Linked list → Linear data structure that includes a series of connected nodes.
Each node stores the data & address of next node

Head → data | next → data | next → data | next → NULL

Linked list complexity:-

	Worst
Search	O(n)
Insert	O(1)
Delete	O(1)

→ Space complexity.

Singly Linked List

HEAD → data | next → data | next → NULL

Node :- struct node
{ int data;
struct node *next; }

Creation :- struct node *head;
" *one = NULL;
" *two = NULL;
" *three = NULL; } I

one = malloc(sizeof(struct node));
two = "
three = " } II

one → data = 1;
two → data = 2;
three → data = 3; } III

one → next = two;
two → next = three;
three → next = NULL;

head = one;

Doubly-Linked List

HEAD → p|d|n ⇄ p|d|n ⇄ p|d|n → NULL
NULL ←

p → previous
d → data
n → next

Node :- struct node
{ int data;
struct node *next;
struct ~~node~~ *prev; }

Creation :- I, II, III

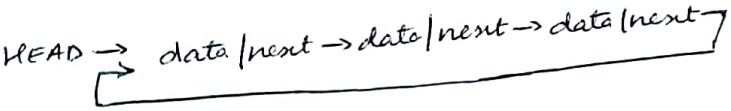
one → next = two;
one → prev = NULL;
two → next = three;
two → prev = one;
three → next = NULL;
three → prev = two;

head = one;

Circular Linked List

Creation :- I, II, III

```
one -> next = two;
two -> next = three;
three -> next = one;
head = one;
```



Sparse Matrix

→ Matrix in which no. of zeroes is more than non-zero element

Reason

→ computation time :- If stored in memory - efficient manner, computational time is saved.

→ storage :- If only non-zero elements are stored, memory/space can be saved

Represented in two ways

→ Array

→ Linked List.

Array

→ stored in 2-D array

→ Row :- stores index of row (non-zero)

→ Column :- stores index of column (non-zero)

→ Value :- consists of actual non-zero values

ex:

	0	1	2	3
0	0	1	0	0
1	0	0	0	0
2	0	5	0	2
3	9	0	0	6
4	7	0	0	0

⇒

R	C	V
0	1	1
2	1	5
2	3	2
3	0	9
3	3	6
4	0	7

Linked - List

Node consists of 4 components:-

→ Row :- index of row

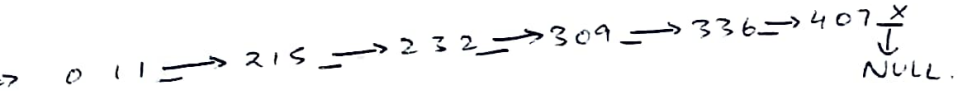
→ Column :- index of column

→ Value :- consists of actual non-zero values

→ Next node :- Pointer to store address of next connected node

ex:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 \\ 9 & 0 & 0 & 6 \\ 7 & 0 & 0 & 0 \end{bmatrix} \Rightarrow$$



∴ Array [5] [4]

0	{0, 0, 3, 0}
1	{0, 0, 5, 7}
2	{0, 0, 0, 0}
3	{0, 2, 6, 0}
4	{4, 0, 0, 0}

→ Output :-

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

Linked List

Types of Sparse Matrices

→ Lower - triangular sparse matrix

⇒ Arr i, j = 0 where i < j

ex:

1	0	0	0	0
2	2	0	0	0
1	4	3	0	0
9	8	7	1	0
1	2	7	8	9

Rowwise → {1, 2, 2, 1, 4, 3, 9, 8, 7, 1, 1, 2, 7, 8, 9}

Column wise → {1, 2, 1, 9, 1, 2, 4, 8, 2, 3, 7, 7, 1, 8, 9}

→ Upper - triangular sparse matrix

⇒ Arr i, j = 0 where i > j

ex:

1	1	2	5	8
0	2	8	9	7
0	0	3	7	2
0	0	0	1	5
0	0	0	0	9

Row-wise → {1, 1, 2, 5, 8, 2, 8, 9, 7, 3, 7, 2, 1, 5, 9}

Column wise → {1, 1, 2, 2, 8, 3, 5, 9, 7, 1, 8, 7, 2, 5, 9}

→ Tri-diagonal sparse matrix.

⇒ Arr i, j = 0, where |i - j| > 1.

on main diagonal → i = j → n

above " → i = j - 1 → n-1 } non-zero elements

below " → i = j + 1 → n-1

ex:

1	1	0	0	0
5	2	8	0	0
0	8	3	2	0
0	0	4	1	5
0	0	0	7	9

Row-wise → {1, 1, 5, 2, 8, 8, 3, 2, 4, 1, 5, 7, 9}

Column-wise → {1, 5, 1, 2, 8, 8, 3, 4, 2, 1, 7, 5, 9}

Diagonal-wise → {1, 8, 2, 5, 1, 2, 3, 1, 9, 5, 8, 4, 7}

Stack array & Linked list

Stack => Data structure that follows LIFO.
=> Used to implement algorithms like towers of hanoi & other graph algorithms.

Operations

→ Push() → Insert data
→ Pop() → Remove data
→ Peek() → Check data
→ isEmpty() → check if empty
→ isFull() → check if full
→ StackTop() → Find what is at top

Stack using array

by default, ~~at~~ top = -1. [empty].
top pointer is used to perform operations.

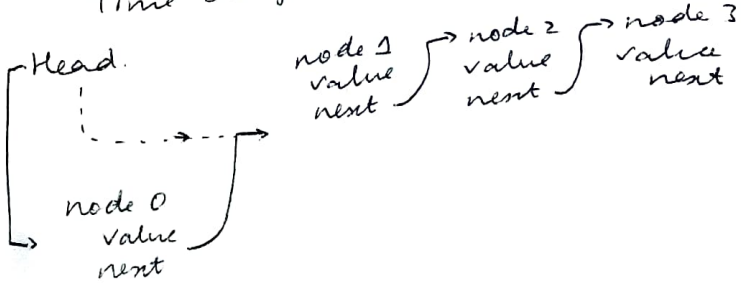
Pseudo code

1. Start
2. ~~class Stack~~
3. pointer Top.
4. Initialize array
5. Constructor top = -1 (empty)
6. Push() if top == full ; else increment top pointer.
7. Pop() if top == -1 ; delete & decrement top pointer.
8. isEmpty() ~~if~~ ; P == -1 ~~or~~ top < 0
9. Display()
10. End.

Stack using Linked-List

~~class Node~~
& ~~public~~ ~~int data;~~

Push → Create node & allocate memory.
→ Must be added at beginning (to not violate property)
→ Make new node as start node
Time complexity → O(1).



Pop → check for underflow condition.
(when pop from empty stacks).
→ Adjust head pointer accordingly.
→ value stored in head pointer is deleted
→ ~~Head node~~ Next node of head node becomes head node.

Time complexity → O(n).

Display → copy head pointer to temporary pointer.
→ Move the pointer through all the nodes and display.

Time - complexity → O(n)

Queue Array & Linked List

Queue → ADT that follows FIFO.

Enqueue → Adds element to queue (to tail).

Dequeue → Returns & deletes front element of queue.

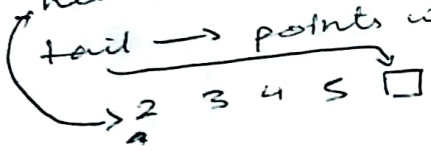
isEmpty → checks if empty.

isFull → checks if full.

Front → Returns the front element of queue.

Queue using Array

head → points to oldest element
tail → points where new element will get added } pointers



∴ IS_EMPTY (Q)
if Q.tail = Q.head → True / False is returned.

∴ IS_FULL (Q)
if Q.head = Q.tail + 1 → True / False is returned.

∴ Enqueue (Q, x)
Q[Q.tail] = x
if Q.tail = = Q.size
Q.tail = 1
else Q.tail = Q.tail + 1.

} provided, not overflow.

∴ Dequeue (Q, x)
x = Q[Q.head]
if Q.head = = Q.size
Q.head = 1.
else Q.head = Q.head + 1.
return x

} provided, not underflow.

Queue Using Linked List

→ will never overflow.

→ if empty, head → NULL.

∴ IS_EMPTY (Q)

if Q.head = null → true/false is returned.

∴ ENQUEUE (Q, n)

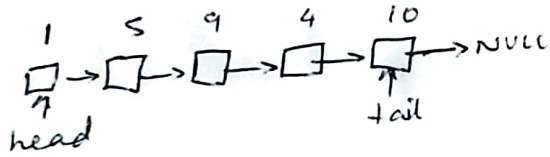
Q.tail.next = n
Q.tail = n

} if empty through, head & tail = n.

∴ DEQUEUE (Q, n)

x = Q.head.data
Q.head = Q.head.next
return x

} provided not underflow.



Double ended Queue & Priority Queue

Double ended Queue aka dequeue

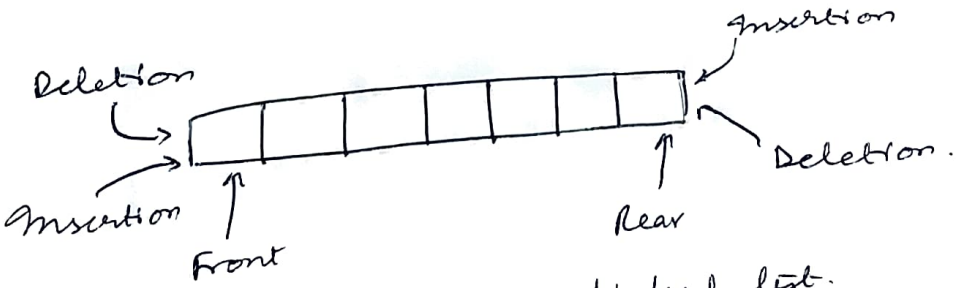
→ Generalized version of queue D.S. that allows both insert/delete at both ends.

operations

insertFront() → Add in front
insertLast() → Add in last
deleteFront() → delete in front
deleteLast() → delete in rear.
getFront() → gets the
getRear() → gets the
isEmpty() → Checks
isFull() → checks.

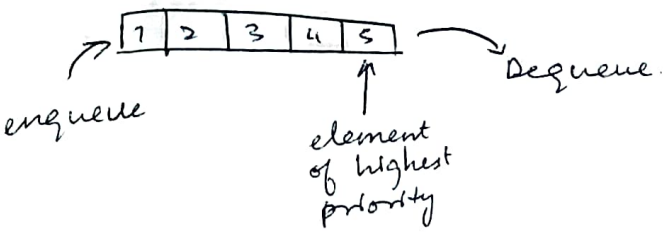
Applications

→. Supports both stack & queue.
→ Supports both clockwise & anti clockwise rotation.
→ O(1) time
→ Problems where elements need to be added or removed from both ends.
→ doubly linked list or circular array.



Also called head-tail linked list.

Priority Queue



Extension of Queue where,

1. Every item has a priority associated to it.
2. High priority is dequed first.
3. If same priority, executed wrt order.

Initial Queue = { }

operation	Return Value	Queue Content
Insert (C)		C
Insert (O)		C O
" (D)		C O D
Remove max	O	C D
Insert (I)		C D I
Insert (D)		C D I N
Remove max	N	C D I
Insert (G)		C D I G.

Operations

Insert (Item, priority)
getHighest Priority ()
delete Highest Priority ()

Can be Implemented using

array
Linked list
Heaps.

Applications

→ Dijkstra's shortest path algorithm. (min efficiency)
→ Prim's algorithm. (key of nodes)
→ Data compression. (Huffman coder
→ A.I.
→ Heap sort.
→ Load Balancing.

Postfix, Prefix & Infix

Notations :- way to write arithmetic expression.

There Notations:-

→ Infix
→ Prefix
→ Postfix.

Infix

=> operators are between operands
=> Algorithm to process Infix is difficult & costly in terms of time/space consumption. (for computers)

ex: a - b + c

Prefix

=> operator is written ahead of operand.
=> known as polish notation.

ex: +ab → a + b.

Postfix

=> operator is written after operand.
=> known as reverse polish notation.

ex: ab + → a + b.

ie,

infix	prefix	postfix
~~a/b+c/d~~	~~+/ab/cd~~	~~ab/cd/+~~
a/b + c/d	+/ab/cd	ab/cd/+
(a+b)* (c+d)	*+ab+cd	ab+cd+*

Precedence & Associativity.

operator	Precedence	Associativity
^	Highest	Right
* & /	second	left
+ & -	Last	left

ie, in a+b*c
→ b*c is evaluated first.

If a+b must be evaluated first,
=> (a+b)*c.

Prefix

=> +9*26.

characters	Stack	
6	6	push.
2	6 2	push.
*	12	pop, multiply, push.
9	12 9	push.
+	21	pop, add, push.

Result :- 21

complexity → O(n).

Post fix

pop twiced.

=> 456*+

character	Stack	
4	4	push.
5	4 5	push.
6	4 5 6	push.
*	4 30	pop, multiply, push.
+	34	pop, add, push.

Result :- 24

complexity → O(n)

Towers of Hanoi

=> Math puzzle of 3 rods & n disks.

=> Move entire stack to another rod provided,

1. 1 disk can be moved at a time
2. Disk can only be moved if it is the uppermost disk.
3. disk can't be placed on top of smaller disk.

Example

2 disks.

rod 1 → A rod 2 → B rod 3 → C.

1) A to B (1st disk)
2) A to C (2nd)
3) B to C (3rd).

ie, n-1 from A to B
last disk from A to C
n-1 from B to C.

=>

∴ no input → 2.

output => Disk 1 moved from A to B
" 2 " " A to C.
" 1 " " B to C.

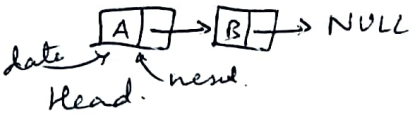
```
void towers(int n, char x, char y, char z)
{ if (n>0)
{ towers(n-1, x, z, y);
  printf("\n%c to %c", x, y);
  towers(n-1, z, y, x); } }

int main()
{ int n = 3;
  towers(n, 'A', 'B', 'C'); }
```

Dynamic Data structure

→ Size of structure isn't fixed.
It facilitates change of DS in run time.

ex: Linked list.



→ They are more flexible.

Cursor Implementation

→ It is the linked list representation of array.
→ It is used when linked list required but pointers aren't available.
→ Used in BASIC & FORTRAN.

It simulates
→ Data stored in a collection of structures.
→ Obtaining structure from global memory. (malloc, free).

Declaration for cursor implementation of linked list.

Slot	Element	Next
0	?	1
1	?	2
2	?	3
3	?	0

```
{ josephus(n-1, k) + k-1) % n+1; }
```

Josephus problem

can be explained with an example :-

In a group of 6,
if we start with 1,
after one round, 1, 3, 5 survive.
In next round, 1, 5 survive.
next round 5 survives.

ie, In a circle, in clockwise/anti direction, k-people are skipped and next is killed till 1 remains.

here, k → 2nd person