Module 01

Introduction to Data Structures

Data structure can be defined as a representation of data along with its associated operations. It is the way of organizing and storing data in a computer system so that it can be used efficiently.

The data elements are referred as members of the data structure.

Members can be of different types & lengths

Example: Arrays
Linked Lists
Binary trees
Stacks
Queues.

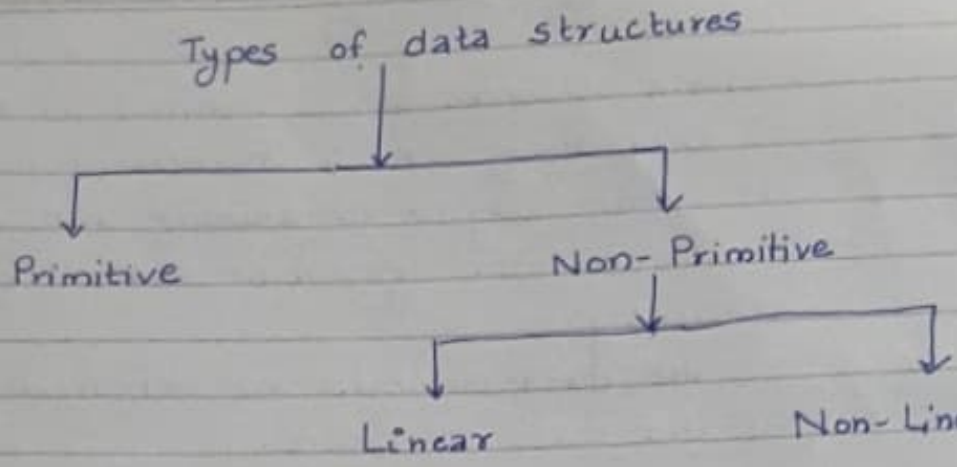
Characteristics of Data Structure:
DS help in Storing
Organizing
Analyzing the data.

1. Depicts the logical representation of data in computer memory.

2. Represents the logical relationship between the various data elements.

3. Helps in efficient manipulation of stored data elements.

4. Allows the programmers to process the data in an efficient manner.



Primitive DS include all the fundamental DS that can be directly manipulated by machine-level instructions.

- Integer
  character
  Real
  Boolean

Non Primitive Ds. refer to all those data structures that are derived from one or more primitive Ds. To form sets of homogenous or heterogenous data elements

2 Types → Linear
Non-Linear

In Linear DS, all the data elements are arranged in a linear or sequential form.

- Arrays
- Stacks
- Queues
- Linked Lists, etc.

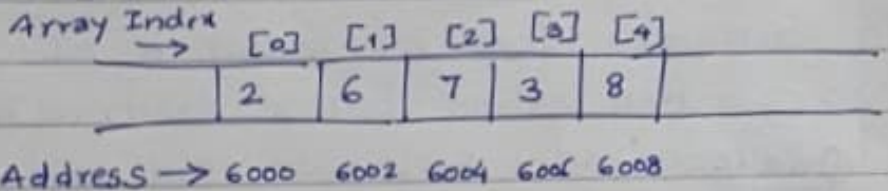
In non-linear DS, there is no definite order or sequence in which data elements are arranged.

- Trees
- Graphs.

Arrays

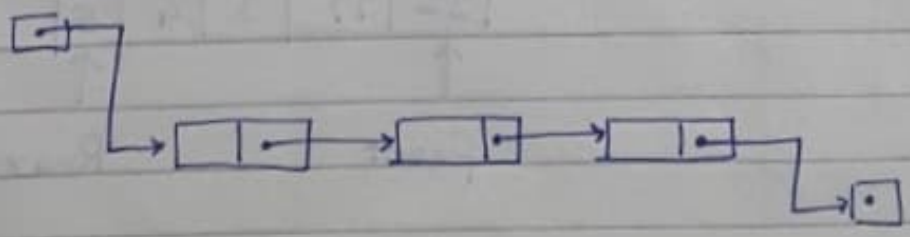
A collection of similar type data elements stored at consecutive locations in the memory.

examples: List of integers
Group of names.



Linked Lists

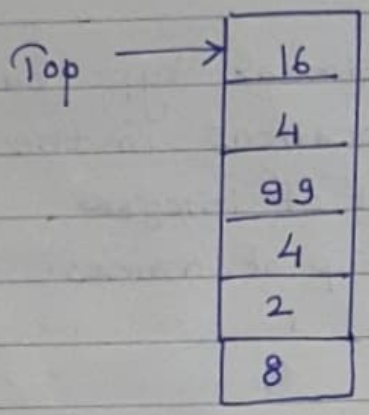
Stores data in the form of a list.
Multiple nodes connected to each other thru pointers



Stacks

Is a linear DS that maintains a list of elements in such a manner that elements can be inserted or deleted only from one end of the list → Top of the stack.

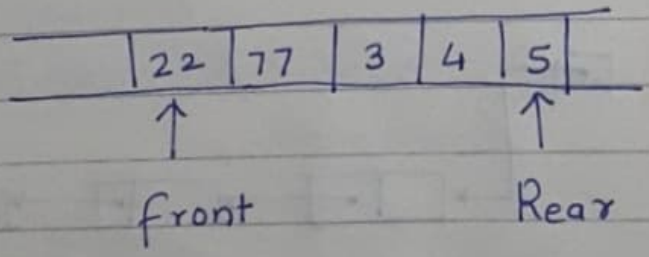
LIFO principle.



Queue

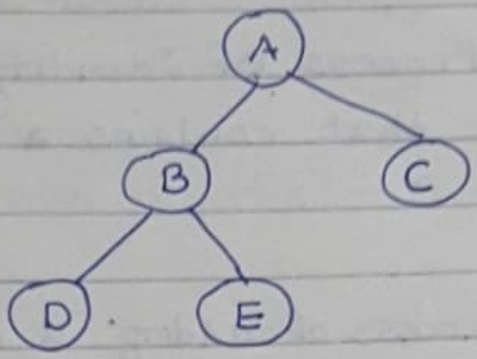
Linear DS that maintains a list of elements in such a manner that elements are inserted from one end of the queue (rear) and deleted from the other end (front)

— FIFO principle.



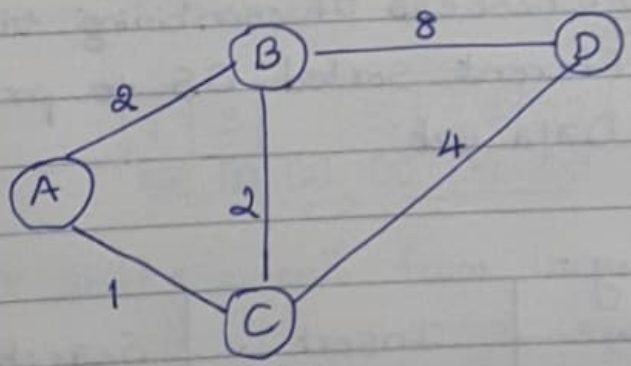
Trees

Is a linked data structure that arranges its nodes in the form of a hierarchical tree structure. Node at top is Root Node.



Graphs

Is a linked DS that comprises of a group of vertices called nodes and a group of edges. Edges are associated with weights.



Data structure Operations:

1. Traversing: Process of accessing each record of a data structure exactly once.

2. Searching: Process of identifying the location of a record that contains a specific key value.

3. Inserting: Process of adding a new record

4. Deleting: Process of removing an existing record.

5. Sorting: Process of arranging the records of a DS in a specific order (Alphabetical, ascending or descending.

6. Merging: Process of combining the records of two different sorted DS to produce a single sorted Data set.

Efficiency:

Data structure	Insert	Search	Delete
Array	O(n)	O(n)	O(n)
Linked List	O(1)	O(n)	O(n)
Stack	O(1)	–	O(1)
Queue	O(1)	–	O(1)
Tree	O(log n)	O(log n)	O(log n)

ARRAYS

An array is a collection of same type elements.

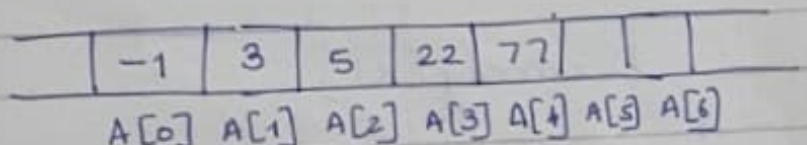
Types of Arrays:

- One-dimensional array
- Multi-dimensional array

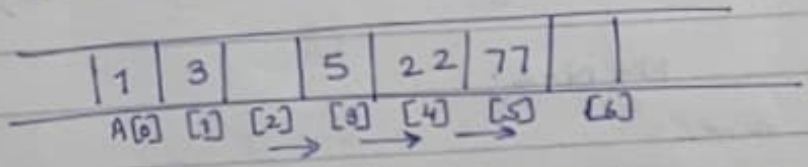
Address of A[K] = B + W * K

B = Base address
W = Word size
K = index identifier

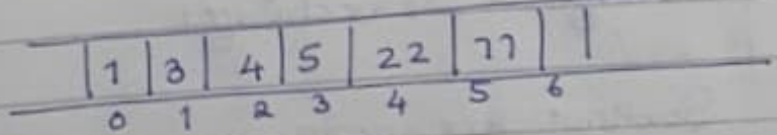
Insertion And Deletion:



To insert element at location 2



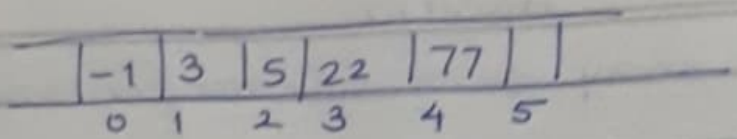
first shift elements from locn 2 by one place.



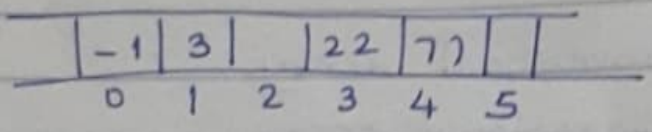
Then insert element at location 2.

if K is the location for the element to be inserted

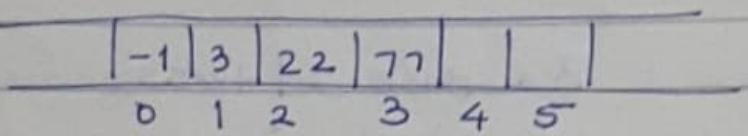
```
for (i = N; i >= K; i--)
    array[i+1] = array[i];

array[K] = P;
N = N+1;
```



Delete an element from location 2



We have to shift all the element from 3rd position towards left by 1.



If K is the location from where element is to be deleted:

```
D = array[K];

for (i=K; i<=N-2; i++)
  array[i] = array[i+1];

N=N-1;
```

Sorting & Searching:

Sorting arranges the elements of an array in a specific order or sequence.

Searching locates a specific element in the array.

Bubble Sort :

{ 5, 4, 3, 2, 1 }

{ 4, 5, 3, 2, 1 }

{ 4, 3, 5, 2, 1 }

{ 4, 3, 2, 5, 1 }

{ 4, 3, 2, 1, 5 } Pass 1 .

{ 4, 3, 2, 1 }
{ 3, 4, 2, 1 }

{ 3, 2, 4, 1 }

{ 3, 2, 1, 4 } Pass 2

{ 3, 2, 1 }
{ 2, 3, 1 }

{ 2, 1, 3 } Pass 3

{ 2, 1 }
{ 1, 2, } Pass 4

{ 1, 2, 3, 4, 5 } final Result .

```
for (i = 5; i > 1; i--)
  for (j = 0; j < i-1; j++)
    if (array[j] > array[j+1])
      { t = array[j+1];
        array[j+1] = array[j];
        array[j] = t;
      }
```

Linear Search.

If K is the element to be searched in the array:

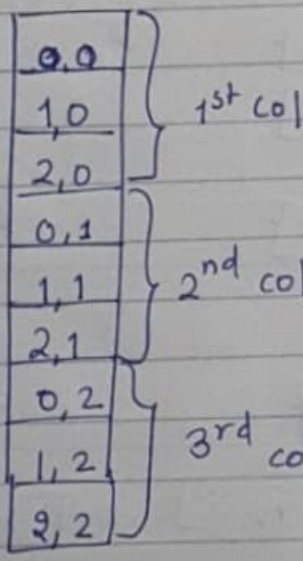
```
for (i=0 ; i< 5 ; i++)
   if (K== array[i])
    {
       flag= 1;
       index= i;
       break;
    }
  else
     ;
```

Multidimensional Arrays:

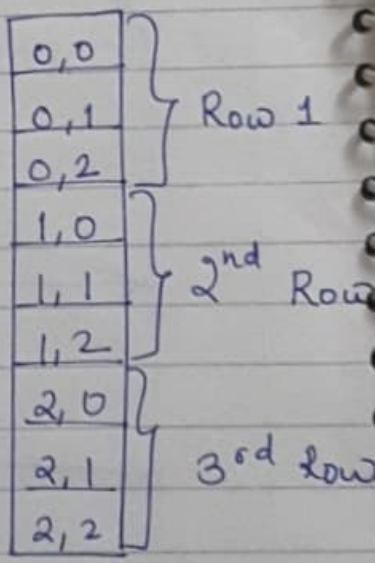
Array [3] [3] in

Column-major

	0	1	2
0	00	01	02
1	10	11	12
2	20	21	22



Row-major



CM: Address of A[i,j] = B + W (m (j - LBR) + (i - LBR))

RM: Address of A[i,j] = B + W (n (i - LBR) + (j - LBC))

B = Base address

W = word size

n = no of columns m: no of rows.

LBR: Lower bound of row index

LBC: Lower bound of column index.

Abstract data type (ADT)

An ***Abstract data type*** (ADT) is the specification of a data type within some language, independent of an implementation.

The abstract data type is special kind of datatype, whose behavior is defined by a set of values and set of operations.

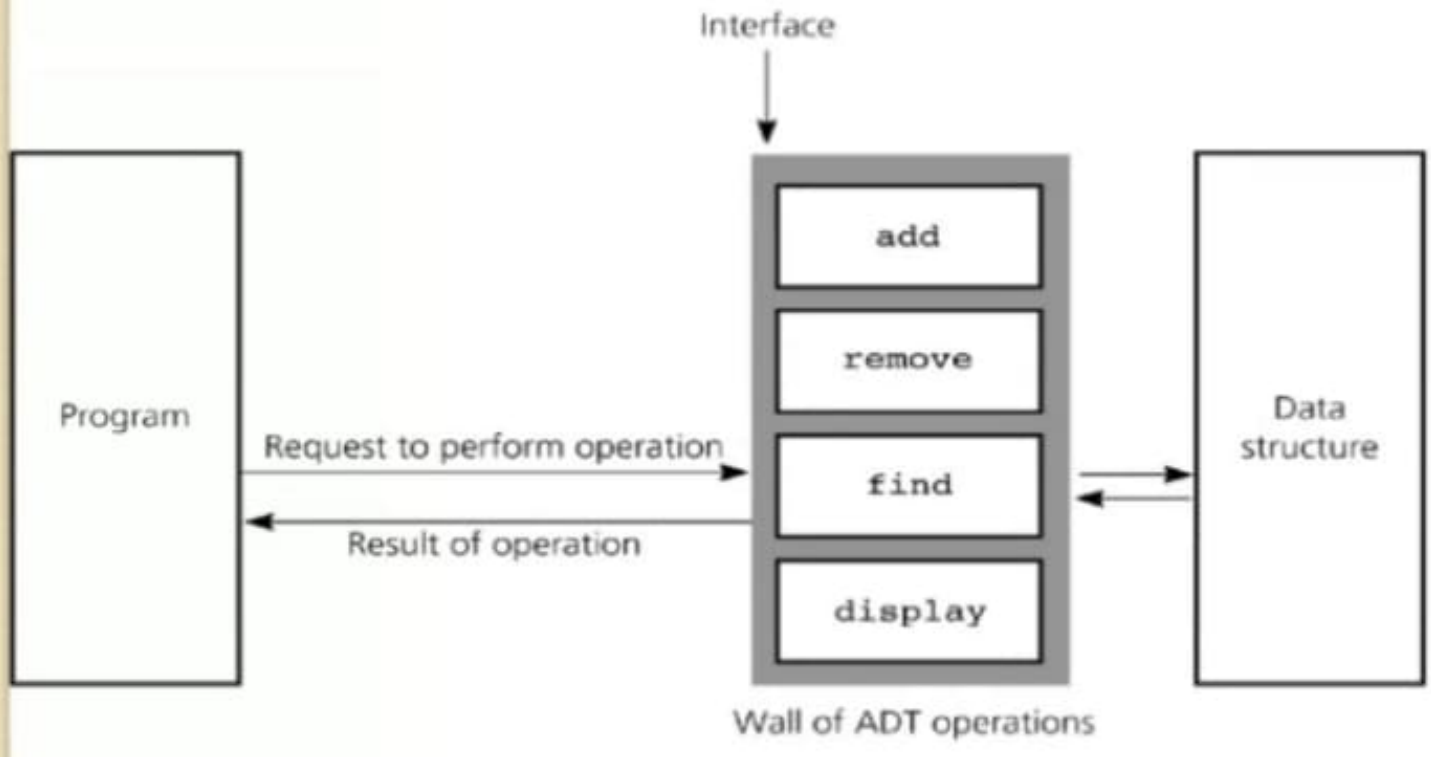
The keyword "Abstract" is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user.

A *data structure* is the implementation for an ADT.

Abstract data type (ADT)

- An ADT is composed of
  - A collection of data
  - A set of operations on that data
- Specifications of an ADT indicate
  - What the ADT operations do, not how to implement them
- Implementation of an ADT
  - Includes choosing a particular data structure

Abstract Data Types



A wall of ADT operations isolates a data structure from the program that uses it

What is Abstract Data Type ?

Definition : ADTs are entitites that are definitions of data and operations but do not have implementation details.

Real world Example

smartphone

Abstract/logical view

- 4 GB RAM
- Snapdragon 2.2GHz processor
- 5.5 inch LCD screen
- Dual Camera
- Android 8.0

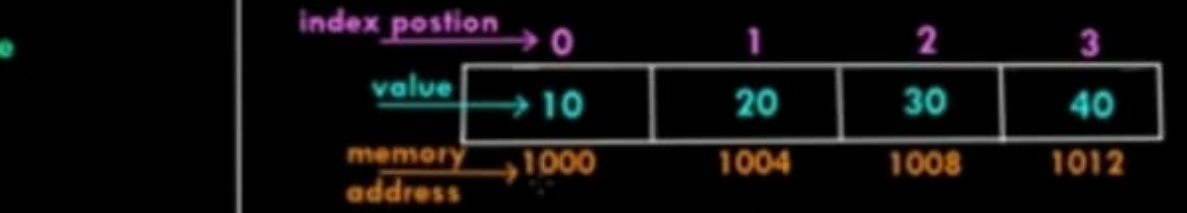
- call()
- text()
- photo()
- video()

Implementation view

```
class Smartphone{
  private:
    int ramSize;
    string processorName;
    float screenSize;
    int cameraCount;
    string androidVersion;
  public:
    void call();
    void text();
    void photo();
    void video();
};
```

Data Structure Example

Integer Array



Abstract/logical view

- store a set of elements of int type
- read elements by postion i.e index
- modify elements by index
- perform sorting

Implementation View

```
int arr[5] = {1,2,3,4,5};
cout<<arr[1];
arr[2]=10;
```

insertion

https://youtu.be/KELqVT7hjeE

deletion

https://youtu.be/CZYR2v8rYLA

```
#include<stdio.h>
int main()
{
   int arr[100] = { 0 };
   int i, x,k, pos, n = 10;

   // initial array of size 10
   for (i = 0; i < 10; i++)
      arr[i] = i + 1;

   // print the original array
   for (i = 0; i < n; i++)
      printf("%d   ", arr[i]);
   printf("\n");
    printf("\n");
```

```
   printf("\nelement to be inserted=");
   scanf("%d",&x);

   // position at which element
   // is to be inserted
   pos = 5;

   // increase the size by 1
   n++;

   // shift elements forward
   for (i = n; i >= pos; i--)
```

```
printf("\nelement to be inserted=");
scanf("%d",&x);

// position at which element
// is to be inserted
pos = 5;

// increase the size by 1
n++;

// shift elements forward
for (i = n; i >= pos; i--)
   arr[i+1] = arr[i];

// insert x at pos
arr[pos] = x;

// print the updated array
for (i = 0; i < n; i++)
   printf("%d   ", arr[i]);
printf("\n");

printf("\nelement to be deleted from position=");
scanf("%d",&k);
```

```
    for (i = k; i<= n-2; i++)
       arr[i] = arr[i+1];
    n--;

    // print the updated array
    for (i = 0; i < n; i++)
       printf("%d    ", arr[i]);
    printf("\n");

    return 0;
}
```

BUBBLE SORT PROGRAM (https://youtu.be/9I2oOAr2okY)

```
#include <stdio.h>
int main()
{
   int count, temp, i, j, number[30];

  printf("How many numbers are u going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d numbers: ",count);
  for(i=0;i<count;i++)
  scanf("%d",&number[i]);

  /* This is the main logic of bubble sort algorithm     */
  for(i=count;i>1;i--){
    for(j=0;j<i-1;j++){
      if(number[j]>number[j+1]){
        temp=number[j+1];
        number[j+1]=number[j];
        number[j]=temp;
      }
    }
  }
  printf("Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
   return 0;
}
```

LINEAR SEARCH PROGRAM https://youtu.be/jST8g680yG0

```
#include <stdio.h>
int main()
{
 int array[100], search, c, n;

 printf("Enter number of elements in array\n");
 scanf("%d", &n);

 printf("Enter %d integer(s)\n", n);
 for (c = 0; c < n; c++)
  scanf("%d", &array[c]);

 printf("Enter a number to search\n");
 scanf("%d", &search);

 for (c = 0; c < n; c++)
 {
  if (array[c] == search)    /* If required element is found */
  {
   printf("%d is present at location %d.\n", search, c+1);
   break;
  }
 }
 if (c == n)
  printf("%d isn't present in the array.\n", search);
 return 0;
}
```

Binary Search
https://youtu.be/sr_bR1WwcLY

```
#include <stdio.h>
int main()
{
int i, low, high, mid, n, key, array[100];

printf("Enter number of elements\n");
scanf("%d",&n);

printf("Enter %d integers\n", n);
for(i = 0; i < n; i++)
scanf("%d",&array[i]);

printf("Enter value to find\n");
scanf("%d", &key);

low = 0;
high = n - 1;
mid = (low+high)/2;

while (low <= high)
{
if(array[mid] < key)
low = mid + 1;

else if (array[mid] == key)
 {
printf("%d found at location %d\n", key, mid+1);
break;
}

else
high = mid - 1;
mid = (low + high)/2;
}

if(low > high)
printf("Not found! %d isn't present in the list\n", key);
return 0;
}
```

```
#include <stdio.h>

int main()
{
 int c, first, last, middle, n, search, array[100];

 printf("Enter number of elements\n");
 scanf("%d", &n);

 printf("Enter %d integers\n", n);

 for (c = 0; c < n; c++)
  scanf("%d", &array[c]);

 printf("Enter value to find\n");
 scanf("%d", &search);

 first = 0;
 last = n - 1;
 middle = (first+last)/2;

 while (first <= last)
 {
```

```
{


  if (array[middle] < search)
    first = middle + 1;
  else if (array[middle] == search)
  {
    printf("%d found at location %d.\n", search, middle+1);
    break;
  }
  else
    last = middle - 1;

  middle = (first + last)/2;
 }
 if (first > last)
  printf("Not found! %d isn't present in the list.\n", search);
 return 0;
}
```

Introduction to Stack

Introduction to stack
ADT of Stack
Operations on Stack
Array Implementation of Stack


20200729_192753.jpg


20200729_193025.jpg


20200729_193038.jpg


Stack Working -- Step by Step Animation
(IMPLEMENTATION & CODE)

Array Stack Visualization

STACK AS AN ABSTRACT DATA TYPE

ARRAY IMPLEMENTATION OF STACK.docx

Add class comment

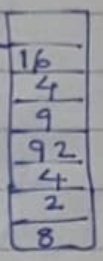
MODULE 02

STACKS

Stack is a linear data structure in which items are added or removed from only at one end, called the top of the stack.

LIFO (Last in first out)

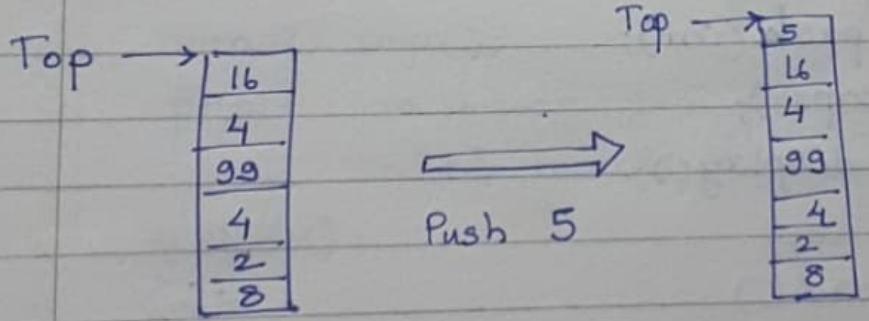
stack of books
stack of plates.

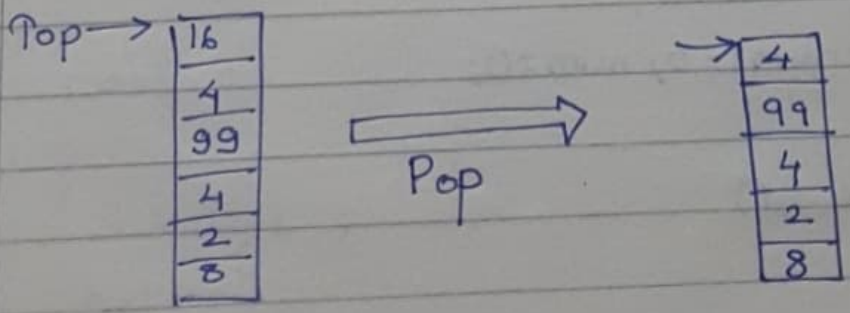


Stack Operations:

Push: Adding an element to the stack.

Pop: Reading or deleting an element from the stack.





Push operation: ① Receive the element to be inserted
② Increment the top
③ Store the element at top

Pop operation: ① Remove element at top.
② Decrement top.

Array Implementation of Stacks.

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

int stack [100];
int top = -1;

void push (int);
int pop();
void display();

void main()
{
 int choice;
 int num1=0, num=0;

 while (1)
 {
  clrscr();
  printf ("select a choice from the following:");
  printf ("\n[1] Push an element into the stack");
  printf ("\n[2] Pop out an element from the stack");
  printf ("\n[3] Display the stack elements");
  printf ("\n[4] Exit \n");
  printf ("\n\t Your Choice: ");
  scanf ("%d", &choice);

  switch (choice)
  {
   case 1:
   {
    printf ("\n\t Enter the element to be pushed into
          the stack: ");
    scanf ("%d", &num1);
    push (num1);
    break;
   }

   case 2:
   {
    num2 = pop();
    printf ("\n\t %d element popped out of stack
          \n\t", num2);
    getch();
    break;
   }

   case 3:
   {
    display();
    getch();
    break;
   }

   case 4:
   exit(0);
   break;
```

```
default:
printf("\n Invalid choice ! \n");
break;
}
}

void push (int element)
{
 if (top==99)
 {
  printf ("stack is full \n");
  getch ();
  exit(1);
 }
 top= top+1;
 stack[top] = element;
}

int Pop()
{ if (top == -1)
  {
   printf ("\n\t Stack is Empty \n");
   getch ();
   exit (1);
  }
 return (stack [top--]);
 }

void display ()
 { int i;
   printf ("\n\t The various stack elements are:\n");
   for (i= top ; i>=0; i--)
   printf ("%t %d \n", stack [i]);
}
```

Applications of Stack:

1- Well formness of Parenthesis

7-((X*((X+Y)/(J-3))+Y)/(4-2.5))

We want to enoure parentheses are nested correctly

1. There are an equal number of R & L parentheses
2. Every RP is preceded by a matching LP.

Give a parenthesis count
Increase the count on LP.
Decrease the count on RP.

1 Parenthesis count at the end of exp. must be 0.
2. Parenthesis count at each point should be nonnegate

7 - ((X * ((X + Y) / (J - 3)) + Y) / (4 - 2.5))
0 0 1 2 2 2 3 4 4 4 4 3 3 4 4 4 4 3 2 2 2 1 1 2 2 2 2 1 0

((A + B) x
1 2 2 2 2 1

A + B (x
0 0 0 1

) A + B (- (
-1 -1 -1 -1 0 0 1

(A + B)) - (C + D
1 1 1 1 0 -1 -1 0 0 0 0

ARRAY IMPLEMENTATION OF STACK https://youtu.be/-KQpk-dIA8s

```
#include<stdio.h>

int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);

int main()
{
  //clrscr();
  top=-1;
  printf("\n Enter the size of STACK[MAX=100]:");
  scanf("%d",&n);

  printf("\n\t STACK OPERATIONS USING ARRAY");
  printf("\n\t----------------------------------");
  printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

  do
  {
    printf("\n Enter the Choice:");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1:
      {
        push();
        break;
      }
      case 2:
      {
        pop();
        break;
      }
      case 3:
      {
        display();
        break;
      }
      case 4:
      {
        printf("\n\t EXIT POINT ");
        break;
      }
      default:
      {
        printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
      }

    }
  }
  while(choice!=4);
  return 0;
}
```

```
void push()
{
  if(top>=n-1)
  {
    printf("\n\tSTACK is over flow");

  }
  else
  {
    printf(" Enter a value to be pushed:");
    scanf("%d",&x);
    top++;
    stack[top]=x;
  }
}

void pop()
{
```

```
            break;
        }
        case 4:
        {
            printf("\n\t EXIT POINT ");
            break;
        }
        default:
        {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }

    }
  }
  while(choice!=4);
  return 0;
}
```

```
void push()
{
  if(top>=n-1)
  {
    printf("\n\tSTACK is over flow");

  }
  else
  {
    printf(" Enter a value to be pushed:");
    scanf("%d",&x);
    top++;
    stack[top]=x;
  }
}

void pop()
{
  if(top<=-1)
  {
    printf("\n\t Stack is under flow");
  }
  else
  {
    printf("\n\t The popped elements is %d",stack[top]);
    top--;
  }
}

void display()
{
  if(top>=0)
  {
    printf("\n The elements in STACK \n");
    for(i=top; i>=0; i--)
      printf("\n%d",stack[i]);
    printf("\n Press Next Choice");
  }
  else
  {
    printf("\n The STACK is empty");
  }

}
```

IMPLEMENTATION OF STACK USING STRUCTURE

When structure variable accessed directly

```
#include <stdio.h>
#define MAXSIZE 5

struct stack
{
   int stk[MAXSIZE];
   int top;
};

typedef struct stack STACK;

STACK s;

void push(void);
int  pop(void);
void display(void);

void main ()
{
   int choice;
   int option = 1;
   s.top = -1;

   printf ("STACK OPERATION\n");
   while (option)
   {
      printf ("------------------------------------------\n");
      printf ("     1   -->   PUSH            \n");
      printf ("     2   -->   POP           \n");
      printf ("     3   -->   DISPLAY            \n");
      printf ("     4   -->   EXIT        \n");
      printf ("------------------------------------------\n");

      printf ("Enter your choice\n");
      scanf    ("%d", &choice);
```

```
      switch (choice)
      {
      case 1:
         push();
         break;
      case 2:
         pop();
         break;
      case 3:
         display();
         break;
      case 4:
         return;
```

```
        switch (choice)
        {
        case 1:
           push();
           break;
        case 2:
           pop();
           break;
        case 3:
           display();
           break;
        case 4:
           return;
        }
        fflush (stdin);
        printf ("Do you want to continue(Type 0 or 1)?\n");
        scanf    ("%d", &option);
    }
}
/* Function to add an element to the stack */
void push ()
{
    int num;
    if (s.top == (MAXSIZE - 1))
    {
        printf ("Stack is Full\n");
        return;
    }
    else
    {
        printf ("Enter the element to be pushed\n");
        scanf ("%d", &num);
        s.top = s.top + 1;
        s.stk[s.top] = num;
    }
    return;
}
/* Function to delete an element from the stack */
int pop ()
{
    int num;
    if (s.top == - 1)
    {
        printf ("Stack is Empty\n");
```



```
        return (s.top);
    }
    else
    {
        num = s.stk[s.top];
        printf ("poped element is = %dn", s.stk[s.top]);
        s.top = s.top - 1;
    }
    return(num);
```

```
    return;
}
/* Function to delete an element from the stack */
int pop ()
{
    int num;
    if (s.top == - 1)
    {
        printf ("Stack is Empty\n");
```

```
        return (s.top);
    }
    else
    {
        num = s.stk[s.top];
        printf ("poped element is = %dn", s.stk[s.top]);
        s.top = s.top - 1;
    }
    return(num);
}
/* Function to display the status of the stack */
void display ()
{
    int i;
    if (s.top == -1)
    {
        printf ("Stack is empty\n");
        return;
    }
    else
    {
        printf ("\n The status of the stack is \n");
        for (i = s.top; i >= 0; i--)
        {
            printf ("%d\n", s.stk[i]);
        }
    }
    printf ("\n");
}
```

IMPLEMENTATION OF STACK USING STRUCTURE

When u pass a pointer

```
#include<stdio.h>
#include<conio.h>
#define max 6

typedef struct stack
    {
    int data[max];
    int top;
    }stack;

void init(stack *);
int empty(stack *);
int full(stack *);
void push(stack *,int);
int pop(stack *);
void display(stack *);

void main()
{
stack s;
int ch,x;
init(&s);
clrscr();

do{
printf("1.push\n 2.pop\n 3.display\n 4.exit\n");

printf("enter your choice\n");
```

```
scanf("%d",&ch);

switch(ch)
{
    case 1:printf("enter value\n");
        scanf("%d",&x);

        if(!full(&s))
            push(&s,x);
        else
            printf("Stack is overflow\n");
        break;

    case 2:if(!empty(&s))
```

```
scanf("%d",&ch);

switch(ch)
{
    case 1:printf("enter value\n");
        scanf("%d",&x);

        if(!full(&s))
            push(&s,x);
        else
            printf("Stack is overflow\n");
        break;

    case 2:if(!empty(&s))
        {
        x=pop(&s);
        printf("popped is %d\n",x);
        printf("remaining elements\n");
        display(&s);
        }
        else
        {
        printf("stack is empty\n");
        }
        break;

    case 3: if(empty(&s))
       printf("stack is empty\n");
else
printf("stack elements are\n");
display(&s);
        break;
```



```
}
}
while(ch!=4);
}

void init(stack *s)
{
 s->top=-1;
}

int empty(stack *s)
```

```
}
}
while(ch!=4);
}

void init(stack *s)
{
 s->top=-1;
}

int empty(stack *s)
    {
      if(s->top == -1)
      {
      return(1);
      }
     return(0);
    }

int full(stack *s)
    {
     if(s->top == max-1)
     {
     return(1);
     }
     return(0);
    }

void push(stack *s, int x)
    {
    s->top = s->top+1;
    s->data[s->top]=x;
```



```
    }

int pop(stack *s)
    {
    int x;
    x=s->data[s->top];
    s->top=s->top-1;
    return(x);
    }

void display(stack *s)
    {
```

```
 if(s->top == max-1)
 {
 return(1);
 }
 return(0);
}

void push(stack *s, int x)
    {
    s->top = s->top+1;
    s->data[s->top]=x;
```

```
    }

int pop(stack *s)
    {
    int x;
    x=s->data[s->top];
    s->top=s->top-1;
    return(x);
    }

void display(stack *s)
    {
    int i;
     for(i=s->top;i>=0;i--)
     {
     printf("%d\n",s->data[i]);
     }
    }
```

ARRAY IMPLEMENTATION OF STACK https://youtu.be/-KQpk-dIA8s

```
#include<stdio.h>

int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);

int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);

   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t----------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("\n\t EXIT POINT ");
            break;
         }
         default:
         {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
         }

      }
   }
   while(choice!=4);
   return 0;
}
```

```
void push()
{
   if(top>=n-1)
   {
      printf("\n\tSTACK is over flow");

   }
   else
   {
      printf(" Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      stack[top]=x;
   }
}

void pop()
{
   if(top<=-1)
   {
      printf("\n\t Stack is under flow");
   }
   else
```

```
            break;
        }
        case 3:
        {
            display();
            break;
        }
        case 4:
        {
            printf("\n\t EXIT POINT ");
            break;
        }
        default:
        {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
        }

    }
}
while(choice!=4);
return 0;
}
```

```
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");

    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}

void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}

void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }

}
```

Applications of Stack

Well form-ness of Parenthesis
Infix to Postfix Conversion
Postfix Evaluation
Recursion


20200729_193038.jpg


20200729_193050.jpg


20200729_193059.jpg


20200729_193148.jpg


20200729_193202.jpg


20200729_193211.jpg


20200729_193221.jpg


WELLFORMNESS_OF_PARENTHESIS.docx


notes.jpg

```
default:
printf("\n Invalid choice \n");
break;
}
}

void push (int element)
{
 if (top==99)
 {
  printf ("stack is full \n");
  getch();
  exit(1);
 }
 top= top+1;
 stack[top] = element;
}

int Pop()
{ if (top == -1)
  {
   printf ("\n\t Stack is Empty \n");
   getch();
   exit (1);
  }
 return (stack [top--]);
 }

void display ()
 { int i;
   printf ("\n\t The various stack elements are:\n");
   for (i= top ; i>=0; i--)
   printf("%d\n", stack [i]);
}
```

Applications of Stack:

1- Well formness of Parenthesis

7-((X*((X+Y)/(J-3))+Y)/(4-2.5))

We want to ensure parentheses are nested correctly

1. There are an equal number of R & L parentheses
2. Every RP is preceded by a matching LP.

Give a parenthesis count
Increase the count on LP.
Decrease the count on RP.

1 Parenthesis count at the end of exp. must be 0.
2. Parenthesis count at each point should be nonnegate

7 - ((X * ((X + Y) / (J - 3)) + Y) / (4 - 2.5))
0 0 1 2 2 2 3 4 4 4 4 3 3 4 4 4 4 3 2 2 2 1 1 2 2 2 2 1 0

((A + B) x
1 2 2 2 2 1

A + B (x
0 0 0 1

) A + B (- (
-1 -1 -1 -1 0 0 1

(A + B)) - ((+ D
1 1 1 1 0 -1 -1 0 0 0 0

A stack may be used to keep track of the types of scopes encountered.
Whenever a scope opener is encountered, it is pushed onto the stack. Whenever a scope ender is encountered, the stack is examined.

- If the stack is empty
  Scope ender does not have a matching opener and the string is ∴ invalid.

- If the stack is nonempty,
  Pop the stack and compare with scope ender. If a match occurs, we continue.

  If match does not, string is invalid.

- When the end of the string is reached, the stack must be empty.
  Otherwise one or more scopes have been opened and have not been closed, hence the string is invalid.

{ x + (y - [a+b]) * c - [(d+e)]}/
(h - (j - (k - [l - m])))


```
valid = true;
s = the empty stack;

while (we have not reached end of the string)
{
    read the next (sym) of the string;

    if (sym == '(' || sym == '[' || sym == '{')
        push (s, sym);

    if (sym == ')' || sym == ']' || sym == '}')
        if (empty(s))
            valid = false;
        else {
            i = pop(s);

            if (i is not matching opener of
                sym)
            valid = false;
        }
}

if (! empty(s))
    valid = false;

if (valid)
    printf ("%s", "string is valid");

else
    printf ("%s", "string is invalid");
```

INFIX, POSTFIX AND PREFIX

	Infix	Postfix
(i)	A + B	AB+
(ii)	A+B-C	AB+C-
(iii)	(A+B) * (C-D)	AB+CD-*

(iv) Infix: A$B * C - D + E / F / (G + H)

Postfix: AB$C * D - EF / GH+ / +

(v) Infix: ((A+B) * C - (D - E)) $ (F+G)

Postfix: AB+C* DE- - FG+ $

(vi) Infix: A - B / (C * D $ E)

Postfix: A B C DE $ * / -

Infix	Prefix
A + B	+AB
A + B - C	-+ABC
(A+B) * (C-D)	*+AB-CD
A$B * C-D+E/F/(G+H)	+-*$ABCD//EF+GH
((A+B) * C-(D-E)) $ (F+G)	$-*+AB-DE+FG
A - B / (C * D $ E)	-A/B*C$DE

(i) A + B * C

	Sym	Postfix string	opstk
1	A	A	+
2	+	A	+
3	B	AB	+*
4	*	AB	+*
5	C	ABC	
6		ABC*	
7		ABC*+	

(ii) (A + B) * C

	Sym	Postfix string	opstk
1	(		(
2	A	A	(
3	+	A	(+
4	B	AB	(+
5	)	AB+	
6	*	AB+	*
7	C	AB+C	*
8		AB+C*	

3. ((A-(B+C)*D)$(E+F)

Sym	Postfix string	opstk
(		(
(		((
A	A	((-
-	A	((-(
(	A	((-(
B	AB	((-(
+	AB	((-(
C	ABC	((-(
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	
$	ABC+-D*	$
(	ABC+-D*	$
E	ABC+-D*E	$
+	ABC+-D*E	$
F	ABC+-D*EF	$
)	ABC+-D*EF+	$
	ABC+-D*EF+$	

Evaluating a Postfix Expression:

Consider the following infix stmt:

I: [(6-(2+3)*(8+8/2))]$2+3

Po: 623+- 382/+*2$3+

Symb	opnd1	opnd2	value	opndstk
6				6.
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3				1,3
8				1,3,8
2				1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2				7,2
$	7	2	49	49
3				49,3
+	49	3	52	52

02:11

2. I: 9 - ((3 * 4) + 8) / 4
Po: 9 3 4 * 8 + 4 / -

Sym	opnd1	opnd2	value	opndstk
9				9
3				9,3
4				9,3,4
*	3	4	12	9,12
8				9,12,8
+	12	8	20	9,20
4				9,20,4
/	20	4	5	9,5
-	9	5	4	4

3: I: (2+3) * 4 / 5
23 + 4 * 5 /

Sym	opnd1	opnd2	value	opndstk
2				2
3				2,3
+	2	3	5	5
4				5,4
*	5	4	20	20
5				20,5
/	20	5	4	4

Ans = 4

Recursion:
A definition, which defines an object in terms of a simpler case of itself, is called a recursion definition.

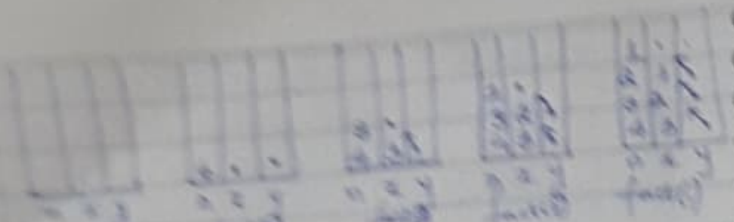
Let us see how recursive definition of the factorial function:

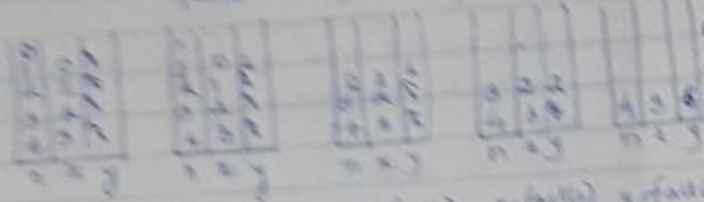
$5! = 5 * 4!$
$4! = 4 * 3!$
$3! = 3 * 2!$
$2! = 2 * 1!$
$1! = 1 * 0!$
$0! = 1$

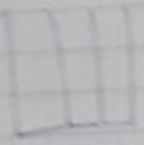
$0! = 1$
$1! = 1 \times 1 = 1$
$2! = 2 * 1! = 2 * 1 = 2$
$3! = 3 * 2! = 3 \times 2 = 6$
$4! = 4 * 3! = 4 \times 6 = 24$
$5! = 5 \times 4! = 4 * 24 = 120$

```
int fact(int n)
{
    int x, y;
    if (n==0)
        return(1);
    x = n-1;
    y = fact(x);
    return(n*y);
}
```

02.11.1







printf("%d", fact(3)) = 24.

Fibonacci Numbers in C:

Fibonacci sequence is the sequence of integers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

fib(n) = n if n==0 or n==1

fib(n) = fib(n-2) + fib(n-1) if n>=2

fib(2) = fib(1) + fib(0)
= 1 + 0
= 1

fib(3) = fib(2) + fib(1)
= fib(1) + fib(0) + fib(1)
= 1 + 0 + 1
= 2

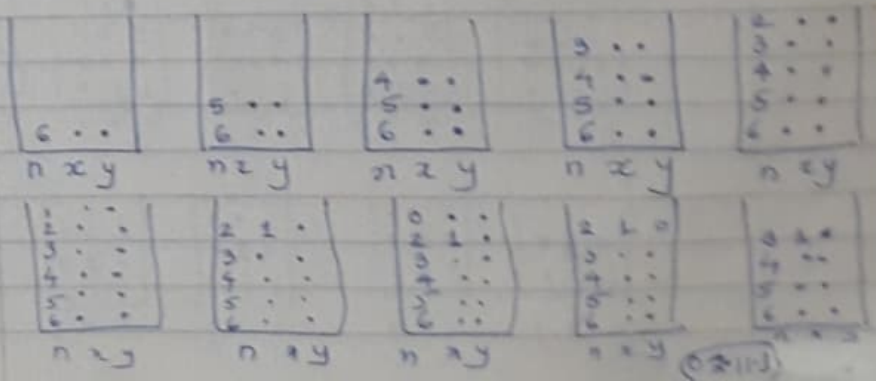
fib(4) = fib(3) + fib(2)
= fib(2) + fib(1) + fib(2)
= fib(1) + fib(0) + fib(1) + fib(2)
= 1 + 0 + 1 + 1
= 3

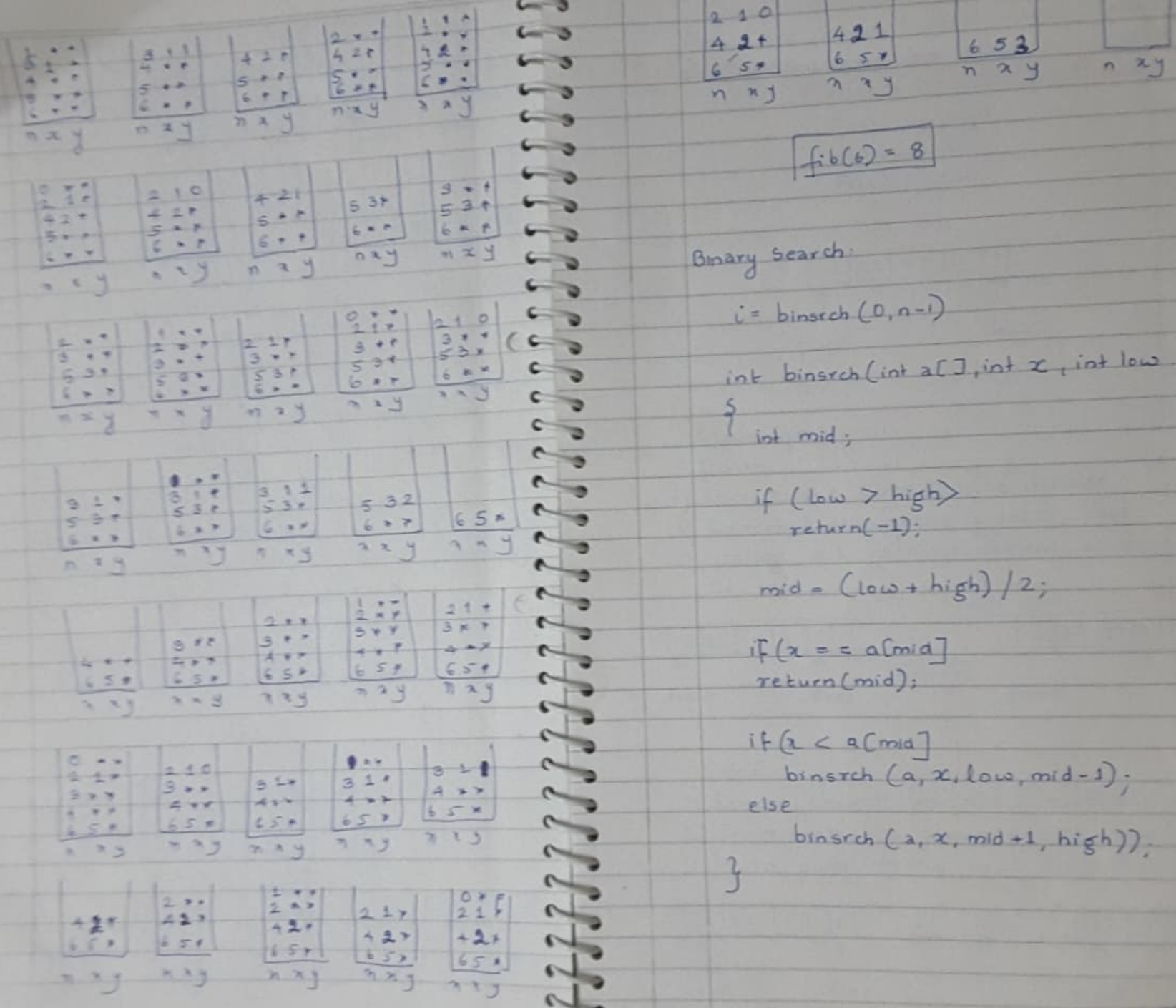
fib(5) = fib(4) + fib(3)
= fib(3) + fib(2) + fib(3)
= fib(2) + fib(1) + fib(2) + fib(3)
= fib(1) + fib(0) + fib(1) + fib(2) + fib(3)
= 1 + 0 + 1 + 1 + 2
= 5

fib(6) = fib(5) + fib(4)
= fib(4) + fib(3) + fib(4)
= fib(3) + fib(2) + fib(3) + fib(4)
= fib(2) + fib(1) + fib(2) + fib(3) + fib(4)
= fib(1) + fib(0) + fib(1) + fib(2) + fib(3) + fib(4)
= 1 + 0 + 1 + 1 + 2 + 3
= 8

```
int fib(int n)
{
  int x, y;
  if (n <= 1)
    return(n);

  x = fib(n-1);
  y = fib(n-2);

  return (x+y);
}
```

fib(6)





fib(6) = 8

Binary search:

```
i = binsrch (0, n-1)

int binsrch (int a[], int x, int low, int high)
{
   int mid;

   if (low > high)
      return(-1);

   mid = (low + high) / 2;

   if (x == a[mid]
   return(mid);

   if (x < a[mid]
      binsrch (a, x, low, mid-1);
   else
      binsrch (a, x, mid+1, high));
}
```

02.11.23

WELL FORM NESS OF PARENTHESIS

```
#include<stdio.h>
#include <stdbool.h>
#include <string.h>

    #define MAX_SIZE 100

    struct Stack{
       int top;
       char arr[MAX_SIZE];
    } st;


    void init()
{
       st.top = -1;
    }


    bool isEmpty()
{
       if(st.top == -1){
          return true;
       }
```

```
else
{
          return false;
       }
    }


    bool isFull()
{
       if(st.top == MAX_SIZE-1){
          return true;
       }
```

```
            return true;
        }
```

```
else
{
            return false;
        }
    }

    bool isFull()
{
        if(st.top == MAX_SIZE-1){
            return true;
        }
else
{
            return false;
        }
    }

    void push(char item)
{
        if(isFull()){
            printf("Stack is full");
        }
```



```
else
{
            st.top++;
            st.arr[st.top] = item;
```

```
void push(char item)
{
    if(isFull()){
        printf("Stack is full");
    }
```

```
else
{
    st.top++;
    st.arr[st.top] = item;
    }
}
```

```
char pop()
{   char a;
    if(isEmpty())
    {
        printf("Stack is empty");

    }
    else
{
    a=st.arr[st.top--];
    return a;
    }
}
```

```
}
```

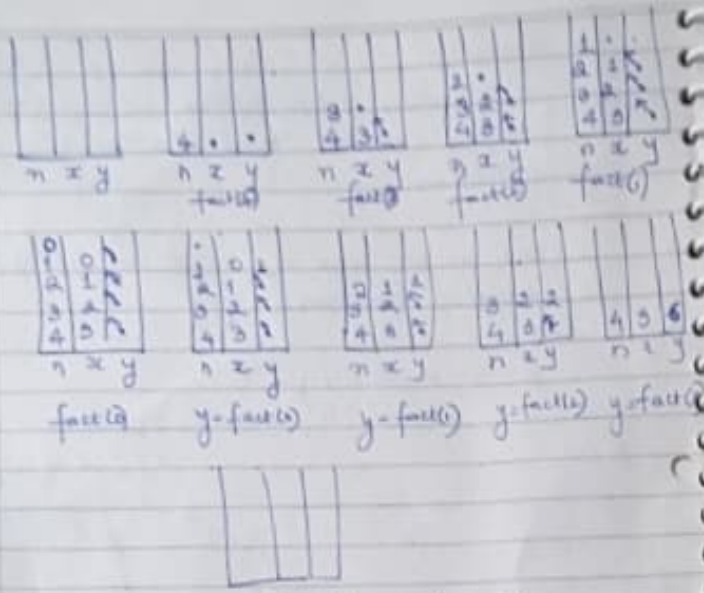
```
bool ArePair(char opening,char closing)
{
    if(opening == '(' && closing == ')') return true;
    else if(opening == '{' && closing == '}') return true;
    else if(opening == '[' && closing == ']') return true;
    return false;
}

void main()
{
  char in_expr[MAX_SIZE];
  int length=0,i,j;

  init();

  printf("Enter an expression to check:");
  gets(in_expr);

  length = strlen(in_expr);

  for(i=0;i<length;i++){
    if(in_expr[i] == '(' || in_expr[i] == '{' || in_expr[i] == '['){
```



```
        push(in_expr[i]);
    }
```

```
        printf("Enter an expression to check:");
        gets(in_expr);

        length = strlen(in_expr);

        for(i=0;i<length;i++){
            if(in_expr[i] == '(' || in_expr[i] == '{' || in_expr[i] == '['){
```

```
                push(in_expr[i]);
            }

else if(in_expr[i] == ')' || in_expr[i] == '}' || in_expr[i] == ']')
{
                char a = pop();
                printf("%c",a);
                printf("%c",in_expr[i]);
                if(!ArePair(a,in_expr[i])){
                    printf("\nResult - Invalid expression - Not a Balanced one !");
                    return 0;
                }
            }
        }
        if(isEmpty()){
            printf("\nResult - Valid expression - Perfectly Balanced !");
        }else{
            printf("\nResult - Invalid expression - Not a Balanced one !");
        }

    }
```



printf("%d", fact(4)) = 24

Fibonacci Numbers in C:

fibonacci sequence is the sequence of integers:

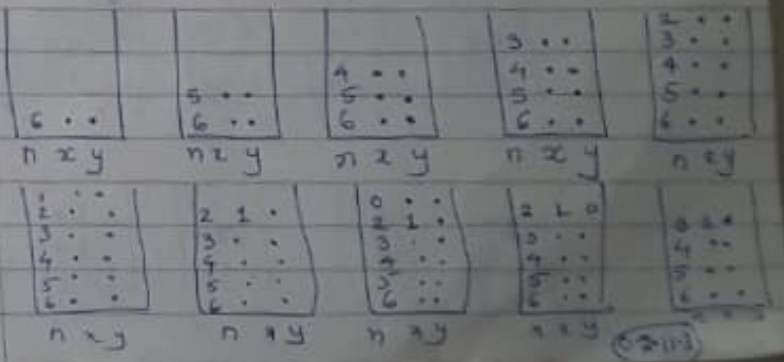
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

fib(n) = n if n == 0 or n == 1

fib(n) = fib(n-2) + fib(n-1) if n >= 2

fib(6) = fib(4) + fib(5)
= fib(2) + fib(3) + fib(5)
= fib(0) + fib(1) + fib(3) + fib(5)
= 0 + 1 + fib(3) + fib(5)
= 1 + fib(1) + fib(2) + fib(5)
= 1 + 1 + fib(2) + fib(5)
= 2 + fib(0) + fib(1) + fib(5)
= 2 + 0 + 1 + fib(5)
= 3 + fib(3) + fib(4)
= 3 + fib(1) + fib(2) + fib(4)
= 3 + 1 + fib(2) + fib(4)
= 4 + fib(0) + fib(1) + fib(4)
= 4 + 0 + 1 + fib(4)
= 5 + fib(2) + fib(3)
= 5 + fib(0) + fib(1) + fib(3)
= 5 + 0 + 1 + fib(3)
= 6 + fib(1) + fib(2)
= 6 + 1 + fib(2)
= 7 + fib(0) + fib(1)
= 7 + 0 + 1 =
= 8

```
int fib(int n)
{
  int x, y;
  if (n <= 1)
    return(n);

  x = fib(n-1);
  y = fib(n-2);

  return (x+y);
}
```



Introduction to Queue

Introduction to Queue
ADT of Queue
Operations on Queue
Array Implementation of Queue,

20200729_193528.jpg

20200729_193540.jpg

20200729_193553.jpg

QueueAsADT

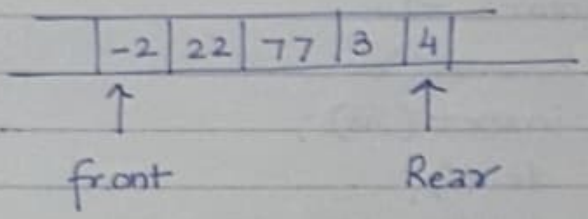
ARRAY IMPLEMENTATION OF QUEUE.docx

Queues

Queue is a linear data structure in which items are inserted at one end called 'Rear' and deleted from the other end called 'Front'.

FIFO (Fast In first Out)

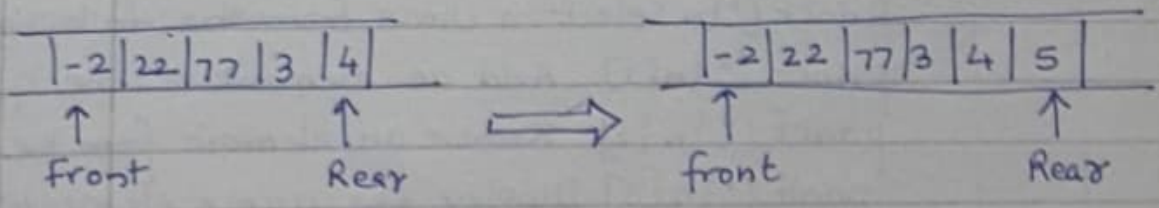
Queue of people
Assembly line
Print Queue



Queue Operations:

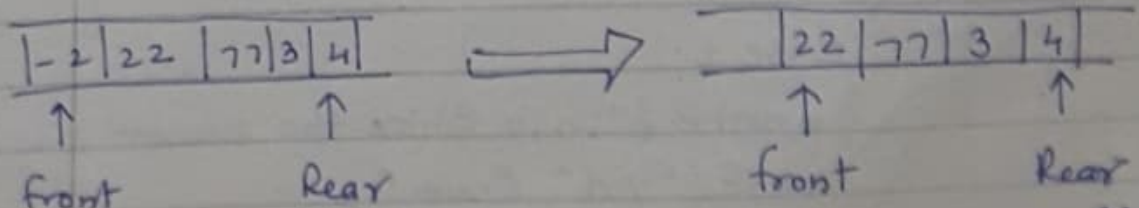
Insert

a) Receive the element to be inserted
b) Increment the queue pointer, rear
c) Storing the received element at new location of rear



Delete

1- Remove element from the front end of the queue.
2- Increment the queue pointer, front, to make it to point to the next element in the queue



02.13

Array Implementation of Queue:

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

int queue[100];
int front= -1;
int rear= -1;

void insert(int);
int del();
void display();

void main()
{
  int choice;
  int num1=0, num2=0;
  while (1)
  {
   printf("\n Select a choice from the following:");
   printf("\n[1] Add an element to the queue");
   printf("\n[2] Remove an element from the queue");
   printf("\n[3] Display the queue elements");
   printf("\n[4] Exit\n");
   printf("\n\t Your choice:");
   scanf("%d", &choice);

   switch(choice)
   {
    case 1:
    { printf("\n\t Enter the element to be added:");
      scanf("%d", &num1);
      insert(num1);
      break;
    }

    case 2:
    { num2 = del();
      if(num2 == -9999)
          ;
      else
        printf("\n\t %d element removed\n", num2);
    getch();
    break;
    }

    case 3:
    { display();
      getch();
      break;
    }

    case 4:
    exit(1);
    break;

    default:
    printf("\nInvalid choice!\n");
    break;
   }
  }
}
```

```
void insert (int e)
{ if (front == -1)
  { front = rear = front+1;
    queue [front] = e;
    return;
  }
  if (rear == 99)
  {
    printf ("queue is full\n");
    getch();
    return;
  }

  rear = rear+1;
  queue [rear] = e;
}
```

```
int del()
{
  int i;
  if (front == -1 && rear == -1)
  {
    printf ("\n\t queue is empty\n");
    getch();
    return (-9999);
  }

if (front != -1 && front == rear)
  {
    i = queue [front];
    front = -1;
    rear = -1;
    return(i);
  }

return(queue [front++]);
}
```

```
void .display()
{
  int i;
  if (front == -1)
  {
    printf ("\n\t queue is Empty!\n");
    return;
  }

printf("\n\t the various queue elements are:\n");

for (i = front ; i <= rear ; i++)
  printf ("\t %d", queue [i]);

}
```

QUEUE AS AN ABSTRACT DATA TYPE

- isFull() : This is used to check whether queue is full or not
- isEmpty() : This is used to check whether queue is empty or not
- enqueue(x) : This is used to insert x at the rear of the queue
- delete() : This is used to delete one element from the front of the queue
- front() : This is used to get the front most element of the queue
- size() : this function is used to get number of elements present into the queue

ARRAY IMPLEMENTATION OF Queue https://youtu.be/vPVWuoR43oM

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

#define MAX 50

void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      printf("1.Insert element to queue \n");
      printf("2.Delete element from queue \n");
      printf("3.Display all elements of queue \n");
      printf("4.Quit \n");
      printf("Enter your choice : ");
      scanf("%d", &choice);
      switch (choice)
      {
         case 1:
         insert();
         break;
         case 2:
```

```
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(1);
         default:
         printf("Wrong choice \n");
      } /* End of switch */
   } /* End of while */
} /* End of main() */
```

```
    printf("Enter your choice : ");
    scanf("%d", &choice);
    switch (choice)
    {
      case 1:
      insert();
      break;
      case 2:
      delete();
      break;
      case 3:
      display();
      break;
      case 4:
      exit(1);
      default:
      printf("Wrong choice \n");
    } /* End of switch */
  } /* End of while */
} /* End of main() */

void insert()
{
  int add_item;
  if (rear == MAX - 1)
  printf("Queue Overflow \n");
  else
  {
    if (front == - 1)
    /*If queue is initially empty */
    front = 0;
    printf("Inset the element in queue : ");
    scanf("%d", &add_item);
    rear = rear + 1;
    queue_array[rear] = add_item;
  }
} /* End of insert() */

void delete()
{
  if (front == - 1 || front > rear)
```

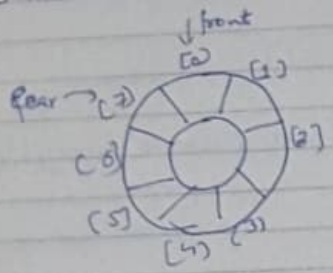
```
    else
    {
      if (front == - 1)
      /*If queue is initially empty */
      front = 0;
      printf("Inset the element in queue : ");
      scanf("%d", &add_item);
      rear = rear + 1;
      queue_array[rear] = add_item;
    }
  } /* End of insert() */
```

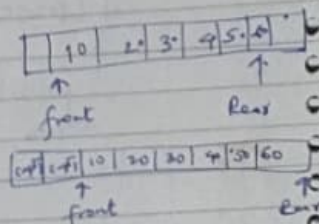
```
  void delete()
  {
    if (front == - 1 || front > rear)
    {
      printf("Queue Underflow \n");
      return ;
    }
    else
    {
      printf("Element deleted from queue is : %d\n", queue_array[front]);
      front = front + 1;
    }
  } /* End of delete() */

  void display()
  {
    int i;
    if (front == - 1)
      printf("Queue is empty \n");
    else
    {
      printf("Queue is : ");
      for (i = front; i <= rear; i++)
        printf("%d ", queue_array[i]);
      printf("\n");
    }
  } /* End of display() */
```

Circular Queues:

A circular queue is a queue whose start and end locations are logically connected with each other.

Circular queues remove one of the main disadvantages of array implemented queues in which a lot of memory space is wasted due to inefficient utilization.





Though 2 positions are empty it shows queue full bcz rear is at the end.

Insert Operation:

1. Checking whether the queue is already full.
2. Updating the rear pointer
   a) If the queue is empty, set front & rear to point to first location of the queue.
   b) If rear is pointing last location, set rear to point to the first location in the queue.
   c) If none of the above situations exist, increment rear pointer by 1.
3. Insert the new element at rear locn.

Delete operation:

1. check whether queue is already empty.
2. Retrieve the element at the front of the queue.
3. Update the front pointer.
   a) If Queue has only one element left, set front and rear to NULL.
   b) If front is pointing to last locn of the queue, set front to point first locn.
   c) If none of the above, simply increment front by 1.
4. Return the element retrieved from front locn.

```
void Insert (int element)
{
  if ((front == 0 && rear == N-1) ||
      front == rear + 1)
  {
    printf("Queue is full")
    getch()
    return;
  }

  if (front == -1)  /* Empty Queue */
  {
    front = 0;
    rear = 0;
  }

  else if (rear == N-1
    rear = 0;
```

```
else
 rear = rear + 1;

queue [rear] = element;
}


int del()
{ int c;
  if (front == -1)
  {
    printf (" Queue is empty ");
    getch();
    return (-9999);
  }

  c= queue [front];

  if (front == rear)
  {
    front = -1;
    rear = -1;
    return (c);
  }
else if (front == N-1)
 { front = 0;
   return (c);
 }
else
 {
   front = front + 1;
   return (c);
 }
}
```

DOUBLE-ENDED QUEUES:

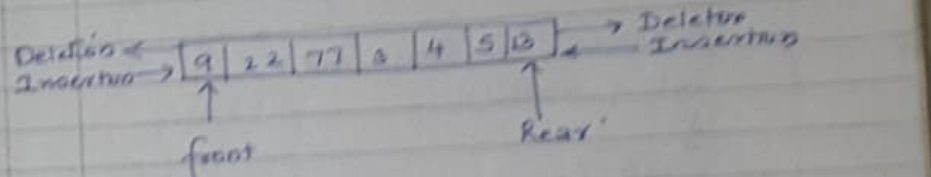
Special type of queue that allows insertion and deletion of elements at both ends, i.e., front & rear

Also called as deque.

Types:

1. Input-restricted deque: Allows deletion from both ends but insertion restricted at only one end.

2. Output-restricted deque: Allows insertion at both ends but deletion restricted at only one end.



Operations:

1. c_front : Insert at front end of the queue.
2. d_front : Delete from front end of the queue
3. c_rear: Insert at rear end of the queue
4. d_rear: Delete from rear end of the queue

```
/Insertion at front end */

 void c_front (int e)
 {
    if (front == -1)
    {
      front = rear = front + 1;
      queue [front] = e;
      return;
    }
```

```
if (front==0)
{
 printf("Queue is full.\n");
 getch();
 return;
}

 front= front-1;
 queue[front]=e;
}


/* Deletion at front end */
int d_front()
{
 int i;
 if (front==-1 && rear=-1)
 {
  printf("\n\t Queue is empty.\n");
  getch();
  return(-9999);
 }

 if (front==rear)
 {
   i=queue[front];
   front=rear=-1;
 return(i);
}

return (queue[front++]);
}
```

```
/* Insertion at rear end */

void i_rear(int element)

{
 if (rear==-1)
  {
    front=rear=rear+1;
    queue[rear]=element;
    return;
  }

 if (rear==99)
  {
    printf("Queue is full.\n");
    getch();
    return;
  }

 rear=rear+1;
 queue[rear]=element;
}


/* Deletion at rear end */

int d_rear()
{
  int i;
  if (front==-1 && rear==-1)
  {
    printf("\n\t Queue is Empty.\n");
    getch();
    return(-9999);
  }
```

02.23

```
if (front == rear)
{
 (= queue [rear];
front = -1;
rear = -1;
return (d);
}

return (queue [rear--]);
}
```

PRIORITY QUEUES:

A type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.

Rules for deletion of elements:

1. An element with highest priority is deleted before all other elements of lower priority.

2. If two elements have the same priority then they are deleted as per the order in which they were added into the queue (FIFO).

Implementation:

1 Elements can be added arbitrarily into the queue and deleted as per their priority values

OR

2 The elements may be sorted as per their priorities at the time of their insertion itself, and deleted in a sequential fashion.

Structure of queue should be such that each node is able to store both its value as well as its priority information.

```
struct queue
{
  int element;
  int priority;
  struct queue * next;
};
```

02.25

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6

int intArray[MAX];
int itemCount=0;

int peek()
 { return intArray[itemCount-1];
 }

bool isEmpty(){
    return itemCount==0;
}

bool isfull()
 {
   return itemCount==MAX;
}

int size()
 {
   return itemCount;
}


void insert(int data)
 {
   int i=0;

   if(!isFull())
   {
     if(itemCount==0)
      {
        intArray[itemCount++]=data;
      }
     else
      {
       for(i=itemCount-1;i>=0;i--)
        {
          if(data>intArray[i])
            {
              intArray[i+1]=intArray[i];
            }
          else
            {
              break;
            }
        }

       intArray[i+1]=data;
       itemCount++;
      }
   }
}

int removeData()
   {
     return intArray[--itemCount];
   }
```

```
int main()
 {
  insert(3);
  insert(5);
  insert(9);
  insert (1);
 insert(12);

  insert(15);

if (isFull ()){ printf ("Queue is full/\n"); }

int num = removeData();
printf (" Element removed: %d \n", num);

insert(16);
insert(17);
insert (18);

printf("Element at front: %d \n", peek());

printf(" ------------\n");
printf("index: 5 4 3 2 1 0\n");
printf(" - - - - - - -\n");
 printf(" Queue: ");

while ( !isEmpty()){
    int n = removeData();
    printf ("%d ", n);
}
}
```

Queue using structure:

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

Struct queue{
 int items[100];
 int front, rear;
};

int rem (struct queue*);
void insert (struct queue *, int);
void display (struct queue *);

void main()
{
 struct queue q;
  int k;

 q.front = q.rear = -1;

 insert (&q, 10);
 insert (&q, 20);
 insert (&q, 30);
 insert (&q, 40);

 display (&q);

 k= rem(&q);
 printf(

 k= rem(&q);
 printf(

display(&q);
}
```

Introduction of Double Ended Queue


20200729_193613.jpg


20200729_193623.jpg


20200729_193633.jpg

```
else
 rear = rear + 1;

queue [rear] = element;
}


int del()
{ int c;
  if (front == -1)
  {
    printf("Queue is empty");
    getche();
    return(-9999);
  }

  c= queue[front];

  if (front == rear)
  {
    front = -1;
    rear = -1;
    return(c);
  }
else if (front == N-1)
 { front = 0;
   return(c);
 }
else
 {
   front = front + 1;
   return(c);
 }
}
```

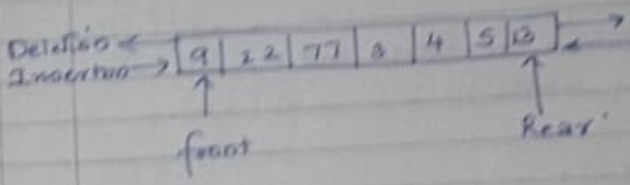
DOUBLE-ENDED QUEUES:

Special type of queue that allows insertion and deletion of elements at both ends, i.e., front & rear. Also called as deque.

Types:

1. Input-restricted deque: Allows deletion from both ends but insertion restricted at only one end.

2. Output-restricted deque: Allows insertion at both ends but deletion restricted at only one end.



Operations:

1. c_front : Insert at front end of the queue.
2. d_front : Delete from front end of the queue
3. c_rear: Insert at rear end of the queue
4. d_rear: Delete from rear end of the queue

```
/Insertion at front end */

void c_front (int e)
{
    if (front == -1)
    {
      front = rear = front + 1;
      queue[front] = e;
      return;
    }
```

```
if (front == 0)
{
 printf("Queue is full.\n");
 getch();
 return;
}

 front = front - 1;
 queue[front] = e;
}


/* Deletion at front end */
int d_front()
{
 int i;
 if (front == -1 && rear == -1)
 {
  printf("\n \t Queue is empty.\n");
  getch();
  return(-9999);
 }

 if (front == rear)
 {
  i = queue[front];
  front = rear = -1;
 return(i);
}

return (queue[front++]);
}
```

```
/* Insertion at rear end */

void i_rear (int element)
{
 if (rear == -1)
 {
  front = rear = rear+1;
  queue[rear] = element;
  return;
 }

 if (rear == 99)
 {
  printf ("Queue is full.\n");
  getch();
  return;
 }

 rear = rear+1;
 queue [rear] = element;
}


/* Deletion at rear end */

int d_rear()
{
 int i;
 if (front == -1 && rear == -1)
 {
  printf("\n \t Queue is Empty.\n");
  getch();
  return(-9999);
 }
```

02.23

```
if (front == rear)
{
 (* queue [rear];
front = -1;
rear = -1;
return (0);
}

return (queue [rear--]);
}
```

PRIORITY QUEUES:

A type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.

Rules for deletion of elements:

1. An element with highest priority is deleted before all other elements of lower priority.

2. If two elements have the same priority then they are deleted as per the order in which they were added into the queue (FIFO).

Implementation:

1 Elements can be added arbitrarily into the queue and deleted as per their priority values

OR

2 The elements may be sorted as per their priorities at the time of their insertion itself, and deleted in a sequential fashion.

Structure of queue should be such that each node is able to store both its value as well as its priority information:

```
struct queue
{
  int element;
  int priority;
  struct queue * next;
};
```

02.25

Introduction to Linked List

Introduction
Representation of Linked List
Linked List v/s Array,


20200729_193854.jpg


20200729_193907.jpg


20200729_193916.jpg


20200923_072158.jpg

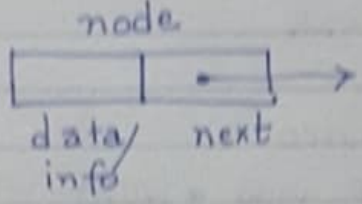

20200729_193940.jpg

Module 03

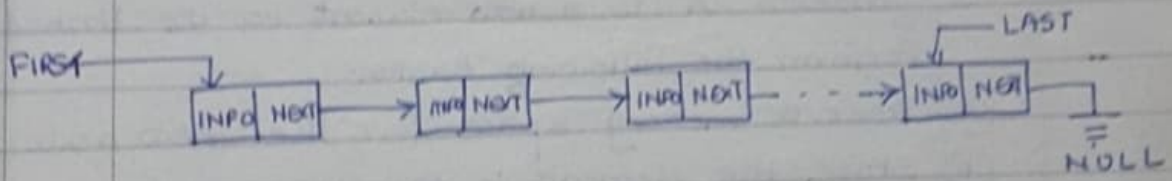
LINKED LISTS

Linked list is a collection of data elements stored in such a manner that each element points at the next element in the list.

These elements are referred as nodes.



NEXT contains the address of the next node.
The NEXT part of last node contains NULL.
The beginning of the list is indicated with the help of a special pointer called FIRST.
The end of the list is indicated by LAST.



Advantages of Linked Lists:

1. Linked lists facilitate dynamic memory mgt. by allowing elements to be added or deleted at any time during program execution.
2. Efficient utilization of memory space.
3. Easy to insert or delete elements

Disadvantages of Linked Lists:

1. Requires more memory space in comparison to an array element as it requires to store address of next node.
2. Accessing of an element is a little more difficult as there is no index identifier associated with each list element. Hence we have to traverse all.

03.01

LINKED LIST IMPLEMENTATION

- Declaring the list node
- Defining the linked list operations

Declaration:

```
struct node
{
   int INFO;
   struct node *NEXT;
};

typedef struct node NODE;
struct node *FIRST = NULL;
Struct  node *LAST = NULL;
```

Operations:

1. Insert: Adds a new element to the linked list.

Perform the following tasks:

(a) Reserve memory space for the new node.
(b) Store the element in the INFO part.
(c) Connect the new node to the existing nodes of list.

Depending on the location where the new node is to be added, there are 3 scenarios possible:

→ Insert at the beginning of the list.
→ Insert at the end of the list
→ Insert somewhere at the middle of the list

```
void insert (int value)
{
  struct node *PTR = (struct node*) malloc (
                  sizeof (struct node));

PTR -> INFO = value;
if (FIRST == NULL)
{
   FIRST = LAST = PTR;
   PTR -> NEXT = NULL;
}
else
 {
   LAST -> NEXT = PTR;
   PTR -> NEXT = NULL;
   LAST = PTR;
}
}
```

Delete operation: Removes an existing element from the linked list:

a) Identify the location of the element
b) Retrieve the value of element.
c) Link pointer of the preceding node is reset.

Depending upon the location of the element to be deleted, 3 scenarios possible:

a) Delete from beginning of the list
b) Delete from end of the list.
c) Delete from somewhere middle of the list.

```
int delete (int value)
{
   Struct node *LOC, *TEMP;
   int i;
   i = value;
   LOC = search(i);
```

03.03

```
if (LOC == NULL)
return (-9999);

if (LOC == FIRST)
{
 if (FIRST == LAST)
 FIRST = LAST = NULL;
 else
  FIRST = FIRST -> NEXT;
  return (value);
}

for (TEMP = FIRST; TEMP -> NEXT != LOC ; TEMP = TEMP -> NEXT)
 ;

TEMP -> NEXT = LOC -> NEXT;

if (LOC == LAST)
 LAST = TEMP;
 return (LOC -> INFO);
}
```

3. Search operation: Helps to find an element in the linked list.
   - → Traverse the list sequentially starting from the first node.
   - → Return the location of the searched node.
   - → Return a search failure if entire list is traversed without any match.

```
Struct node * search (int value)
{
 Struct node * PTR;

 if (FIRST == NULL)
 return (NULL);

 for (PTR = FIRST ; PTR != LAST ; PTR = PTR -> NEXT)
 if (PTR -> INFO == value)
 return (PTR);

 if (LAST -> INFO == value)
 return (LAST);
 else
 return (NULL);
}
```

4. PRINT: prints or displays the linked list elements on the screen.
   To print, traverse the linked list from start till end using NEXT pointers.

```
void print()
 {
  Struct node * PTR;
  if (FIRST == NULL)
 {
  printf ("\n\t Empty list ! !");
  return;
 }
```

```
printf("\n Linked list elements:\n");

if (FIRST == LAST)
{
  printf("\t %d", FIRST -> INFO);
return;
}

for (PTR = FIRST; PTR != LAST; PTR = PTR -> NEXT)
printf("\t%d", PTR -> INFO);

print( "\t%d", LAST -> INFO);
}
```

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

struct node
{
  int INFO;
  struct node * NEXT;
};

struct node * FIRST = NULL;
struct node * LAST = NULL;

void insert (int);
int delete (int);
void print (void);
struct node * search (int);
```

```
void main()
{
  int num1, num2, choice;
  struct node * location;

while (1)
{
 clrscr();
printf("\n\n Select an option\n");
printf("\n 1 - Insert");
printf("\n2 - Delete");
printf("\n3 - Search");
printf("\n4 - Print");
printf("\n5 - Exit");

printf("\n\n Enter your choice : ");
scanf("%d", &choice);

switch (choice)
{
  case 1:
  {
   printf("\n Enter the element to be inserted
          into the linked list:");
   scanf("%d", &num1);
   insert (num1);
   printf("\n %d Successfully inserted into LL!", num1);
   getch();
  break;
  }

  case 2:
  {
```

```
printf ("\Enter the element to be deleted from
        the linked list:");
scanf ("%d", &num1);
num2 = delete (num1);

if (num2 == -9999)
printf("\n\t %d is not present in the LL\n\n", num1);

else
printf("\n\t Element %d successfuly deleted \n\t", num2);
getch ();
break;
}

case 3:
{
printf ("\nEnter the element to be searched:");
scanf("%d", &num1);
location = Search (num1);

if (location == NULL)
printf ("\n\t %d is not present in LL\n\t", num1);

else
{ if (location == LAST)
printf ("\n\t Element %d is the last element in
        the list", num1);
else
printf("\n\t Element %d is present b4 element
        %d in the LL \n\t", num1, (location)->next)->info);
}
getch();
break;
}
```

```
case 4:
{
print();
getch();
break;
}

case 5:
{
exit(1);
break;
}

default:
{
printf ("\n Incorrect choice. Please try again:")
getch();
break;
}
}
}
```

Types of Linked List

Singly Linked List
Circular Linked List
Doubly Linked List,


20200729_194029.jpg


20200729_194046.jpg


20200729_194055.jpg


20200729_194102.jpg


20200729_194114.jpg


20200729_194240.jpg

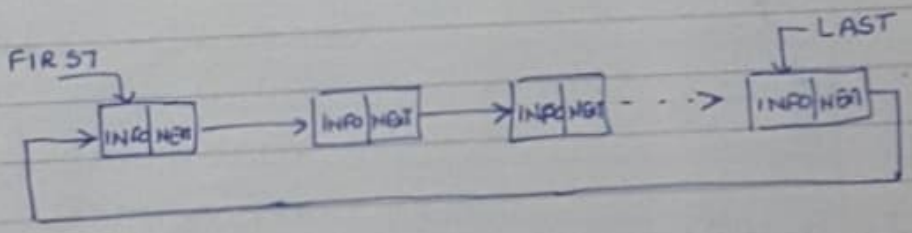
TYPES OF LINKED LISTS:

1. Singly linked list
   Each node points at the successive node. Thus, the list can be traversed in the forward direction.

2. Circular list
   The first and the last node are logically connected with each other. The NEXT part of last node contains the address of the FIRST node.

3. Doubly linked list
   A node points at both its preceding as well as succeeding nodes. Thus, the list can be traversed in both forward as well as backward direction.

CIRCULAR LINKED LIST:



Operations: Insert
Delete
Search
Print

1 Insert
Same manner as singly linked list but only exception is when element is inserted at the end of the list, its NEXT pointer is assigned the address of the first element in the list, thus ensuring it stays circular.

03.17

```
void insert (int value)
{
  struct cl_node *PTR;
  PTR = (struct cl_node*) malloc (sizeof (struct cl_node));
  PTR->INFO = value;
  if (FIRST == NULL)
  {
    FIRST = LAST = PTR;
    PTR->NEXT = FIRST;
  }
  else
  { LAST->NEXT = PTR;
    PTR->NEXT = FIRST;
    LAST = PTR;
  }
}
```

2. Delete: Same manner as a singly linked list. Only exception is when the element to be deleted at end of list, the NEXT pointer of second last node is assigned the address of first element to keep it circular.

```
int delete (int value)
{
  struct cl_node *LOC, *TEMP;
  int i;
  i = value;
  LOC = search(i);
  if (LOC == NULL)
  return (-9999);

  if (LOC == FIRST)
  {
    if (FIRST == LAST)
    FIRST = LAST = NULL;
    else
    { FIRST = FIRST->NEXT;
      LAST->NEXT = FIRST;
    }
    return (value);
  }

  for (TEMP = FIRST; TEMP->NEXT != LOC; TEMP = TEMP->NEXT)
  ;

  if (LOC == LAST)
  {
    LAST = TEMP;
    TEMP->NEXT = FIRST;
  }
  else
    TEMP->NEXT = LOC->NEXT;

  return (LOC->INFO);
}
```

3. Search:
Search can be started from anywhere in the list. An unsuccessful search is signified when the same node is reached from where the search was started.

03.19

```
Struct c1_node *search (int value)
{
  Struct c1_node *PTR;

  if (FIRST == NULL)
  return(NULL);

  if (FIRST == LAST && FIRST -> INFO == value)
  return (FIRST);

  for(PTR= FIRST; PTR1= LAST; PTR = PTR -> NEXT)
  If (PTR -> INFO == value)
  return(PTR);

  If ( LAST -> INFO == value)
  return(LAST);
  else
   return (NULL);
}
```

4. PRINT:
The circular nature allows us to start the print operation from anywhere in the list.

```
void print()
{
  Struct  c1_node *PTR;

  if (FIRST == NULL)
  {
    printf("\n \t Empty List!!");
  return;
}
printf ("\n Circular Linked list elements: \n");

if (FIRST == LAST)
{
  printf ("\t:%d", FIRST -> INFO);
  return;
}

for (PTR = FIRST; PTR1 = LAST; PTR = PTR -> NEXT)
printf ("t:%d", PTR -> INFO);

printf ("t:%d", LAST -> INFO);

}
```

Declaration:

```
Struct c1_node
{
  int INFO;
  Struct c1_node *NEXT;
};

Struct c1_node *FIRST = NULL;
struct c1_node *LAST = NULL;

void insert (int);
int delete (int);
void print (void);
Struct c1_node * search (int);
```

// Write the main program with switch case.

03.21

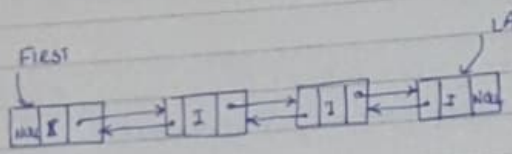
DOUBLY LINKED LIST:

Each node of a doubly linked list has three parts: INFO, NEXT & PREVIOUS.

INFO contains data element

NEXT address of the next ~~element~~ node.

PREVIOUS contains address of the next node.



FIRST NODE: PREVIOUS part is NULL.

LAST NODE: NEXT part is NULL.

```
struct dlnode
{
  int INFO;
  struct dlnode *NEXT;
  Struct dlnode *PREVIOUS;
};

struct dlnode *FIRST = NULL;
struct dl_node *LAST = NULL;
```

operations:

```
void insert (int);
int delete (int);
void print (void);
Struct dl_node *search (int);

void main()
{
  // write main program with switch-case.
}
```

```
// Insert function
void insert (int value)
{
Struct dl_node *PTR;

PTR = (struct dl_node *) malloc (sizeof (struct dl_node));

PTR -> INFO = value;

if (FIRST == NULL)
{
  FIRST = LAST = PTR;
  PTR -> NEXT = NULL;
  PTR -> PREVIOUS = NULL;
}

else
{
  LAST -> NEXT = PTR;
  PTR -> NEXT = NULL;
  PTR -> PREVIOUS = LAST;
  LAST = PTR;
}
}
```

03.23

```
//Delete function
int delete (int value)
{
  struct dl_node *LOC, *TEMP;
  int i;
  i= value;

  LOC =search(i);

  if (LOC = =NULL)
  return(-9999);

  if (LOC = = FIRST)
  {
   if (FIRST = = LAST)
   FIRST = LAST=NULL;
   else
   {
    FIRST -> NEXT-> PREVIOUS = NULL;
    FIRST = FIRST -> NEXT;
   }
  return(value);
}

for(TEMP= FIRST; TEMP -> NEXT !=LOC ;TEMP=TEMP->NEXT)
;
if (LOC = = LAST)
{
  LAST = TEMP;
  TEMP -> NEXT=NULL;
}
```

```
else
{
  TEMP -> NEXT = LOC -> NEXT;
  LOC -> NEXT -> PREVIOUS = TEMP;
}
return (LOC -> INFO);
}

//Search function
struct dl_node * search (int value)
{
  struct dl_node *PTR;

  if (FIRST = = NULL)
   return (NULL);

  if (FIRST = = LAST && FIRST -> INFO = = value)
   return (FIRST);

  for(PTR= FIRST; PTR != LAST; PTR = PTR -> NEXT)
  if (PTR -> INFO = = value)
  return (PTR);

  if (LAST -> INFO = = value)
  return(LAST);
  else
   return (NULL);
}
```

08·25

```
display (first);

create_poly (second, 1.5, 6);
create_poly (second, 2.5, 5);
create_poly (second, -3.5, 4);
create_poly (second, 4.5, 3);
create_poly (second, 65, 1);

printf("\n\n");
display (second);

printf("\n");
while (i++ < 79)
  printf("-");
printf("\n\n");

add_poly (first, second, total);
display (total);
}

void create_pdy (struct polynode *q, float x, int y
                 int y)
{
  struct polynode *temp;
  temp = q;

  if (q == NULL)
  {
    q = (struct polynode *) malloc (sizeof (struct polynode));
    temp = q;
  }

else
{
```

```
while ( temp -> next != NULL)
  temp = temp -> next;

temp -> next = (struct polynode *) malloc (
                sizeof (struct polynode));

temp = temp -> next;
}

temp -> coeff = x;
temp -> exp = y;
temp -> next = NULL;
}

void display( struct polynode *q)
{
  while (q != NULL)
  {
    printf ("%.1f x^%d : ", q -> coeff, q -> exp);
    q = q -> next;
  }

printf ("\b\b\b");
}

void add_poly (struct polynode *x, struct polynode *y,
               struct polynode *s)
{
  struct polynode *z;

  if (x == NULL && y == NULL)
    return;
```

Operations on Singly Linked List and Doubly Linked List


20200729_194122.jpg

20200729_194203.jpg

20200729_194221.jpg

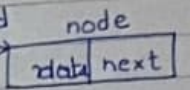
20200729_194231.jpg

```
// PRINT FUNCTION
void Print()
{
  struct dl_node *PTR;

  if (FIRST == NULL)
  {
    printf ("\n\t Empty List!!");
    return;
  }

  printf ("\n Doubly linked list elements: \n");

  if (FIRST == LAST)
  {
    printf ("\t %d", FIRST->INFO);
    return;
  }

  for (PTR = FIRST ; PTR != LAST ; PTR = PTR->NEXT)
    printf ("\t %d", PTR->INFO);

  printf ("\t %d", LAST->INFO);
}
```

Basic Linked List Operations:

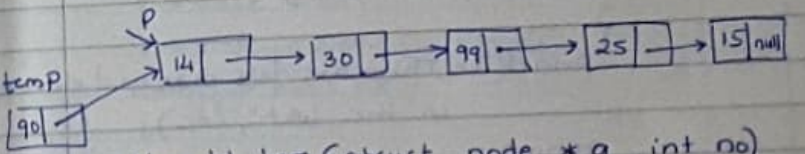
1. Creating a Linked List

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

typedef struct linked_list node;

node *head;

head = (node*) malloc (size of (node));
```



2. Inserting an Element:

a) Insertion at the Beginning of the List:

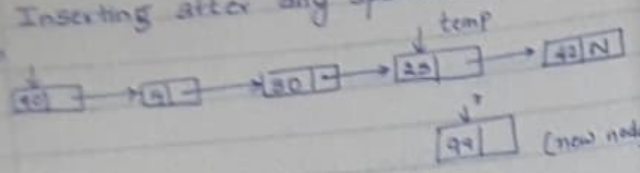



```
void add_beg (struct node *q, int no)
{
    node *temp;
    temp = (node*) malloc (sizeof ( node));
    temp->data = no;
    temp->next = q;
    q = temp;
}
```

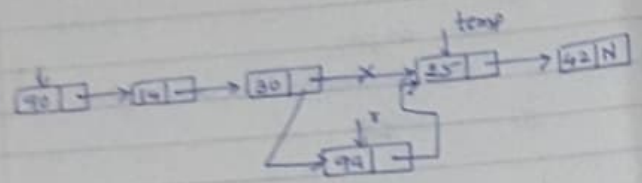
03.27

b) Inserting after any specified node:

Before:



After:



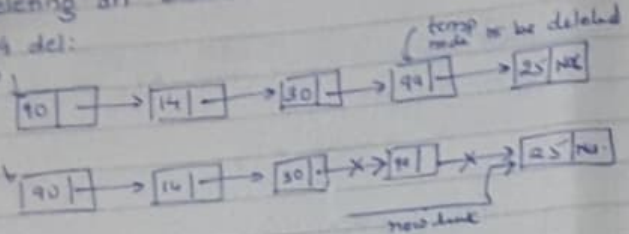

```
void add_after (node *q, int loc, int no)
{
  node *temp, *r;
  int i;

  temp = q;

  for(i=0; i<loc; i++)
  {
    temp = temp -> next;

    if (temp == NULL)
    {
    printf("\n There are less than %d elements
            in the list", loc);

    return;
    }
  }
  r= (node *) malloc (sizeof (node));
  r->data = no;
  r-> next = temp ->next;
  temp-> next = r;
}
```

c) Function to insert a node at the end of the list

```
void add_end ( node *q, int no)
{
  node *temp, *r;

  if( q == NULL)
  {
    temp= (node *) malloc (sizeof ( node))
    temp -> data = no;
    temp -> next = NULL;
    q= temp;
  }

  else
  {
    temp=q;

    while(temp -> next != NULL)
    temp =temp -> next;

    r = (node *) malloc (sizeof (node));
    r-> data = no;
    r-> next = NULL;
    temp -> next = r;
  }
}
```

Deleting an element:

B4 del:




```
void del (node *q, int no)
{
  node *old, *temp
  temp = q;

  while (temp != NULL)
  {
    if (temp -> data == no)
    {
      if (temp == q)
        q = temp -> next;

      else
        old -> next = temp -> next;

      free (temp);
      return;
    }
  else
  {
      old = temp;
      temp = temp -> next;
  }
}
printf ("\n Element %d not found", no);
}
```

Display the contents of the list:

```
void display (node * start)
{
  printf ("\n");

  while (start != NULL)
  {
    printf ("%d", start -> data);
    start = start -> next;
  }
}
```

Counting the number of nodes in a LL.

```
int count (node * q)
{
  int c=0;

  while (q != NULL)
  {
    q = q -> next;
    c++;
  }

  return c;
}
```

Reversing a linked List:

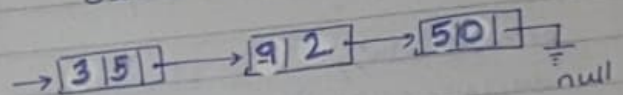
```
void reverse(struct node *x)
{
  struct node *q, *r, *s;

  q = x;
  r = NULL;

while (q != NULL)
  {
  s = r;
   r = q;

   q = q->next;

   r->next = s;
}

  x = r;
}
```

Application of Linked List:

Polynomial representation and Addition.

Consider the polynomial:

$$3x^5 - 9x^2 + 5$$




```
struct polynode
 {
   float coeff;
   int exp;
   struct poly *next;
};

void create_poly(struct polynode , float , int);
void display(struct polynode *);
void add_poly(struct polynode *, struct
              polynode *, struct polynode*);

void main()
{
  struct polynode *FIRST, *SECOND, *TOTAL;
  int i=0;

  first = second = total = NULL;

  create_poly(first, 1.4, 5);
  create_poly(first, 1.5, 4);
  create_poly(first, 1.7, 2);
  create_poly(first, 1.8, 1);
  create_poly(first, 1.9, 0);
```

09/03

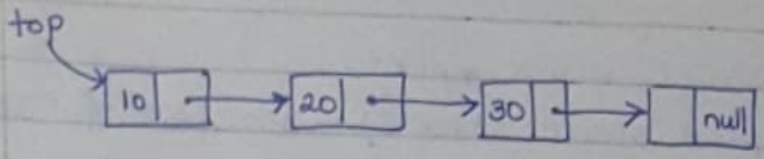
Stack and Queue using Singly Linked List

20200729_193952.jpg

20200729_194008.jpg

20200729_194020.jpg

Linked Implementation of stacks:



Dynamically allocating memory space at run time while performing stack operation.
Hence stack consumes only that much amount of space as is required for holding its data elements.

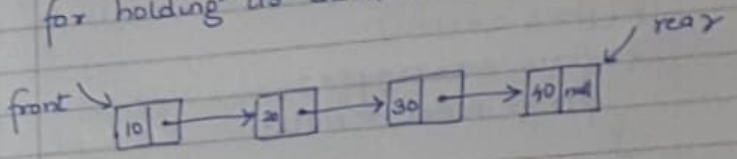
```
#include <stdio.h>

Struct stack
{
   int element;
   Struct stack * next;
} * top;

void pust (int);
Int pop ();
void display ();

void man()
{

   /* Write main program here */
   // with switch case

}
```

03.11

```
void push(int value)
{
 struct stack *ptr;
 ptr = (struct stack *) malloc(sizeof (struct stack));

 ptr -> element = value;
 ptr -> next = top
 top = ptr ;
 return;
}

int pop()
{
  if (top == NULL)
  { printf ("\n STACK is empty");
   getch ();
   exit(1).
  }
else
  { int temp = top -> element;
    top = top -> next;
    return (temp);
  }
}

void display()
 {
   struct stack *ptr1 = NULL;
   ptr1 = top;
   printf("\n The stack elements are:\n");
   while (ptr1 != NULL)
   { printf("%d\t", ptr1 -> element);
     ptr1 = ptr1 -> next;
 } }
```

Linked Implementation of Queues:

Involves dynamically allocating memory space at run time. Hence the queue consumes only that much amount of space as is required for holding its data elements.




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

Struct queue
 { int element;
   struct queue *next;
 };

struct queue *front = NULL;
struct queue *rear = NULL;

void insert (int);
int del();
void display (void);

void main()
{
  int num1, num2, choice;

  /* write main program here */
  // with switch choice.

}
```

0.3.13

```
void insert (int value)
{ struct queue *ptr= (struct queue *) malloc(
                    sizeof (struct queue));
ptr->element = value;

if (front == NULL)
 { front = rear = ptr;
   ptr -> next = NULL;
 }
else
 { rear -> next = ptr;
   ptr -> next = NULL;
   rear = ptr;
}}

int del()
{ int i

 if (front == NULL)
 return (-9999);

 else
 {
  i= front -> element;
  front = front -> next;
  return (i)
 }
}
```

```
void display()
{
struct queue *ptr = front;

 if(front == NULL)
 {
   printf ("\n It queue is Empty!!");
 }

else
{ printf ("\n Elements present in the Queue are\n");

 while (ptr != rear)
 {
   printf ("\t%d", ptr -> element);

   ptr = ptr -> next;
 }

 printf ("\t%d", rear -> element);
}
}
```

Singly Linked List Application

Polynomial Representation
Polynomial Addition.


20200729_194231.jpg


20200729_194240.jpg


20200729_194249.jpg


20200729_194300.jpg

Reversing a linked List:

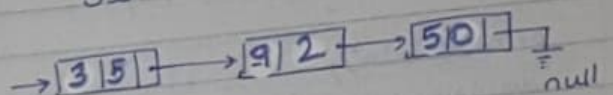
```
void reverse(struct node *x)
{
  struct node *q, *r, *s;

  q = x;
  r = NULL;

  while (q != NULL)
    {
      s = r;
      r = q;

      q = q->next;

      r->next = s;
    }

  x = r;
}
```

Application of Linked List:

Polynomial representation and Addition.

Consider the polynomial:

$$3x^5 - 9x^2 + 5$$




```
struct polynode
  {
    float coeff;
    int exp;
    struct poly *next;
  };

void create_poly(struct polynode , float , int);
void display( struct polynode *);
void add_poly (struct polynode *, struct
               polynode *, struct polynode*);

void main()
{
  struct polynode *FIRST, *SECOND, *TOTAL;
  int i=0;

  first = second = total = NULL;

  create_poly ( first, 1.4, 5);
  create_poly (first, 1.5, 4);
  create_poly (first, 1.7, 2);
  create_poly (first, 1.8, 1);
  create_poly (first, 1.9, 0);
```

```
display (first);

create_poly (second, 1.5, 6);
create_poly (second, 2.5, 5);
create_poly (second, -3.5, 4);
create_poly (second, 4.5, 3);
creati_poly (second, 65, 1);

printf("\n\n");
display (second);

printf("\n");
while (i++ < 79)
  printf("-");
printf ("\n\n");

add_poly (first, second, total);
display (total);
}

void create_pdy (struct polynode *q, float x, int y)
{
  struct polynode *temp;
  temp = q;

  if (q == NULL)
  {
    q = (struct polynode *) malloc (sizeof (struct polynode));
    temp = q;
  }

else
{
  while ( temp -> next != NULL)
    temp = temp -> next;

  temp -> next = (struct polynode *) malloc (
                  sizeof (struct polynode));

  temp = temp -> next;
}

temp -> coeff = x;
temp -> exp = y;
temp -> next = NULL;
}

void display( struct polynode *q)
{
  while (q != NULL)
  {
    printf ("%.1f x^%d : ", q -> coeff, q -> exp);
    q = q -> next;
  }

printf ("\b\b\b");
}

void add_poly (struct polynode *x, struct polynode *y,
               struct polynode *s)
{
  struct polynode *z;

  if (x == NULL & y == NULL)
    return;
```

```
while (x != NULL && y != NULL)
{
  if (s == NULL)
  {
    s = (struct polynode *) malloc (sizeof (struct polynode));
    z=s;
  }

  else
  { z -> next = (struct polynode *) malloc (sizeof (
                                    struct polynode));


    z = z -> next;
  }

  if (x -> exp < y -> exp)
  {
    z -> coeff = y -> coeff;
    z -> exp = y -> exp;
    y = y -> next;
  }

  else
  {
    if (x -> exp > y -> exp)
    {
      z -> coeff = x -> coeff;
      z -> exp = x -> exp;
      z = z -> next;
    }
    else
    {
      if (x -> exp == y -> exp)
      {
        z -> coeff = x -> coeff + y -> coeff;
        z -> exp = x -> exp;

        x = x -> next;
        y = y -> next;
      }
    }
  }
}

while (x != NULL)
{
  if (s == NULL)
  {
    s = (struct polynode *) malloc (sizeof (struct polynode));
    z=s;
  }
  else
  {
    z -> next = (struct polynode *) malloc (sizeof (struct polynode));
    z = z -> next;
  }

  z -> coeff = x -> coeff;
  z -> exp = x -> exp;
  z = x -> next;
}

while (y != NULL)
{
  if (s == NULL)
  {
```

```
s=(struct polynode*) malloc(sizeof(struct polynode));
z=s;
}
else
{ z->next = malloc(sizeof(struct polynode));
  z=z->next;
}

z->coeff = y->coeff;
z->exp = y->exp;
y = y->next;
}

z->next = NULL
}
```

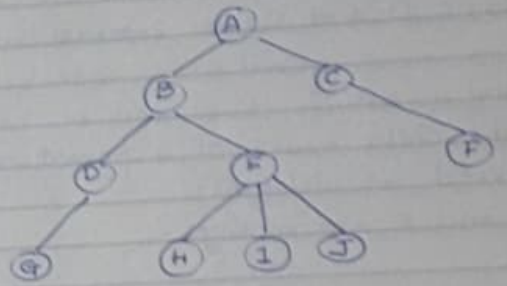

Trees

Tree is a non-linear data structure which stores the data elements in a hierarchical manner. Each node of the tree stores a data value, and is linked to other nodes in a hierarchical fashion.

Basic Concept:

A tree is defined as a finite set of elements or nodes such that

1. One of the nodes present at the top of the tree is marked as root node.
2. The remaining elements are partitioned across multiple subtrees present below the root node.



Tree T

A: Root node.

others are child nodes and some are leaf nodes

Tree Terminology:

1. Root:
   Top node in the tree. → (A)

2. Parent:
   A node that has one or more child nodes present below.
   → (B) is parent of (D) & (E)