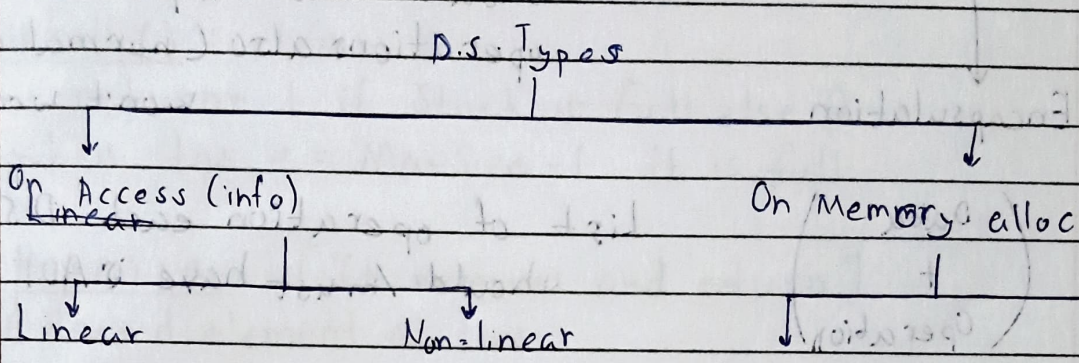
Chapter : 1 Introduction to DS

1) What is data structure?

→ It is study of various structure use to store an access information / data.

Basic Types :- (5 mks)



1) Sequential access - Direct or guided access
2) Easy to code (+ve) - Too complex to code (-ve)
3) Slow in access (-ve) - Faster in access (+ve)

e.g. Stack, queue, L.L — e.g. Tree, Graph (G, D)

Static	Dynamic
1) Memory allocation at compile time.	1) Memory allocated at run time
2) Once allocated can't ↑ or ↓	2) Can ↑ or ↓ on demand
3) Memory given on prediction, hence not efficient.	3) Memory given on demand hence more efficient.
4) Easy to code. int a[100]	4) Complex due to commands used in giving memory malloc (10 * size of(int)) free(ptr)

ADT

Abstract Data Type → D.S are complex

↓

when needed to implement we should/must implement operations also (normal operation won't work)

↓

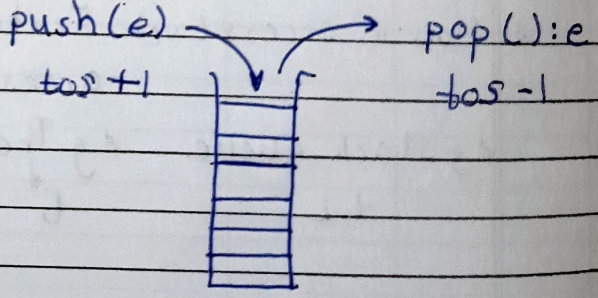
Encapsulation

(Data + Operation) — List of operation each D.S should/must have is ADT

→ ADT

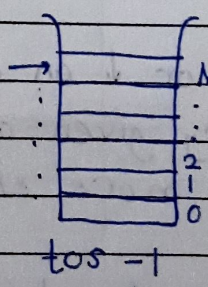
List of operations to be coded to implement DS

→ Stack (static)
- Linear
- Single ended
- property LIFO



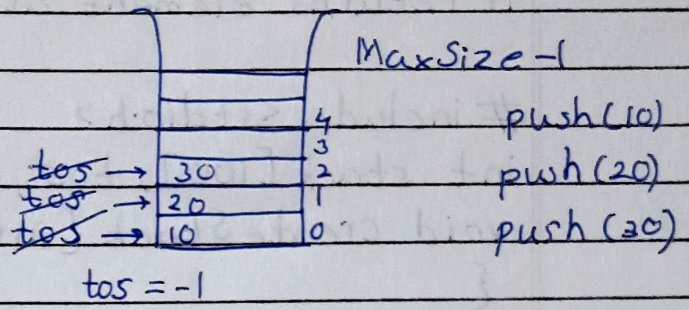
ADT stack

① Create stack (size)



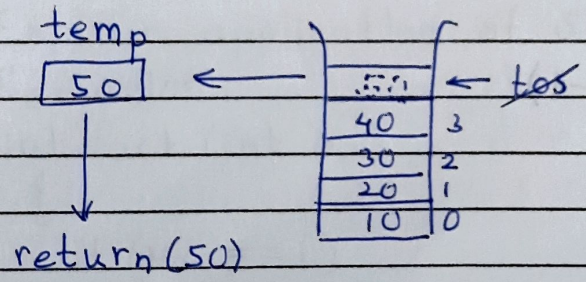
MaxSize=size;
tos=-1;

② push(e)
adds data to stack
① tos + 1
② stack[tos] = e



③ isFull() : 1/0
it returns 1 if Stack is Full else 0
when tos == MaxSize-1 it is full.

④ Pop() : e [It removes and return]
① Record element at tos
② Reduce tos → tos - 1
③ Return (old element at tos)



⑤ isEmpty() : 1/0
it returns 1 if empty else 0 if tos is at -1 then empty

⑥ printStack() : LIFO
print data from top to 0.
↑
tos

⑦ peek() : e

it returns element at tos

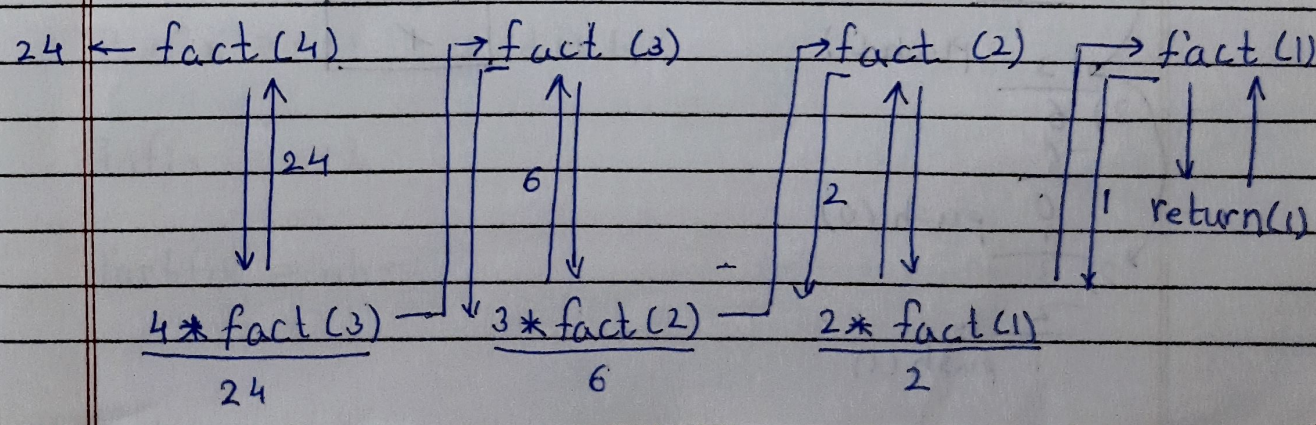
```
#include <stdio.h>
int stack[100], tos, MaxSize;
void createStack(int size)
 {
   MaxSize=size;
   tos=-1;
 }
void push(int e)
 {
   tos++;
   stack[tos]=e;
 }
int isFull()
 {
   if(tos==MaxSize-1)
     return(1);
  else
     return(0);
 }
int pop()
 {
   int temp;
   temp=stack[tos];
   tos--;
   return(temp);
}
int isEmpty()
{
   if(tos==-1)
```

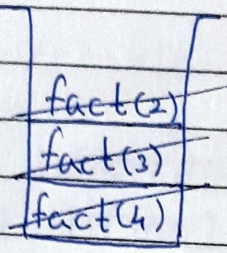
```
    return (1);
  else
    return (0);
}
int peek ()
{
  return (stack [tos]);
}
void printStack ()
{
  int i;
  for (i=tos ; i>=0 ; i--)
    if ("\n %d", stack [i]);
}
```

5mark Q) Explain application of Stack

① Recursion :-

```
int fact (int No:)
{
  if (No == 1)
  return (1);
  else
  return (No * fact (No - 1));
}
```





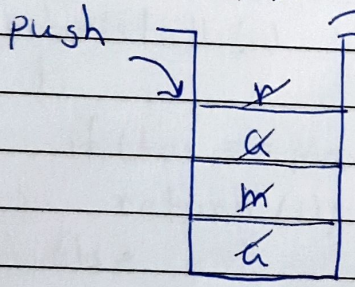
+ve - Make code look smaller hence more readable

-ve - More complex to understand
- Takes more memory
- Slow in execution

Imp :-
- Recursion should be used as last option.

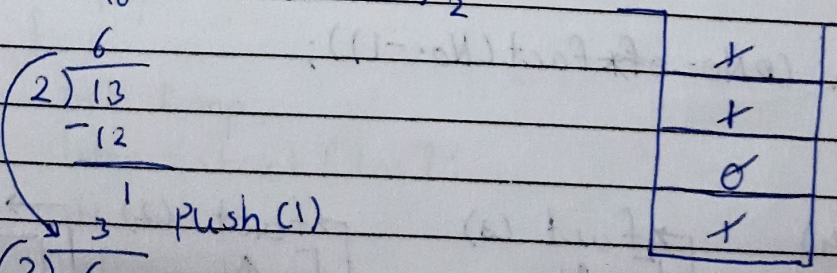
② Reversing Word :-
- Push every word till string is over
- Pop character by character and copy to string

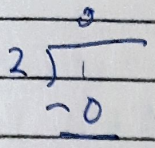
S = 'ram'



③ Decimal to Binary :-

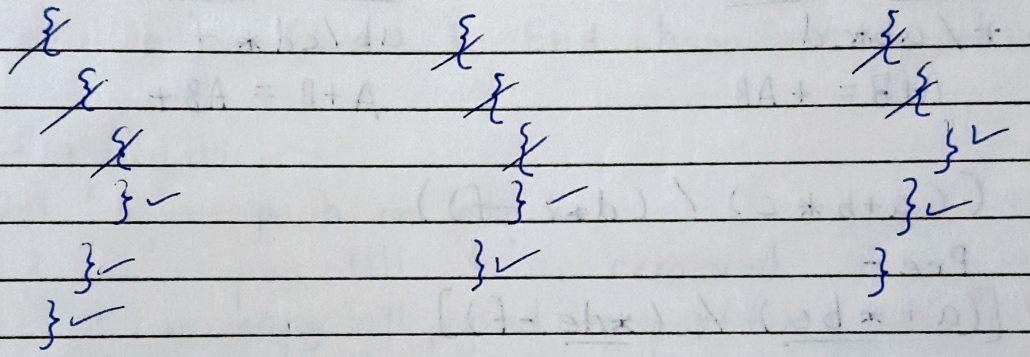
$(13)_{10} \rightarrow (1101)_2$

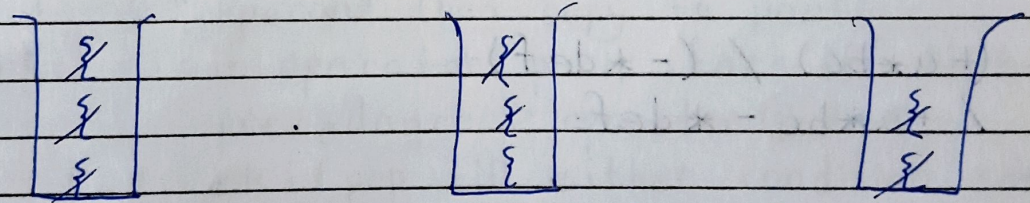




push (0)

4) Wellness of Control: or { [( )] } (Balance)
- push each opening
- on every closing pop.
- if balanced then at end stack becomes empty
- else error -





Error: Expected Error: Unexpected

5) Undo / Redo Command

* Expression Evaluation and Conversion :-

Prefix = +ab add (a,b)

Infix = a+b

Postfix = ab+

* Rules :
- if all equal to go L → R
- if not equal then go for higher and then lower
(lower : + - higher : * / %)
- in case of ( ) solve from inner most.

e.g

1) a/b + c*d

Pre -
/ab + *cd
+/ab*cd
A+B = +AB

Post -
ab/ + cd*
ab/cd* +
A+B = AB+

2) ((a+b*c) / (d*e-f))

Pre -
[(a + *bc) / (*de -f)]

(+a*bc) / (-*def)
/ +a*bc - *def.

Post -
[(a + bc*) / (de* -f)]

(abc*+) / (de*f-)
abc*+ de*f-/

3) (a+b*c) + (d-e+f)

Pre - (Extra)
(a + *bc) + (-de+f)
(+a*bc) + (+-def)
++a*bc+-def.

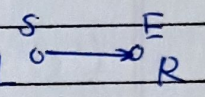
Post -

$(a+bc*)+(de-+f)$

$(abc*+)+(de-f+)$

$abc*+de-f++$

Algorithm :- (Postfix)

1) accept infix and initialize stack 

2) read infix Start to End character by character.

3) if read is :
   3.1 '(' : push on stack
   3.2 ')' : pop till '(' is removed
   copy all poped to postfix (not '(')
   3.3 : Operand then copy to postfix
   3.4 : if operator push stack iff it has precedance > operator @ tos.
   else pop till either condition satisfies or stack becomes empty then push.
   Copy all poped to postfix.

4) Continue stop 3 till infix not over.

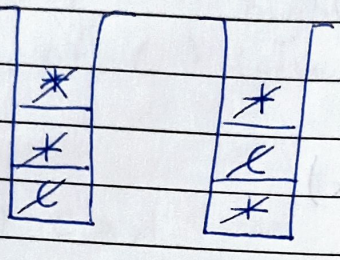
5) Copy leftover from stack to postfix.

6) stop.

e.g (a + b * c) + (a - e + f)

Postfix | abc*+de-f++ |

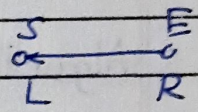
Stack



Read: (push to stack
a copy to post
+ push on stack
b copy to post
* push on stack
c copy to post
) pop till '(' and copy poped to post
+ push to stack
(push to stack
d copy to post
- push to stack
e copy to post
+ pop - and push +
f copy to post
) pop till '(' and copy poped to post

Algorithm :- (prefix)
(a+b-c) * (d/e*f)
(+ab-c) * (/de*f)
(-+ab-c)*(*/def)
-+ab-c/def

1) accept infix and initialize stack



2) Read infix from end to start, character by character.

3) if read is
- 3.1 ')' : push on stack
- 3.2 '(' : pop till ')' is removed
  copy all poped to prefix (not ')')
- 3.3 operand then copy to prefix
- 3.4 : if operator push stack iff it has precedance ≥ operator @ tos
  else pop till either condition satisfies or stack becomes empty then push.
  Copy all poped to prefix

4) Continue step 3 till infix not over

5) Copy leftover from stack to prefix.

6) Reverse prefix

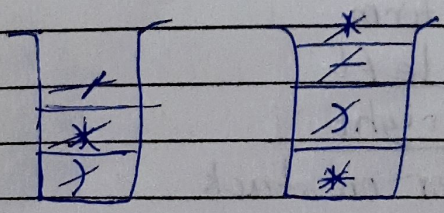
7) Stop

e.g.

(a+b-c)*(d/e*f)

fed/*cba+-*

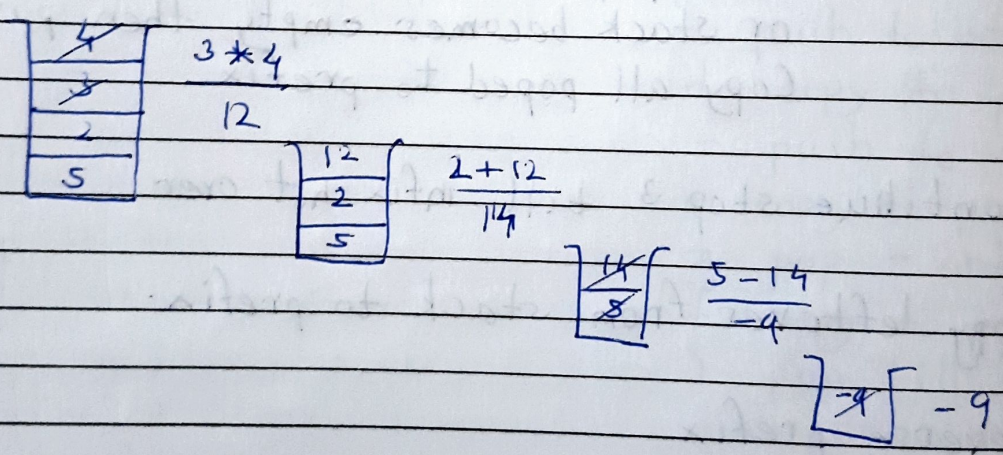
-+abc/def



Algorithm : Postfix evaluation

1) Accept Postfix and initialize stack
2) Read postfix start to End
3) if read is operand push
4) if read is operator perform
   2 pops let 1st pop be right
   2nd pop be left
   operator and push answer on stack
5) Continue step 2,3,4 till postfix not over
6) pop stack and take it as answer.

5 2 3 4 * + *



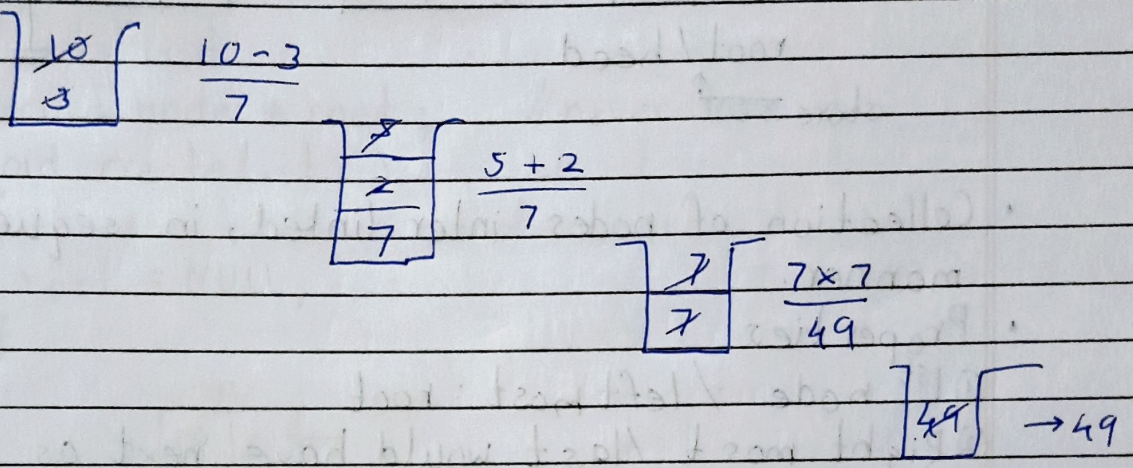
Algorithm : Prefix evaluation

1) Accept Prefix and intialize stack.
2) Read prefix End to start
3) if read is operand push
4) if read is operator perform
   2 pops let 1st pop be left
   2nd pop be right
   operate and push answer on stack

5) continue step 2,3,4 till prefix not over

6) pop stack and take it as answer.

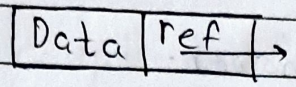
e.g *, +, 5, 2, -, 10, 3

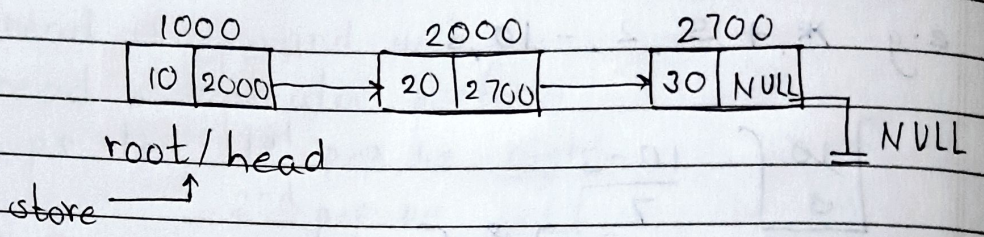


07/09/12

* Linked List :- (Linear, Dynamic Structure)

~~LIFO~~ ~~FIFO~~ as per need



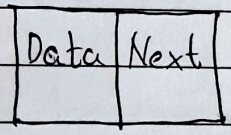


- Collection of nodes inter-linked, in sequential manner.
- Properties

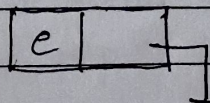
① 1st node / leftmost : root

② Right most / last would have next as NULL.

Node




```
struct node
  {
    int data;
    struct node *next;
  };
struct node *n
n=(struct node *)malloc(sizeof(struct node));
```



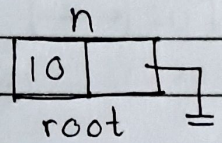

```
n -> data = e;
n -> next = NULL;
```

createlist()
insertLeft(e)
insertRight(e)
deleteLeft();
deleteRight();
search(key)
printlist()

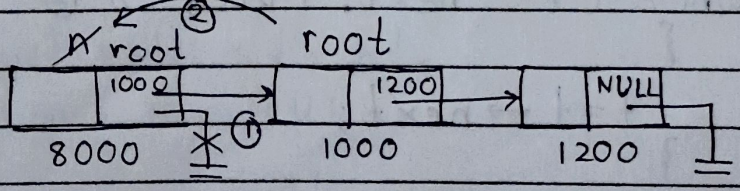
---

deleteElement(e)
insertAt(index,e)

```
# include <stdio.h>
 struct node
 {
   int data;
   struct node * next;
 };
// create list
 struct node * root;     // never lose it
 void createlist ()
 {
   root = NULL;
 }
```

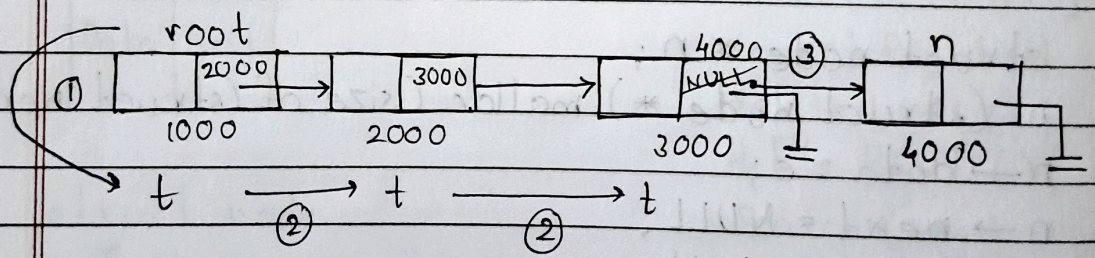



```
 void insertleft (int e)
 {
   struct node * n;
   n=(struct node *) malloc (size of (struct node));
   n → data = e;
   n → next = NULL;
   if (root == NULL)
    {
      root = n;
    }
```



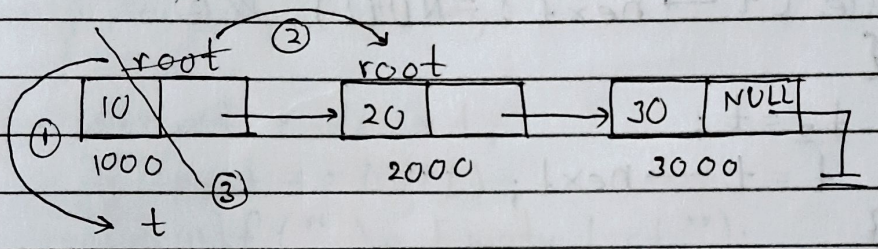

```
   else
    {
```

```
    n->next = root;     // ①
    root = n;           // ②
  }
  printf("\n %d inserted: ", e);
}

void insertRight (int e)
  {
    struct node *n, *t;
    n=(struct node *)malloc (size of (struct node));
    n-> data = e;
    n -> next = NULL;
    if (root == NULL)
     {
       root = n;
     }
```

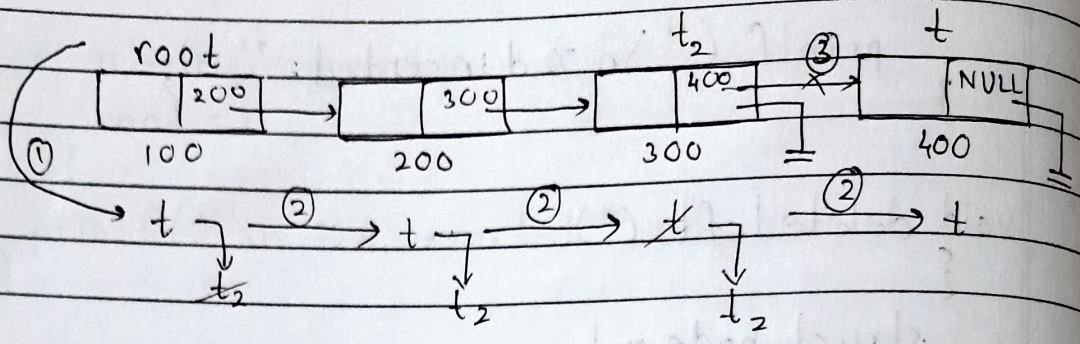



```
    else
     {
       t = root;  // ①
       while (t -> next != NULL)  // ②
          {
            t = t -> next;
          }
       t -> next = n;  // ③
    }
```

```
    printf ("\n %d inserted: ", e);
  }

void deleteLeft ()
  {
   struct node *t;
   if (root == NULL)
     {
      printf ("Empty List");
     }
```

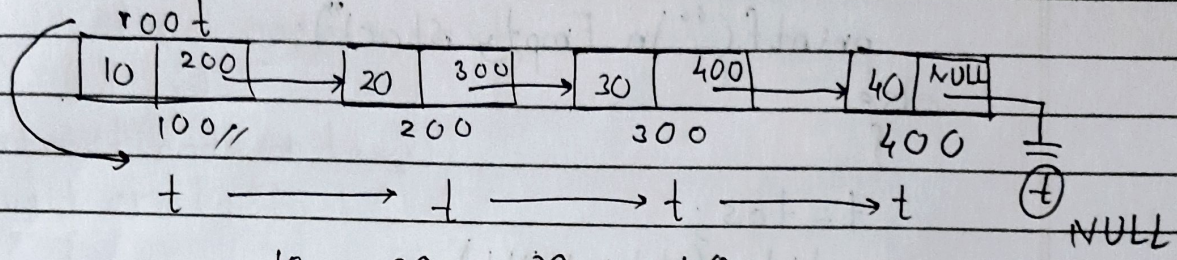



```
   else
     {
       t = root;   // ①
       root = root → next;  // ②
       printf ("\n %d deleted", t → data);
       free(t);  // ③
     }
  }

void deleteRight ()
  {
   struct node *t, *t2;
   if (root == NULL)
     {
      printf ("Empty List");
     }
```




```
else
  {
    t=root;          // ①
    t2=root;
    while (t->next!=NULL)  // ②
      {
        t2=t;
        t=t->next;
      }
    t2->next=NULL;   // ③
    printf("\n %d deleted",t->data);
    free(t);
  }
}

void printLinkedList()
  {
    struct node *t;
    if(root==NULL)
      {printf("\n Empty List");
    }else
      {
        t=root;  // ①
        while(t!=NULL)  // ②
          {printf("%d ",t->data);
```

```
        t = t → next;
      }
    }
  }
```

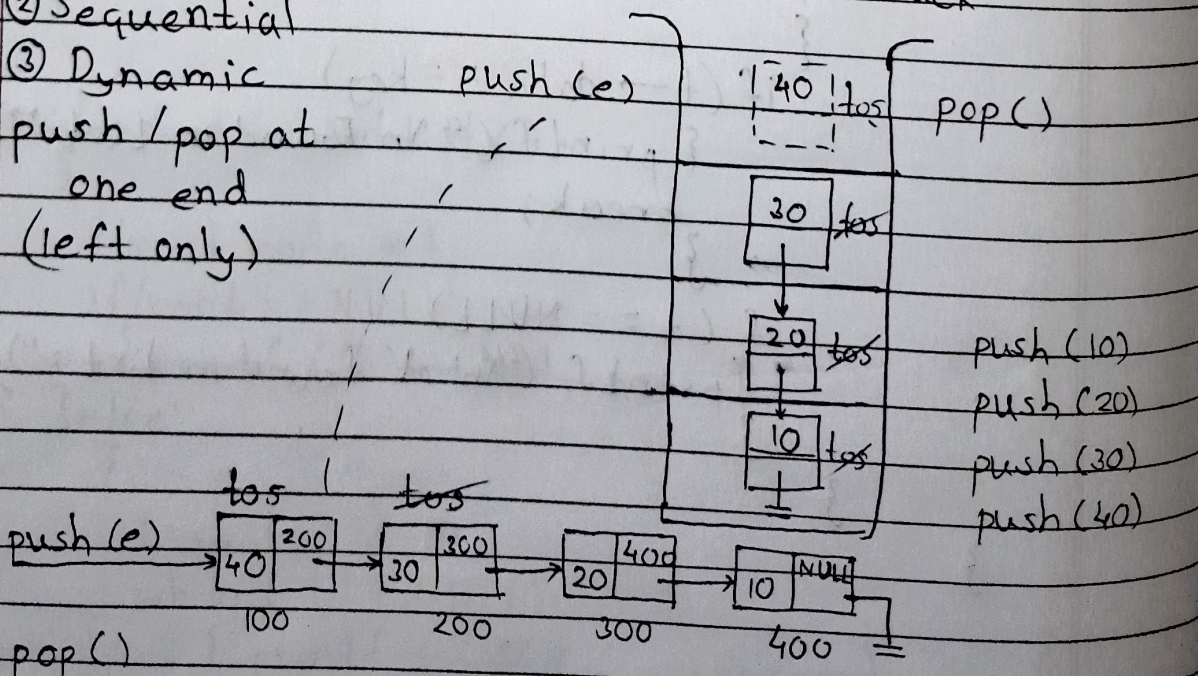



```
void search (int key)
  {
    struct node *t;
    if (root == NULL)
    {printf (" \n Empty List ");
  }else
       {
          t = root;  // ①
          while (t != NULL)
            {
              if (t → data == key)
                 { printf (" \n Found in List:");
                   break;
                 }
              if (t == NULL)
               {printf ("\nNot found in List:");
               }
          }
  }
```

```
void PrintStack()
  {
    struct node *t;
    if(rootos == NULL)
      printf("\n Empty stack");
    else
      {
        t = tos;
        while (t != NULL)
          {
            printf ("%d", t->data);
            t = t->next;
          }
      }
  }
```

* Stack using Linked List:-

① LIFO
② Sequential
③ Dynamic
push / pop at one end (left only)



```
#include <stdio.h>
struct node
  {
    int data;
    struct node *next;
  };
struct node *tos;
void createStack()
  {
    tos=NULL;
  }
void push(int e)
  {
   struct node*n;
   n=(struct node*)malloc(sizeof(struct node));
   n->data=e;
   n->next=NULL;
   if(tos==NULL)
    {
     tos=n;
    }
   else
    {
     n->next=tos;
     tos=n;
    }
  printf("\n%d pushed",e);
 }
void pop()
  {
   struct node*t;
```

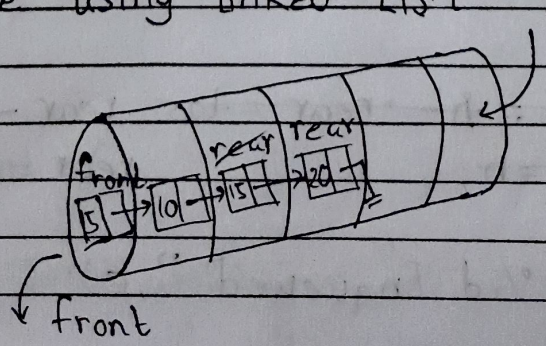
```
    if (tos == NULL)
      printf ("\n Empty stack");
    else
      {
        t = tos;
        tos = tos -> next;
        printf ("\n %d poped", t -> data);
        free (t);
      }
  }
void printStack()
  {
    struct node * t;
    if (tos == NULL)
      printf ("\n Empty stack");
    else
      {
        t = tos; // ①
        printf ("\n Stack has \n");
        while (t != NULL)
          {
            printf ("\n %d", t -> data);
            t = t -> next;
          }
      }
  }

void main ()
  {
    create stack();
    push(10);
```

```
push (20);
push (30);
push (40);
printStack();
pop();
pop();
printStack();
}
```

Output:-

```
10 pushed
20  //
30  //
40  //
Stack has
40
30
20
10
40 poped
30 poped
Stack has
20
10
```

* Dynamic Queue:-
Queue using linked List

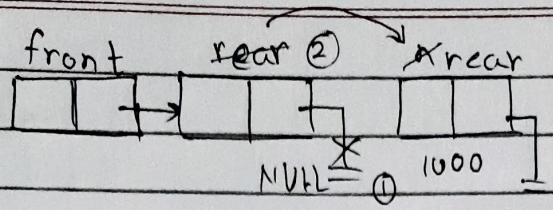


- FIFO
- Double ended
- front → Rear

print: FIFO

Enqueue / Insert_right()
Dequeue / Delete_left()

```
#include <stdio.h>
 struct node
 {
  int data;
  struct node *next;
 };
struct node *rear, *front;
 void create Queue()
  {
   rear = NULL;
   front = NULL;
  }
void Enqueue (int e)
  {
  struct node *n;
  n = (struct node *) malloc (size of (struct node));
  n → data = e;
  n → next = NULL;
 if (front == NULL)
    {
     front = rear = n;
    }
 else
    {
      rear → n = n → rear = tos; rear → next = n; //①
      rear tos = n;                rear = n; //②
    }
  printf ("\n %d Enqueued ", e);
}
```




```
void Dequeue ()
  {
   struct node *t;
   if (front == NULL)
     printf ("\n Empty Queue");
   else
     {
      t = front;      // ①
      front = front → next;   // ②
      printf ("\n %d Dequeued", t → data);
      free (t);      // ③
     }
  }
void print Queue ()
  {
   struct node *t;
   if (front == NULL);
     printf ("\n Empty Queue");
   else
     {
       printf ("\n Queue has \n");
       for (t = front; t != NULL; t = t → next)
          printf ("%d ", t → data);
     }
  }
void main ()
  {
   create Queue ();
```

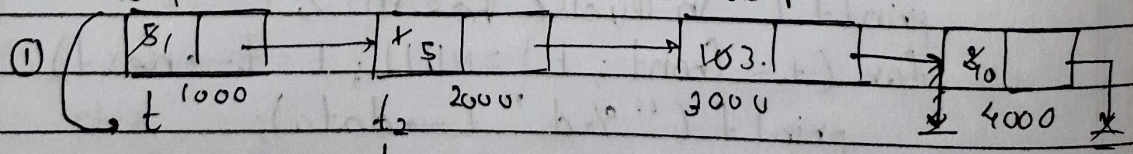
```
Enqueue (10);
    //    (20);
    //    (30);
    //    (40);
print Queue ();
Dequeue ();
Dequeue ();
print Queue ();
}
```

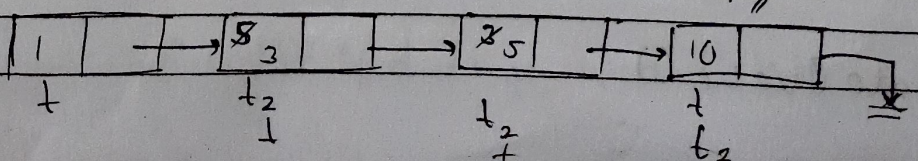
Output -

10 Enqueued
20
30
40

Queue has
10 20 30 40
10 Dequeued
20 Dequeued
Queue has
30 40

* Sorting :- Write a pseudo code to sort linear linked list using bubble code p

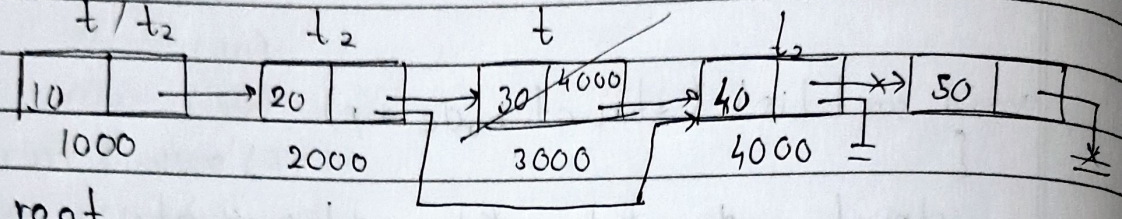




if t.

```
void sortList (struct node *r)
  {
   struct node *t , *t2 , *temproot;
   int temp;
   if (r == NULL)
    printf("Empty List");          sortList(1000)
   else
    {
     temproot = r;
     while (r -> next != NULL)  // passes n-1
     {
      t = temproot ;  //<start from 1st
      t2 = t -> next;
      while (t -> next != NULL)  // sort
       {
        if (t -> data > t2 -> data)  // Ascending
          {
           temp = t -> data;
           t -> data = t2 -> data;
           t2 -> data = temp;
          }
        t = t -> next;
        t2 = t2 -> next;
       }
     r = r -> next;
     }
  }
```

* Delete Specific:- Delete a specific number from linked list if found.



$t_2 \rightarrow next = t \rightarrow next$

key = 10
key = 50
key = 30

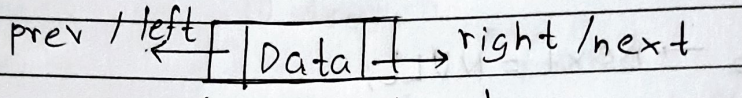
① search
↳ if found
↳ case 1: left most
if (t == root) → delete_left()
↳ case 2: Right most.
if (t → next == NULL) → delete_right()
↳ case 3
if not 1st / 2nd then in betn.

```
void delete (int key)
 {
  struct node *t, *t2;
  if (root == NULL)
   printf ("Empty list");
  else
   {
    t = root;
    t2 = root;
    while (t != NULL && t → data != key)
     {
      t2 = t;
      t = t → next;
     }
```

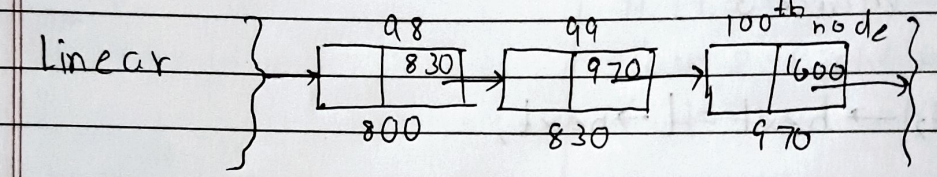
```
if (t != NULL) // found
{   // case 1
    if (t == root)
    {
        root = root->next;
    }
    // case 2
    else if (t->next == NULL)
    {
        t2->next = NULL;
    }
    else   // case 3
    {
        t2->next = t->next;
    }
    printf ("\n %d found and deleted", t->data);
    free(t);
}
else
    printf ("\n %d not found", key);
}

x void PEEK()
{
    if (top == NULL)
    {
        Printf("\n Stack is Empty");
    }
    else
    {
        printf ("The topmost element= %d", top->data);
    }
}
```

* Doubly linked list:-

1) Need of Doubly linked list:-
Linear linked list can only travel in one direction as a end result we cannot access last data while moving forward, to access last element we have to start again from first.
Sol^n is doubly linked list.



It has special node structure.



Drop current goto 1st (root) and again start till we reach 99th node.

```
#include <stdio.h>
#include <conio.h>
```