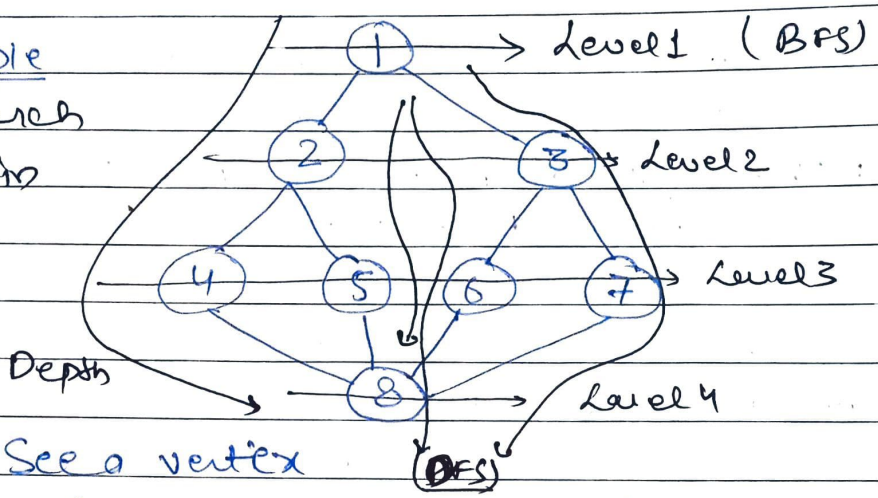
Date :
Page No.

Directed and Undirected Graph] Just remember

Introduction to BFS and DFS

Example

In BFS search level wise to DFS search depth wise.



① Visited → See a vertex

② Explored → See and explore or visit all the vertex near a node.

VISITED EXPLORED

0 0 → not visited not explored

1 0

1 1

1	2	3	4	5	6	7	8	Visited
1	0	0	0	0	0	0	0	

Queue have node not explored using BFS

→ Use stack is DFS

We keep record of visited in an array and not explored are kept in queue (BFS) and in stack (DFS).

To keep track of visited array atleast n elements block required so,

Space complexity is atleast [O(n)].

And Queue and stack both also have O(n).

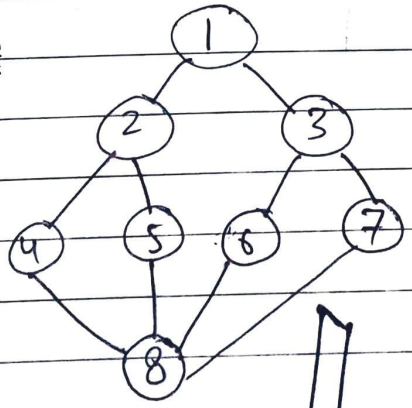
BFS Algorithm (Breadth first search)

```
BFS(v)
// The graph 'G' and array visited[] are global;
   visited[] initialised to '0'.
{ u=v ; visited[v]=1;
repeat {
for all vertices w adjacent to u do
{ if (visited[w]==0) {
      add w to q;
      visited w=1; pf(w);
  }}
if q is empty then return;
delete the next element from q; u; }}
```

Until all the nodes adj to particular node is visited.

next element means element to dequeue from q.

Ex



	1	2	3	4	5	6	7	8
u	∅	∅	∅	∅	∅	∅	∅	∅
After visit	1	1	1	1	1	1	1	1

u = X X X 4 X X 7 8

u=1 w={2,3}
u=2 w={1,4,5}
u=3 w={1,6,7}
u=4 w={2,8}
u=5 w={2,8}
u=6 w={3,8}
u=7 w={3,8}
u=8 w={4,5,6,7}

Order of element in w may be any other but enter in queue on the basis of order

Queue (q)

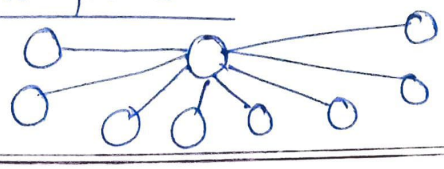
7	3	4	8	1	6	7	8

Order

1 2 3 4 5 6 7 8

(In this case)

Worst Case Scenario for BFS



Date :
Page No.

The way we visited the nodes or order may be more than one also depends on your order of access.

BFS analysis in case of linked list implementation of graph.

Space Complexity - Space are occupied in two data structures

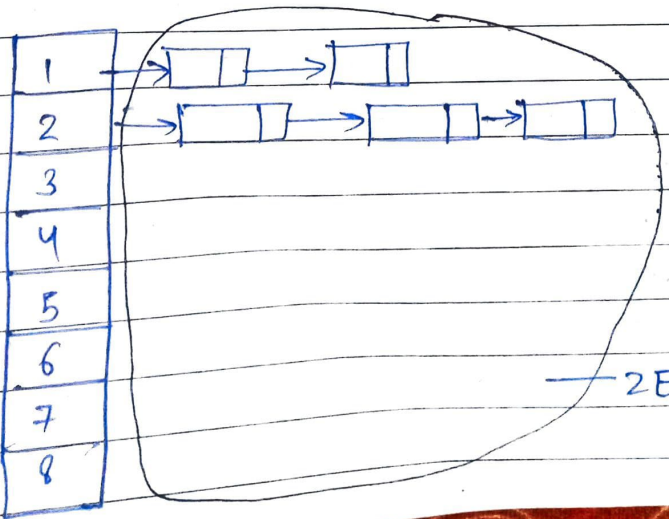
1. Array → In array all the n nodes are present so,
   Space complexity = O(n)

2. Queue → In worst it may happens that all the nodes may also available in the Queue so,

   Space Complexity = O(n)

Overall, space complexity → O(n) or O(V)

Time Complexity →



→ O(E) + O(n).
as n = V here
⇒ O(E+V)

Time required to work on 2E elements is O(E) because there are 2E elements and each element must be visited once atleast. So total no. of elements are 2E.

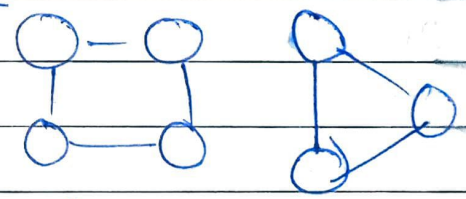
Other than this work also done in perform operation on visited[], so O(V) for that one.
V=n (no. of nodes/vertices).

Breadth first Traversal Using BFS

- Visit only reachable nodes → search
  Visit all the nodes → Traversal

If any of element in visited[] remains zero it means graph is disconnect

Ex → Disconnected Graph



[Components of Graph]

Here we built traversal using search.

for undirected graph in case of adjacency list representation the time complexity will be

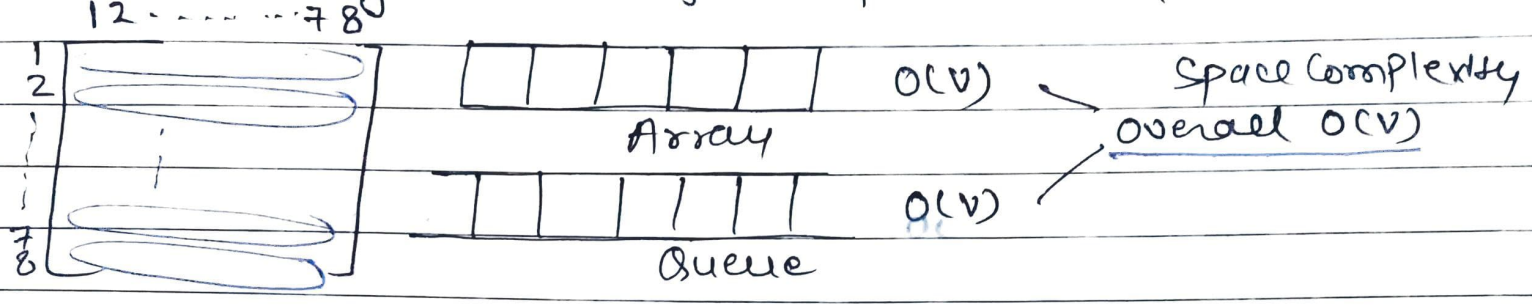
2E * Constant = O(E+V)

(2E: no. of nodes visited) (E+V: for initialization of visited[].)

space and time complexity is same

for Matrix representation:
Total n^2 elements will examined.
Space Complexity O(V).

BFS analysis on adjacency matrix representation:

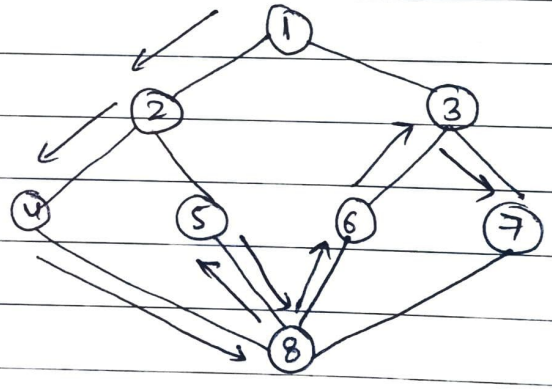


Each node is going to check for all the n nodes so, total n nodes going to check for n adjacenty list element.

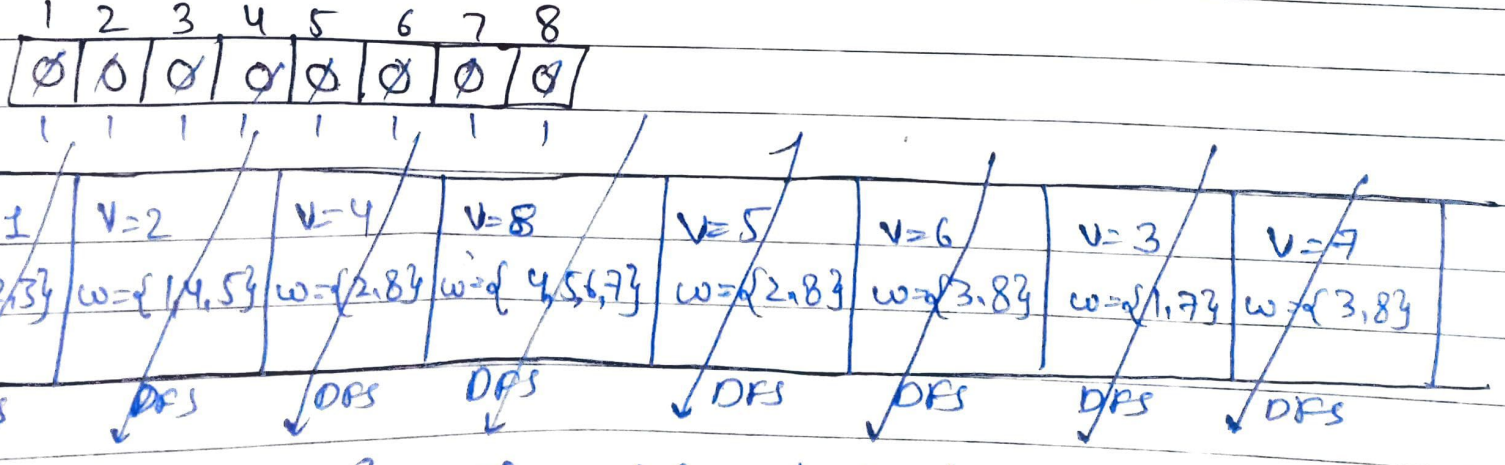
Time complexity = $O(n^2) = O(V^2)$

DFS Algorithm → It can be applied on both directed and undirected graph.

```
DFS(V){
visited[V]=1;
for each vertex w adj to V do
{
   if (visited[w]=0) then
        DFS(w);
}}
```



Order of exploration 1 2 4 8 5 6 3 7 → It may be different



Every element from stack will popped off

We are not exploring every nodes corresponds to a particular node simultaneously. We find a node that is unexplored and started exploring it.

Analysis of DFS and DFT

Space Complexity

i) for visited [] → O(V)

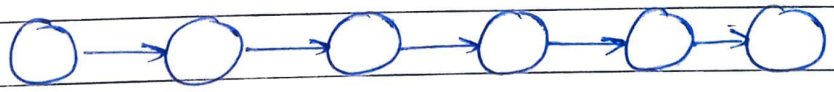
ii) for stack → In the example discussed in DFS. जैसे ही V=5 होता है उसके बाद हम वापस V=8 पर आते हैं और V=5 वाले को pop कर देते हैं।

लेकिन worst case में यह भी हो सकता है कि सारे element एक साथ stack में present हों तो उस case में,

Space ~~Time~~ Complexity = O(V)

Overall S(n) = O(V)

Worst case scenario in DFS → a chain graph.



Time Complexity.

In case of adj matric T(n) = O(V²)

list T(n) = O(V+E)

for visited array

→ 2E * C

In adj list represented explore 2E elements with constant time ⟨⟩.

Depth first Traversal is same as Breadth first traversal

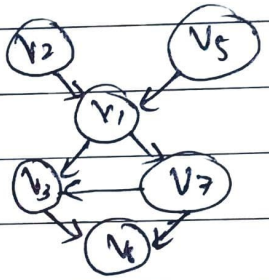
Replace BFS → DFS

Introduction to Topological Sort

(for directed acyclic graph)

i should always come before j

The order of vertices in such a way that all the dependencies are taken care.



V_2 V_5 → V_1 → V_7 → V_3 → V_6

only switch

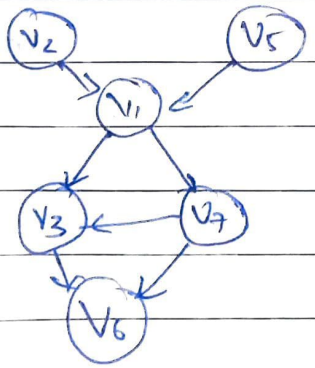
we don't chose V_3 bcoz it depends upon V_3.

Algorithm for topological sort (I)

Step-1 Identify vertices that have no incoming edge

Step-2 Delete this vertex of in degree 0 and all its outgoing edges from the graph. Place it in the output.

Ex —



Output → V_2 V_5 V_1 V_7 V_3 V_6

Time Complexity

for step-1 $O(V^2)$ bcoz in worst case it may be possible that vertices with no incoming may found at last or end. So we will traverse n vertices and we need to perform the same action for n vertices so,

$O(V) * O(V) = O(V^2)$ (worst case)

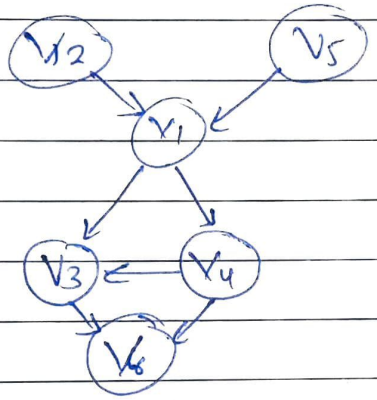
$O(V)$ in best case

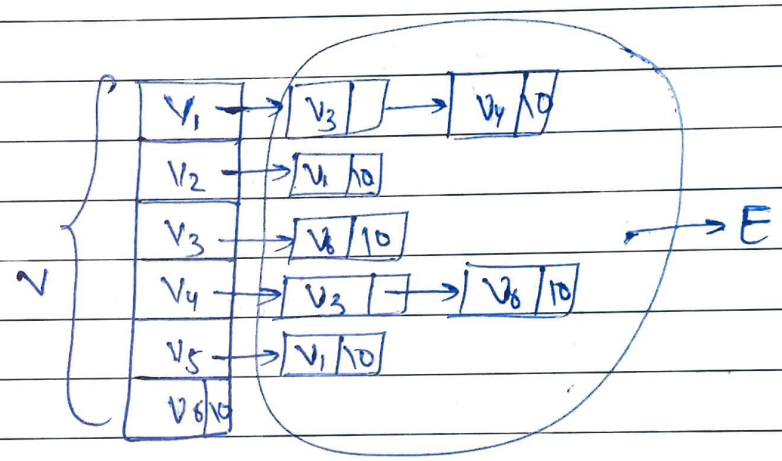
Date :
Page No.

for step-2 O(E) bcoz once you visit the edge of vertex you will delete it, so you are not going to visit it again. So all the edges visited only once.

Overall T(n) = $O(V^2+E)$ → $O(V+E)$ [By some means]

Algorithm -2





1	2	3	4	5	6
2	0	2	1	0	2
~~1~~	-1	~~1~~	0	-1	~~0~~
0		0	-1		0
-1		-1			-1

→ array for no of in degree edges.

Time taken to maintain array → O(V+E)

- for all vertex in adj linked list
- for all element of linked list

As $V_2 = -1$ then immediately V_1 becomes 1 from 2 bcoz as V_2 node deleted the edge from $V_2 \to V_1$ is also deleted So, in degree of V_1 decreases by 1.

If there are V vertices and we are searching every time till end vertices so overall we will search V^2 time. So $T(n) = O(V^2)$

But we can do it more efficiently as- taken a separate (queue), that contains the element with indegree zero.

v_2	v_5	v_1

→ As any vertex will have opt indegree zero it will add in it.

So, we have no need to search the complete array of vertices, we will just find the order of 1 as O(1).

So, $O(V^2)$ will reduce to $O(1 \cdot V) = O(V)$.

So, overall time complexity is

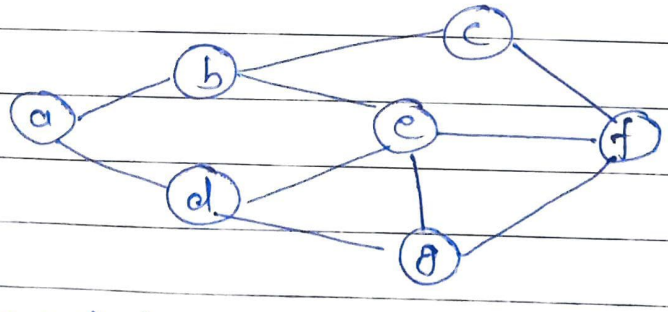
$$O(V) + O(V+E) \rightarrow \boxed{O(V+E)}.$$

If in interview ask about time complexity of topological sort.

Ans → $O(V+E)$ | Tarjan's Algorithm

GATE (2003) Question on DFS (✓)
GATE IT Question 2008 (✓)

In Graph



abefcgd is possible as

a→b (possible)
b→e (possible
e→f (possible)
f→c (possible)

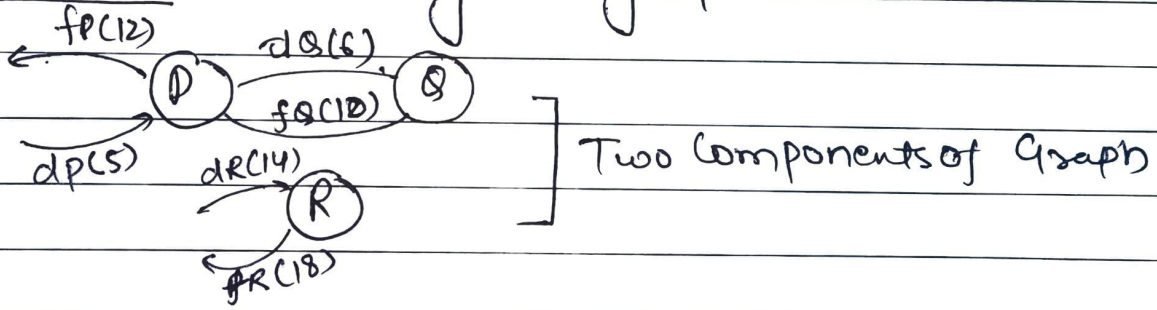
(c→g) seems as not possible but bcoz f is already visited so we poped of stack of v=e and come back to stack of v=f and then explore (f→g) and (g→d)

But in option adbcgef
(a→d) possible
(d→b) not possible bcoz we can even choose a transition from (d→a) and then (d→b) but we can't do this bcoz we have options of moving forward to e or g from d. so we don't go back to a

Difference in option (2) and (4)
In option (2) at c all the nodes b, e, f are visited but in (4) at d e, g are not visited.

Gate 2014 Depth of DFS recursion stack. (1)

Gate IT 06 Guessing the graph structure.



In undirected graph, there can be maximum $\frac{n(n-1)}{2}$ edges. We can choose any of the $\frac{n(n-1)}{2}$ edges.

So, total no. of undirected graph with n vertices is $2^{(n(n-1)/2)}$.

In undirected graph. Sum of degree of all vertices is even and no. of odd degree vertices is even.

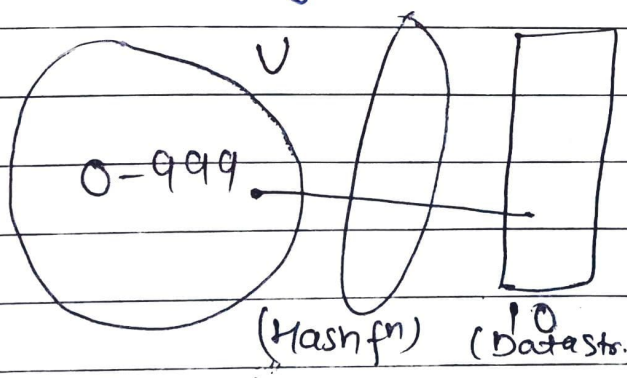
Hashing

Searching takes, Time complexity in different data structures:-

1. Unsorted array - O(n)
2. Sorted array - O(log n)
3. Linked list - O(n)
4. Binary Tree - O(n)
5. Binary Search Tree - O(n)
6. Balanced Binary Search Tree - O(log n)
7. Priority Queue - O(n)
8. Hashing - O(1)
9. Direct Address Table : similar to array.

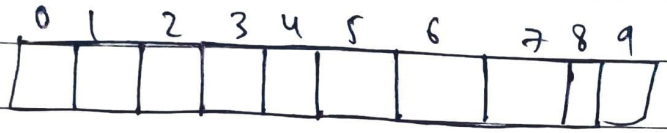
↳ Every element have keys (Index) and stored at certain index.

It is useful when no. of keys and no. of elements are very close



If we have huge no. of keys but we want to store only less amount of elements then taking the size of data structure equals to no. of keys is just a waste of space.

So, we use mapping in which we mapped the useful keys from (key set) to Data structure space using hash functions



Hash function is a function that accept any value from keys and mapped with value in Data Str.

h(0-999) ⟶ (0-9) → In above example

Ex- Let the keys to be inserted are (121, 145, 132, 999) and hash function is key (mod 10)

then

121%10 → 1 145%10 → 5 132%10 → 2 999%10 → 9

0	1	2	3	4	5	6	7	8	9
N	121	132	N	N	145	N	N	N	999

N → Null

✓ Insertion will take O(1) time.

Let us suppose you want to search 132 then 132%10 → 2 it means you will find 132 at index 2. And you will got it but if

Key = 150 ⇒ 150%10 → 0

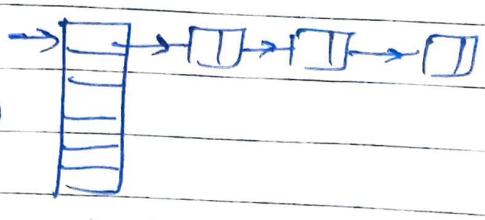
But there is a Null at 0 index. So 150 not found.

But the problem is → Collision

Soln-1 → Choose an appropriate hash function. (Not completely reduce the collision)

Soln-2 Chaining

Correspond to each index make a linked list.



We can insert more no. of same Index

(iii) Open Addressing → If the cell already occupied then recompute hash value with some minor modification

a) Linear Probing
b) Quadratic Probing
c) Double Hashing

In Chaining we waste some space in using pointer so, open addressing is more efficient
But both have equal importance.

Any Insertion and Search → open Addressing is better
Insertion, Deletion → Chaining is better.

Chaining :- We place each newly created node in the beggining.
In worst case if all the n keys inserted on a only one index then S is take O(n).
worst case deletion → O(n).

But If you have uniform distribution function then each slot will have $(\frac{n}{m})$ elements as max.
Avg search time → $O(1+n/m)$

If $n = km$ then $\alpha = k$ (constant)
⇒ time = $O(1+k) \rightarrow O(1)$

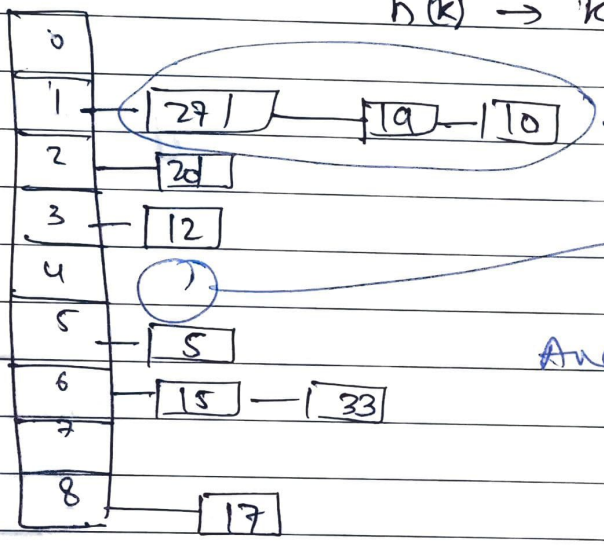
$n = km$ can be done by appropriate hash function → managing load factor

$\alpha = n/m$ called load factor.
m → total index available
n → no. of keys.

If you delete a particular node it wouldn't effect any other element but in open addressing, you need to manage all the elements.

Disadvantage → More space required for pointers.

Gate-96, 2014. Keys (5, 28, 19, 15, 20, 33, 12, 17, 10)
h(k) → k mod 9



→ max chain length = 3

→ min chain length = 0
(index → 0, 4, 7)

Avg chain length or load factor
$\alpha = n/m = 9/9 = 1$ or

$\frac{0+3+1+1+0+1+1+2+0+1}{9}$

$= 9/9 = 1$ (Both are correct)

Gate-2014

Simple Uniform Hashing → If there are n slots in hash table then, the probability of key(k) to opt any slot is

$\frac{1}{n}$.

Probability of an element to get any slot from 97 → $\frac{97}{100}$.

So,
if first 3 are inserted in any of 97 slots then, probability = $\left(\frac{97}{100}\right)\left(\frac{97}{100}\right)\left(\frac{97}{100}\right)$.