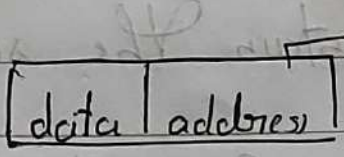
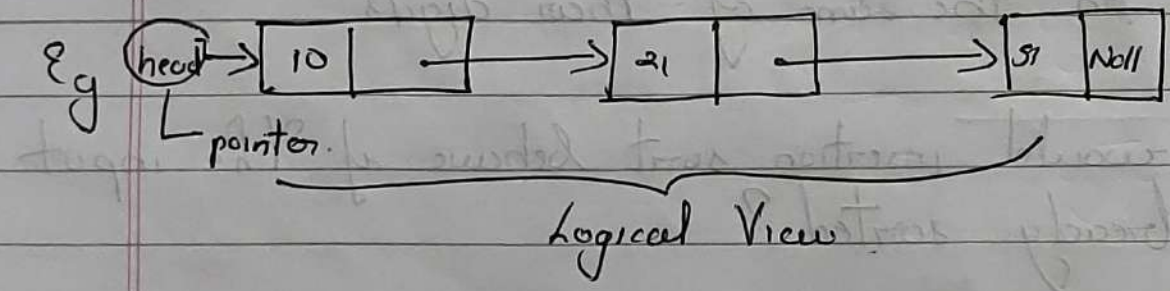
Linked lists

- Linear
- Dynamic
- Heterogenous.

- It is a series of connected nodes.
- Each entity in a linked list is called a node.



node



The address section contains the address of the next node.

Note ADT. (Abstract Data Type)

This is for the user. We just know the type and the operations which can be performed on it.

Data Structure:-

This is for the developer. Here we focus on implementation.

head pointer:-

This contains the address of the first node. If this is empty (null) then the linked list is empty.

→ Linked list can be implemented with 2 pointers i.e.

↳ First ptr → pointing to first node (head)
↳ Second ptr → pointing to last node (last).

Note:- If head and last are empty i.e NULL then the list is empty. Primarily head is checked to see if list is empty.

Note: If both head and last point to the same node then there is only one node in the list.

Pointer

```
datatype * identifier;
int *n;
int num=10;
n= &num;
```

Dynamic Allocation

i malloc - allocates mly
ii calloc - allocates mly and initializes with 0
iii realloc

Syntax:

malloc (size of mly)
or
malloc (size (int))
↳ allocate 4 bytes
malloc (2* size (int))
↳ allocates 8 bytes

Runtime mly is created in heap
Compilation mly is allocated in stack.

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

Structures

```
struct structure_name
{
```

typedef → Look Up

Date
Page

data members;
};

Create objects - struct struct_name obj_name;
Access item - obj.variable_name;

If we create pointer objects then → is used to access data members.

Eg:- struct student *s;
s→name;

* Array variable contains the starting address.

#include <stdlib.h>
↳ When using malloc.

Representation of LL

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

Create a list with 2 pointers

head → pointing to 1st node
last → pointing to last node

Condition for empty list = head = null

```
(struct node *) createANode (int val)
    {
        struct node * new;
        new = (struct node *) malloc (sizeof (struct node));
        new -> data = val
        new -> next = NULL
        return new
    }

createAList (int val, struct node *head, struct node *last)
    {
        struct node * new = createANode (val)
        if (head == NULL)
            head = new
        else  last -> next = new
            last = new
        return last
    }
```

Operations of LL
- Traversal, Search, Delete, Sort, Insert

Traversal

```
temp = head;
while (temp != NULL)
{ printf (temp -> data)
  temp = temp -> next
}
```

Search

```
Search (struct node *head, int key)
{ struct node *temp = head
  while (temp != NULL)
    {
      if (temp -> data == key)
        { printf ("Search Successful");
          return true
        }
      temp = temp -> next;
    }
  return false;
}
```

a) Write the algol pseudocode for counting the no of nodes in the list.

b) To display the odd position node values

c) To display the even position node values

d) Find the middle node value in the list

e) Search for a particular value, ~~print~~ return the next node value if its present else print the last node value

a)

```
Count ( struct node *head)
 { struct node *temp = head; int c=0;
   while (temp ~~->next~~ != null)
    {  c++;
       temp = temp -> next;
    }
   return c;
 }
```

b)

```
Odd ( struct node *head)
 {
  struct node * temp = head;
  int c=0;
  while (temp != null)
   {
      c++
```

```
temp= temp if (c%2 != 0)
                printf ("%d", temp -> data);
            temp = temp -> next;
        }
    }
```

c
```
Even (struct node *head)
{
    struct node *temp = head;
    int c = 0
    while (temp != null)
    {
        c++.
        if (c%2 == 0)
            printf ("%d", temp -> head);
            temp = temp -> next;
        }
    }
```

d
```
Middle (struct node *head)
{
    struct node * temp = head;
    int c=0
    while (temp != null)
    { c++; temp = temp -> next;
```

```
        int mid = c/2;
        temp = head;
        c=0; printf("Middle node is %d", mid);
        while (temp != null)
    }   {
            c++;
            if (c == mid)
                print
{
e   Search (struct node *head, key)
    { struct node *temp = head; int c=0;
      while (temp != null)
        {
          if (temp -> data == key)
          {
            if (temp -> next != NULL)
              print (temp -> next -> data);
            else
              print (Nil print (temp
              print (temp -> data)
          }
        temp = temp -> next
      }
    }
```

Insert

Basic operations here is link change.

Scenarios

I Insert a new node at the beginning

```
InsertToTheFirst(struct node *head, int val)
{
  struct node *new = CreateANode(val)
  new -> next = head
  head = new
  return head
}
```

II Insert a node at the end

```
InsertToTheEnd(struct node *head, int val)
{
  struct node *new = CreateANode(val)
  struct node *temp = head
  while (temp -> next != NULL)
    {
```

```
        temp = temp -> next
    }
temp -> next = new
}
```

III Insert After a particular Position :-

```
InsertAfterPos (struct node *head, int val, int i)
{
struct node *new = CreateANode (val)
struct node *temp = head
while (temp -> data != key)
  {
    temp = temp -> next
  }
new -> next = temp -> next
temp -> next = new
}
```

IV Insert Before a Particular Data

```
InsertBefor (struct node *head, int val, int
             key)
{
  struct node *new = CreateANode (val)
```

```
struct node * temp : head
while (temp -> next -> data != Key)
        temp : temp -> next
new -> next : temp -> next
temp -> next : new
}
```

V Insert At a Particular Position

```
Insat ToPos (struct node *head, int val, int pos)
{  int c=0
   struct node * new = CreateANode (val)
   struct node *temp : head
   while (temp -> data next != NULL)
     {
       CTR;
       if (c == pos-1)
       {
          temp -> next = new;
          new -> next = temp -> next
          temp -> next : new;
        return; }
       else
          temp : temp -> next;
     }
}
```

Deletion

I Delete the first node

```
deleteFirst (struct node *head)
{
    struct node *del = head
    head = head->next
    value = del -> data
    del -> next = NULL
    free (del)
}
```

II Delete the last node :-

```
deletelast (struct node *head)
{
    struct node *temp = head
    struct node *del
    while (temp->next->next != NULL)
    { temp temp -> next
      del
      temp = temp->next
      value = del -> data
      temp -> next = NULL
      free (del)
      return value
}{
```

III To Delete a Node After a Node

```
DeleteAfter ( struct node *head, int key )
{
struct node *temp = head;
while (temp -> data != key)
  {
     temp = temp->next
  }
struct node *del = temp->next
  temp->next = del->next
  del->next = NULL
  value = del->data
  free(del)
  return value
}
```

IV To delete a Node Before a Node

```
Delete Before (struct node *head, int key)
{
struct node * temp = head;
while (temp -> next -> data != key)
  {
     temp = temp -> next
```

```
}
struct node * del = temp -> next
temp -> next = del -> next
del -> next = NULL
value = del -> data
free (del)
return value
}
```

V To delete a node in a particular position.

```
deleteFromPos (struct node *head, int pos)
{ int c=0
  struct node *temp = head;
  while (temp -> next != NULL && c < p-1)
  { c++;
    temp = temp -> next
  }
  struct node *del = temp -> next
value = del -> value temp -> next = del -> next
        free (del)
  return value
}
```

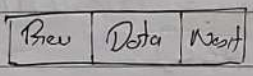
Typedef function

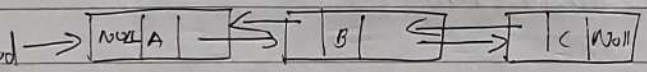
typedef struct node node;

Limitations of Singly linked list

→ Traversal is possible in only one direction

Double linked list





- logical View

```
struct node
{
    int data;
    struct node * next;
    struct node * previous;
};
```

To create a Node

```
struct node * createANode (int val)
{
```

```
    struct node *new;
    new = (struct node *) malloc (sizeof (struct node));
    new -> dat = val;
    new -> next = NULL;
    new -> previous = NULL;      return new;
}

struct node * createDlist (int val, struct node *head, struct node *last)
{
    struct node *new = createANode (int val);
    if (head == NULL)
        head = new
    else
        last -> next = new
        new -> prev = last
    last = new
    return head
}
```

To Traverse

```
Traverse Forward (struct node *head)
{
    temp = head.
    while (temp != NULL)
```

```
    {
        printf (temp->data)
        temp = temp->next
    }
}
```

```
Traverse Backward ( struct node *last)
{   struct node * temp;
    temp = last
    while (temp != NULL)
    {
        printf (temp->data)
        temp = temp->prev
    }
}
```

3. Write the algorithm for a function which displays a list in backward direction where we assume that we have a pointer p pointing to the first node

```
Back Traversal (struct node * p)
{   struct node * temp;
    temp = p
    while (temp->next != NULL)
    {
        temp = temp->next
    }
```

```
while (temp != NULL)
{
    printf (temp -> data)
    temp = temp -> prev
}
}
```

Insertion

Insertion Before A Node

- Traverse upto the given node

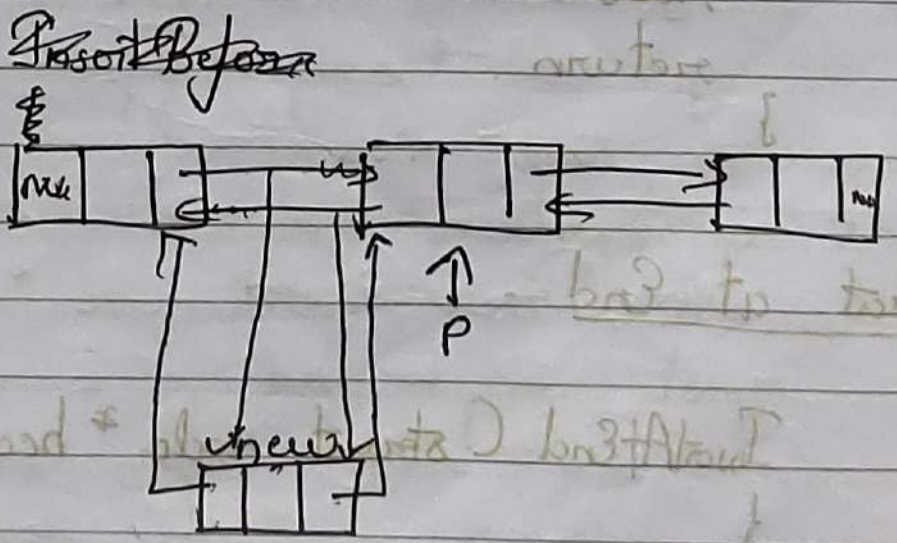
```
InsertBefore (int key, struct node * head) { struct node *temp;
{ temp = head
  while (temp -> next -> data != key)
      temp = temp -> next
      new -> next = temp -> next
      temp -> next = new
      new -> next -> prev = new
      new -> prev = temp
}
```

classmate
Date
Page

- Traverse ~~till before~~ upto the given node

```
InsertBefore (int key, struct node *head)
{
    temp = head
    while (temp->data != key)
        temp = temp->next
    new -> next = temp
    new -> prev = temp -> prev
    temp -> prev -> next = new
    temp -> prev = new
}
```

a. Assume that a pointer p is pointing to a node in a DLL. write the statements to insert a node before p.



```
new->next = p
p->prev->next = new
new->prev = p->prev
p->prev = new
```

Steps : 1 : first change new pointer
2 : change prev
3 : change next

II Insert a new node at beginning

```
InsertToBegining (struct node *head, int val)
{
    struct node *new = createANode(val);
    new->next = head;
    head->prev = new
    head = new
    return
}
```

III Insert at End :-

```
InsertAtEnd (struct node * head, int val)
{
    struct node * new = createANode(val);
```

```
struct node * temp = head
while (temp -> next != NULL)
    temp = temp -> next
temp -> next = new
new -> prev = temp
}
```

IV Insert To a Particular Position

```
InsertToPos ( struct node * head, int val, int pos)
{
  int c=0;
  struct node * new = CreateANode (val)
  struct node * temp = head
  while ( temp -> next != NULL)
   {
     c++;
     if (c==pos-1)
     {
        new -> next = temp -> next
        temp -> next = new
        new -> prev = temp
        return
     }
  else
```

```
        temp = temp -> next
    }
}
```

Deletion

I Delete first Node :-

```
deleteFirst (struct node *head)
{
    struct node * dd = head
    head = head -> next
    head -> prev = NULL
    value = dd -> data
    dd -> next = NULL
    free (dd)
}
```

II Delete the last Node

```
deleteLast (struct node *head)
{
    struct node * temp = head
    struct nod * dd -> next
    while (temp -> next != NULL)
```

```
{
  temp: temp -> next
}
dd = temp -> next
value = dd -> data
temp -> next = NULL
del -> prev = NULL
free(dd)
return value
}
```

III To delete a Node After a Node:

```
DeleteAfter(struct node *head, int key)
{
  struct node *temp = head;
  while(temp -> data != key)
    temp = temp -> next
  struct node *dd = temp -> next
  temp -> next = dd -> next
  del -> next -> prev = temp
  del -> next = NULL
  value del -> prev = NULL
  value = del -> data
```

```
    free(del)
    return value
}
```

IV To Delete a Node Before a Node

```
DeleteBefore(struct node *head, int key)
{
    struct node *temp = head;
    while(temp->next->data != key)
    {
        temp = temp->next
    }
    struct node *del = temp->next
    temp->next = del->next
    del->next = NULL
    value = del->data
    del->prev = NULL
    temp->next->prev = temp
    value = del->data
    free(del)
    return val
}
```

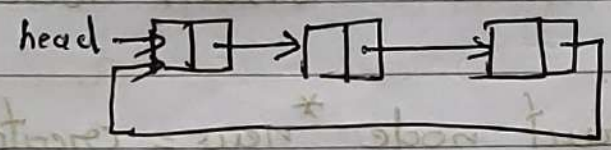
V To delete a node in a particular position

```
deleteFromPos(struct node *head, int pos)
{
    int c=0;
    struct node *temp = head
    while(temp->next!=NULL && c<pos)
    {
        c++;
        temp = temp->next
    }
    struct node *del = temp->next
    value = del->value
    temp->next = temp->next->next
    del->next->prev = temp
    del->prev = NULL
    free(del)
    return value
}
```

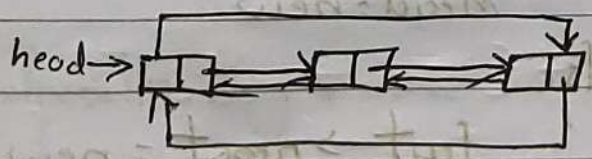
Circular Linked list:-

Two Types:-

i) Circular SLL
ii) Circular DLL



Circular SLL



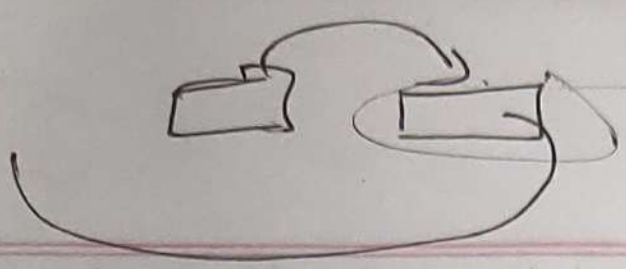
Circular DLL

Create a Node

```
createANode (int val)
{
  struct node *new;
  new = (struct node *) malloc (sizeof (struct node));
  new -> data = val
  new -> next = NULL
  return new
}
```

Create a list (Circular SLL)

```
create CSLL (int val, struct node *head,
             struct node *last)
{
struct node *new = createANode(val);
if (head = NULL)
    head = new
else
  { last -> next = new
    last = new
    last -> next = head
  }
```



}

Traversal :-

```
Traversal ( struct node * head)
{
  struct node * temp = head
  do
  {
    if (temp == NULL) {
      print ("Empty List");
      break;
    }
    print (temp->data);
    temp = temp->next
  } while (temp != head);
}
```

Insertion

```
                                    struct node *list
InsertAtBeg ( struct node * head , val)
{
  struct node * new = create ANode (val)
  new -> next = head
```

head = new
last = ~~head~~ next = head
}

Insert At last

```
InsertAtlast(struct node *head, int val)
{
struct node *new = CreateANode(val)
struct node *temp = head;
while (temp -> next != head)
{
    temp = temp -> next;
}
temp -> next = new;
new -> next = head;
}
```

Deletion

```
DeleteFirst(struct node *head)
{
struct node *del = head
struct node *temp = head
```

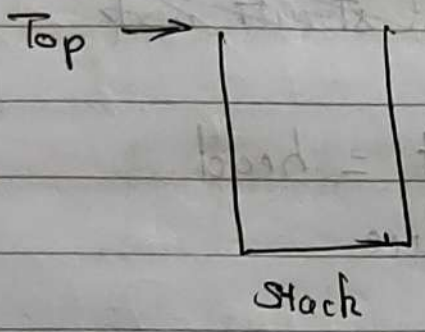
```
head = head -> next
value = del -> data
del -> next = NULL
free (del)
while (temp -> next != head)
    temp = temp -> next
Delete last (struct node

temp -> next = head
return value
}

DeleteLast (struct node *head)
{
    struct node *del;
    struct node *temp = head;
    while (temp -> next -> next != head)
        temp = temp -> next
    temp -> next = temp -> next
    del = temp -> next
    temp -> next = head
    free del -> next = NULL
    value = del -> data
    free (del)
    return value
}
```

classmate
Date
Page

Stack

Linear, Dynamic, Heterogeneous, LIFO

Implementation — Array
Linked List



Top is a pointer pointing to Top of Stack.

Insertion - Push
Deletion - Pop

Push Algo

Increment Top
Add Value to Top

When the stack is full then it is termed as overflow.

Overflow Condition :- Top = size.

During push operations, we always check if the stack is full or not and the insert the data.

~~PUSH~~

If the stack is empty, we call that situation as underflow and the condition is top = -1 (array implementation).

Peek Operation → returns the TOP value.

Applications of Stack

- Infix Expression → Postfix
- Evaluation of Postfix
- String Validation
- Reverse your name using stack

Assignment

Applications of Stack :-

I Infix To Postfix Convention :-

→ Infix notation is the standard arithmetic expression where arithmetic expression has operators written between operands.

→ Eg :- A + B (Infix)

→ Postfix notation a.k.a Reverse Polish Notation writes the operators after the operands.

→ Eg AB +

The stack data structure can be used to convert infix expression to postfix.

Algorithm

```
int precedence (char op)
{
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}

InfixToPostfix (char * infix, char * postfix)
{
    Stack stack;
    int k=0;

    for(int i=0 ; infix[i] != '\0'; i++)
    {
        char ch = infix[i];

        if (isalnum (ch))
            postfix[k++] = ch;

        else if (ch == '(')
            push (&stack, ch);
```

```
else if (ch == ')')
{
    while (!isEmpty(&stack) &&
           top(&stack) != '(')
    {
        postfix[k++] = pop(&stack);
    }
    pop(&stack);
}
else {
    while (!isEmpty(&stack) && precedence
    (top(&stack)) >= precedence(ch))
    {
        postfix[k++] = pop(&stack);
    }
    push(&stack, ch);
}
}
while(!isEmpty(&stack))
{
    postfix[k++] = pop(&stack);
}
postfix[k] = '\0';
}
```

II Evaluation of Postfix :-

→ Evaluation of a Postfix expression is also possible using a stack data structure.

Algorithm

```
int evaluatePostfix (char *exp)
{
    struct Stack * stack = createStack(strlen(exp));
    int i;

    for(i=0; exp[i]; ++i)
    {
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);

            switch (exp[i])
            {
            case '+': push(stack, val2+val1);
```

```
                break;
            case '-':
                push(stack, val2 - val1);
                break;
            case '*':
                push(stack, val2 * val1);
                break;
            case '/':
                push(stack, val2 / val1);
                break;
        }
    }
}
return pop(stack);
}
```

For each operator, two operands are popped from the stack and the result is pushed back onto the stack

Finally the result is the only value left in stack.

III String Validation

→ String Validation is a common application of stacks, especially when checking for balanced symbols like parenthesis (), brackets [] or braces {}. The idea is to use a stack to ensure that every opening symbol has a corresponding and properly nested closing symbol

Explanation:

→ Open symbols are pushed onto the stack

→ Close Symbols are matched against the TOS
  ↳ If symbol on top of stack is the matching opening symbol, it is popped off
  ↳ If not, the string is invalid

Algorithm

```
ValidateString (char *str)
{
    int len = strlen(str);
    for (int i=0; i<len; i++)
```

```
{
    if (str[i] == '(' || str[i] == '{' || str[i]
    == '[')
    {
        push (str[i]);
    }
    else if (str[i] == ')' || str[i] == '}' ||
             str[i] == ']')
    {
        char topChar = pop();

        if (! isMatchingPair (topChar, str[i]))
        {
            return false;
        }
    }
}
```

IV Reverse Name Using Stack

→ To reverse a string using a stack, we can follow the typical stack operations. First push all the characters of the string onto the stack, then pop them off, which will give the reversed order.

Algorithm

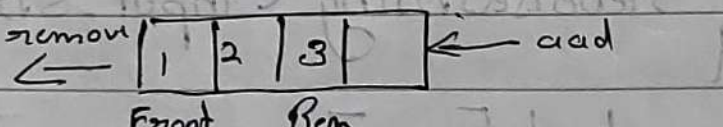
```
reverseString (input_string):
{
    stack [MAX]
    top:-1

    for(i:0 ; i < length(input_string)-1 ; i++)
    {
        top = top+1
        stack[top] = input_string[i]
        reversed_string = ""

        while(top >= 0)
            reversed_string = reversed_string
                + stack[top]
```

top = top - 1
return reversed_string

1/7/19

Queue

→ It is Linear, Dynamic, Heterogenous
→ It has two pointers - rear, front
entry exit



Operation

i) → Dequeue - Deletion
ii) → Enqueue - Insertion

* If front = -1 the queue is empty → underflow

Insertion when queue is empty

↳ if (front == -1) then

increment front

* If rear = size-1 then queue is full → overflow

Enqueue Algo

```
Enqueue
{
  if (Rear = size-1)
    print ("Overflow")
  else
  { increment rear
    add value to rear
    if (front = -1)
      increment front
  }
}
```

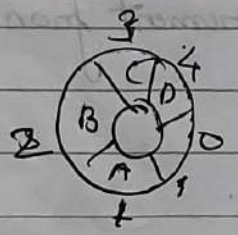
Dequeue Algo

```
Dequeue
{
  if (front = -1)
    print ("Underflow")
  else
```

```
{
    capture front value
    if (front = rear)
        front = -1
        rear = -1
    else
        ~~front~~ increment front
}
```

Enque using LL - Insert at End
Deque using LL - Delete the first node

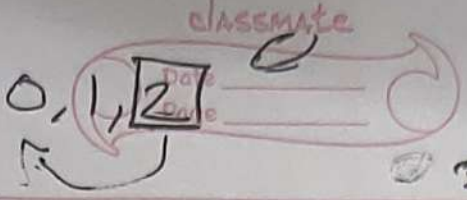
Circular Queue



Enque Algo

```
Enqueue
{
    if front = (Rear + 1) % size)
        print ("Overflow")
```

$2 \% 3 = 2$



$(2+1) \% \text{size} = 0$

```
else
{   rear = (rear+1) % size
    add value to rear
    if (front==-1)
        increment front
}
```

Dequeue Algo

```
Dequeue
{
  if (front==-1
    print("Underflow");
else
  {  val = arr[front]
     if (front == rear)
          front=-1
          rear=-1
     }
     else
     {  front = (front+1)%size;
     }
```

Assignment

Priority Queue

Priority Queue is a type of queue that arrange elements based on priority values.

Elements with higher priority values are typically retrieved before elements with lower priority.

Properties

→ Every element has a priority
→ Higher priority elements are dequeued first
→ Enqueue and Dequeue are priority based
→ If two elements have same priority, they are served according to their order in queue

Operations

→ Insertion

When an element is first inserted it is moved to an empty slot. Then from there elements are swapped till they are in the right position.

→ Deletion:

Remove and return the element with the higher priority (max priority queue) or the lower priority (min priority queue).

Type of Priority Queue

1. Ascending Order Priority Queue

Elements with lower priority is given higher priority in the priority list.

2. Descending Order Priority Queue

Elements with higher priority value is given higher priority in the priority list

Implementation:

→ Array
→ Linked list.
→ Heap → (Main)

24/9

classmate
Date
Page

Deque (Double Ended Queue)

Values can be inserted on both front or rear.

```
EnqueueFront()
{
    if (front == 0)
        print("Can't add elements to the front");
    else
        if (front == -1)
            front = 0
            rear = 0
        else
            decrement front
            add value to front
}

DequeueRear()
{
    if (front == -1)
        print("Underflow");
    else
        remove value from rear
        decrement rear
}
```

Input Restricted
Output Restricted
] Types of Dequeue

Operations :- 4

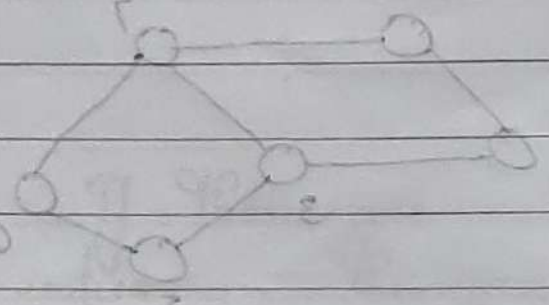
→ Enque - Front, Rear
Dequeue - Front, Rear

Input Restricted

↳ Input can be from only one end

Output Restricted

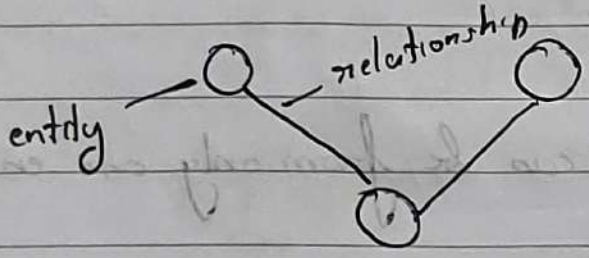
↳ Dequeue can be from only one end.



Module 4

Graph

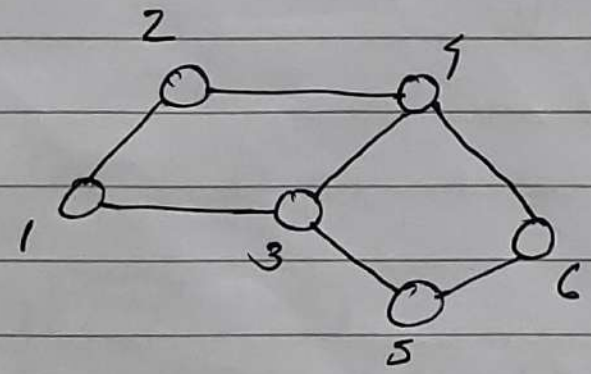
Collection of nodes that contain data and are connected to other nodes



↳ in case of social networks

A graph is a data structure (V, E) that consists of a set of vertices V and a collection of edges E represented as ordered pairs of vertices (u, v)

Eg



V = { 1, 2, 3, 4, 5, 6 }

E = { (1,2), (2,4), (1,3), (4,6), (6,5), (3,5), (4,3) }