Keerthana Ashok

Page No.:
Date:
YOUVA

UNIT - 01

Object Oriented Programming

* **Procedure oriented Programming: (POPs)**

- **To display a message:**

```
#include <iostream>
using namespace std  (to include standard libraries)
int main()
{
    cout << "NIE" << '\n' or endl;  (Printf)
}          ↳ insertion operator
```

NOTE:

Operator overloading: It is when same operator has more than one operation or function.

EX: << : insertion operator
lesser than
shift left

- **Addition of two numbers:**

```
# include <iostream>
using namespace std
int main()
{
    int a, b; (run time assignment)
    cout << "Enter the two numbers" << endl;
    cin >> a >> b;   → extraction operator
    int c = a+b;
    cout << "The sum is" << c << endl;
}
```

NOTE: A variable is a name given to a memory location.

Page No.
Date

* Types of operators:

1. Arithmetic operators:

+, -, *, /, %

```
# include <iostream>
using namespace std
int main()
{
    int a=10, b=2; (compile time assignment)
    Cout << " The sum is" << a+b <<endl;
    Cout << " The difference is" << a-b <<endl;
    cout << " The product is" << a*b <<endl;
    cout << " The quotient is" << a/b << endl;
}
```

2. Increment / Decrement operators:

++, --

a++, a-- : post increment and decrement respectively
++a, --a : pre increment and decrement respectively.

Ex: cout << a++ <<--a << endl; 10 Ex: a=10
cout << a-- <<++a << endl; 10

3. Shorthand Operators:

a = a+b : a+=b
a = a-b : a-=b
a = a*b : a*=b
a = a/b : a/=b
a = a%b : a%=b

4. Comparision and logical operators:

Comparision / Relational operators

<, <=, >, >=, ==, !=

Logical operators

&&, ||, !

Page No.:
Date:

★ If ; if else ; nested if else:

```
if (condition)
{
.....
}
else
{
.....
}
```

```
if (condition)
{
.....
}
else if (condition)
{
.....
}
else
{
.....
}
```

Ex: Valid or invalid date:

```
#include <iostream>
using namespace std
int main()
{
    int date;
    cout << " Enter the date" << endl;
    cin >> date;
    if ( date > 31 || date < 1)
        cout << " The date is invalid" << endl;
    else
        cout << " The date is valid" << endl;
}
```

5. Ternary operators:

(condition) ? statement 1 : statement 2
True statement False statement

Ex: (a > b) ? cout << " greater" : cout << " lesser";

★ Loops:

1. While loop:

The condition is checked at the entry hence called as entry controlled loop.

```
syntax: initialization
        while (condition)
        {
          ------
          ------
          updation (increment or decrement)
        }
```

Ex: To display a message multiple times.

```
int i=0;
while (i<=9)
{
   cout << "NIE" << endl;
   i++;
}
```

2. Do while loop:

The condition is checked at the exit hence called as exit controlled loop.

```
syntax: initialization
        do
        {
          -----
          -----
          updation
        } while (condition);
```

Ex: To display a message multiple times.

```
int i = 0;
do
{
   cout << "NIE" << endl;
   i++;
} while (i<=9);
```

Keerthana Ashok

3. **For loop:**

syntax: for (initialization ; condition ; updation)
{

}

Ex: To display a message multiple times.

```
for (int i=0; i<=9; i++)
{
    cout<<"NIE"<<endl;
}
```

(or)

```
int i=0
for (; i<=9;)
{
    cout<<"NIE"<<endl;
    i++
}
```

NOTE: for (;;) → infinite loop.

* **Array:**

It is a collection of homogenous data elements in a continuous memory locations.

- single dimensional array : a[ ]

  Ex: a[2] = 30

	0	1	2	3	4
0	10	20	30	40	50

- Two dimensional array: a[row][col]

  Ex: a[0][2] = 30

	0	1	2	3	4
1	60	70	80	90	10

* **Functions:**

syntax: returntype function_name (parameters)
{
-----
statements
}

Ex: 1.

```
void disp(); (function defination/prototype)
int main()
{
    disp(); (calling function)
}
void disp() (called function)
{
    cout<<"NIE"<<endl;
}
```

call by value

2.
```
int add (int, int);
int main()
{ int a, b;
  cout<<" Enter the two numbers "<<endl;
  cin >> a >>b;
  int c = add (10, 20);
  cout << c<< endl;
}
int add (int a, int b)
{
  return (a+b);
}
```

called and calling function have different memory location.
The variation in the called function does not affect the main function

3. call by reference

```
int add (int, int);
int main()
{
  int a, b;
  cout<<" Enter the two numbers"<<endl;
  cin>>a>>b;
  int c = add (a, b);
  cout<<c<<endl;
}
int add (int &a, int &b)
{
  return (a+b);
}
```

called and calling function have same memory location.

4. Call by pointer

```
int addd(int, int, int);
int main()
{
    int a,b;
    cout << "Enter the two numbers "<<endl;
    cin >> a >> b;
    int c = add (&a, &b);
    cout << c <<endl;
}
int add (int *a, int *b)
{
    return (a+b);
}
```

- Global and local scope

Ex:
```
int a=30;
int main()
{
    int a=10;
    cout<<a<<endl; // 10
}
cout<<a<<endl; // 30
```

Ex:
```
int a;
int main()
{
    int a=10;
    cout<<a<<endl; // 10
}
cout<<a<<endl; // 0
```

NOTE: when a variable is declared globally and is not assigned any value, then:

a. when int type it is assigned 0.

b. When char type it is assigned '\0' (null character)

* **Switch**

syntax:
```
switch (condition)
{
    case 'condition' : statements;
                       break;
    case 'condition' : statements;
                       break;
    .
    .
    .
    default : statements;
}
```

Ex: (For a range)
```
int a;
cin >> a
switch (a)
{
    case 1... 99 : cout << "a" << endl;
                   break;
    case 0 :       cout << "b" << endl;
                   break;
    case 100 :     cout << "c" << endl;
                   break;
    default :      cout << "10" << endl;
}
```

* **Pointers:**

Pointer is a variable which holds the address of the other variable.

Ex:
```
*a = &b;
cout << b << endl; // value of b
cout << &b << endl; // address of b
cout << *a << endl; // address of b
```

* Object Oriented Programming : (OOPs)

- Features:

1. - classes and objects:

data type	attributes	actions

user defined :	bird
	object
		fly();
		eat();
		sing(); } classes

builtin :	int	signed() unsigned()

Objects are the reference to classes.
The memory is allocated to the classes only when an object is created.
classes are the user defined data types which are the actions of the attributes.

Ex:

```
#include "iostream"
using namespace std;
class human          (class name)
{
public: int a, b;    data members
public: void talk()                    member functions.
    {
        cout << "Talking" << endl;
    }
    void walk()
    {
        cout << "walking" << endl;
    }
};
int main()
{
    human features;   (object)
```

```
features.talk();
features.walk();
features.a;
features.b;
}
```

- Access Specifier

a. Public: it can be accessed by all classes as well as the main function.

b. Private: It cannot be accessed by any function outside the class (even main function cannot access).
Private is the default access specifier.

c. Protected: The main function cannot access the class member. But it can be accessed by only certain other mentioned classes.

NOTE

Initializing a variable inside the class is not possible. This is because no memory is allocated unless an object is created. the initialization can be done in the object.

- Write an OOPS program to compute sum of two numbers using add function and display the result using disp function.

```
#include "iostream"
using namespace std;
class addition
{
    public: int a, b, c;
    public: void add()
        {
            cout << " enter the numbers" << endl;
            cin >> a >> b;
            c = a+b;
        }
```

```
    void disp()
    {
        cout << "Sum is" << c << endl;
    }
};
int main()
{
    addition obj;
    obj.add();
    obj.disp();
}
```

2. Constructor and Destructor:

- Constructor

- It is a special member function and there is no return type.
- It is always specified by public access specifier.
- It is used for initialization.
- Memory allocation function.
- Function name will be same as the class name.

Ex:

```
class human
{
    int a,b;
public: human()        // default constructor
    {                  // constructor function
    }
    void walk()
    {
    }
    void talk()
    {
    }
}
```

member functions

A constructor is automatically invoked when an object is created.

constructors can be:

1. default
2. parameterized
3. copy

Ex: Parameterized constructor

```
class const
{
  public: int a, b;
  public: const (int a, int b) → parameterized constructor
        {
          cout << a+b << endl;
        }
};
int main()
{
   const obj (10, 20);
}
```

Ex: copy constructor:

```
class const
{
  public: int a, b;
  public: const (int a, int b)
        {
          cout << a+b << endl;
        }
        const (const &d) → copy constructor
        {
          cout << d.a + d.b << endl;
        }
};
int main()
{
```

```
const obj(10,20);
const obj1(obj);
}
```

- **Destructor**
  - It is a special member function and there is no return type similar to that of a constructor.
  - Memory deallocation function.
  - Function name will be same as the class name.
  - There is only one kind of destructor: default destructor

Ex:

```
class const
{
  public: int a,b;
  public: const() → default constructor
          {
            cout<<a+b<<endl;
          }
          ~const() → default destructor
          {
            cout<<"destroy"<<endl;
          }
};
int main()
{
  const obj;
  obj.a=10;
  obj.b=20;
}
```

As soon as an object is created a default constructor and a default destructor is invoked

Destructor is represented by tilde '~' symbol

When multiple objects are created equal number of constructors and destructors are invoked. The destructor is in the reverse order as that of the order of the constructor. which is sequential order.

UNIT - 02

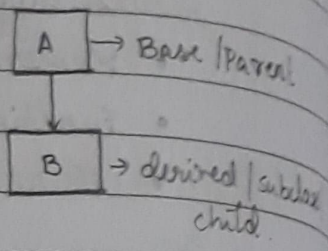
Inheritance and Polymorphism

3. Inheritance:

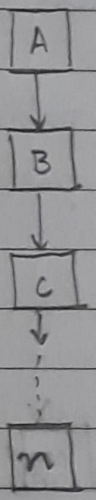
It is the relationship between the classes, where the classes derive the features from the existing classes.

Types of inheritance

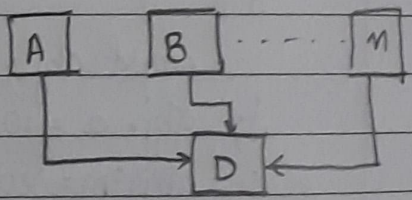
- single inheritance:
  There is only one level of inheritance. It is the relationship between two classes.



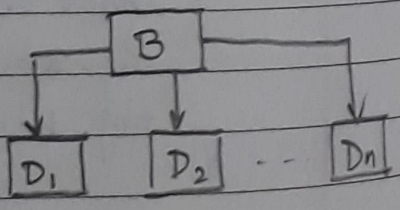
- Multilevel inheritance:
  There is more than one level of inheritance. For n classes there are n-1 level of inheritance.



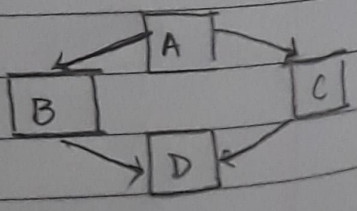
- Multiple inheritance:
  There are multiple base / parent classes but has a single derived class.



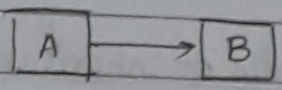
- Hierarchial inheritance:
  There is only one base / parent class but multiple derived classes.



- Hybrid inheritance:
  It is a combination of all inheritances. It is also called as diamond inheritance.



→ Ex: 1. single / simple Inheritance




```
class A
{
public: int a, b;
Public: void add()
      {
        cout << a+b << endl;
      }
};
class B : Public A
{
public: int c, d;
public: void sub()
      {
        cout << c-d << endl;
      }
};
int main()
{
  B obj;
  obj. a = 10;
  obj. b = 20;
  obj. c = 10;
  obj. d = 5;
  obj. add()
  obj. sub()
}
```

Using constructor

```
class A
{
public: int a, b;
public: A()
      {
        a=10, b=20;
        cout << a+b << endl;
      }
};
class B
{
public: int c, d
public: B()
      {
        c=10, d=5;
        cout << a-b << endl;
      }
};
int main()
{
    B obj;
}
```

NOTE:

	Private	Protected	Public
same class	✓	✓	✓
derived class	x	✓	✓
other class	x	✓	✓

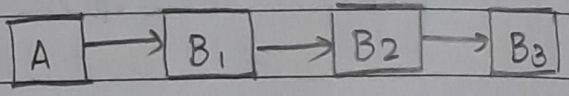
Modes of inheritance:

1. Private mode inheritance
2. Public mode inheritance
3. Protected mode inheritance.

Mode of inheritance

Base	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	cannot be accessed	cannot be accessed	cannot be accessed

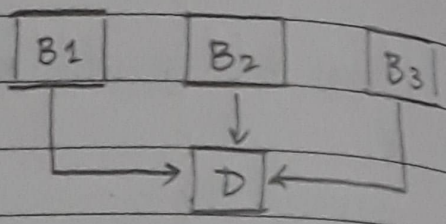
→ Ex: 2. Multilevel Inheritance




```
class A
{
public : int a;
public : void k()
        {
            a=10;
            cout << a << endl;
        }
};
class B1 : Public A
{
public : int b;
public : void m()
        {
            b=20;
            cout << b << endl;
        }
};
```

```
class B2: public B1
{
public: int c;
public: void n()
        {
            c=30;
            cout<<c<<endl;
        }
};
class B3: public B2
{
public: int sum;
public: void add()
        {
            sum = a+b+c;
            cout<< sum << endl;
        }
};
int main()
{
   B3 obj;
   obj.z();
   obj.m();
   obj.n();
   obj.add();
   return 0;
}
```

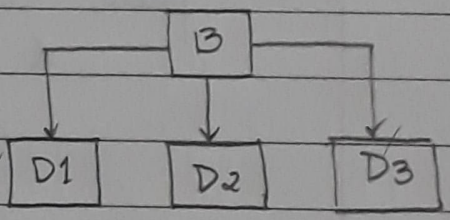
→ Ex: 3. Multiple Inheritance:




```
class B1
{
public: int b;
public: void m()
        {
            b=20;
            cout << b << endl;
        }
};
class B2
{
public: int c;
public: void n()
        {
            c=10;
            cout << c << endl;
        }
};
class B3
{
public: int a;
public: void l()
        {
            a=30;
            cout << a << endl;
        }
};
class D: public B1
         public B2
         public B3
```

```
{
  public: int sum;
  public: void add()
        {
          sum = a+b+c;
          cout << sum << endl;
        }
};
int main()
{
   D obj;
   obj. m();
   obj. n();
   obj. l();
   obj. add();
   return 0;
}
```

→ Ex, 4. Hierarchical Inheritance

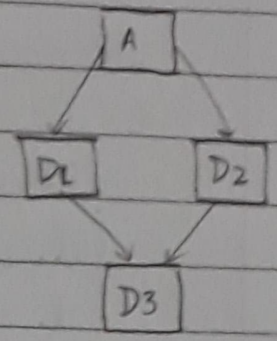



```
class B
{
  public: int s;
  public: void disp()
        {
          s = a+b+c;
          cout << s << endl;
        }
};
class D1 : Public B
{
   public: int a;
   public: void one()
```

```
    {
        a=10;
        cout << a << endl;
    }
};
class B2 : Public B
{
    public: int b;
    public: void two()
            {
                b=20;
                cout << b << endl;
            }
};
class D3: Public B
{
    public: int c;
    public: void three()
            {
                c=20;
                cout << c << endl;
            }
};
int main()
{
    D x;
    D y;
    D z;
    x.disp();
    y.disp();
    z.disp();
}
```

Keerthana Ashok

→ Ex: 5. Hybrid Inheritance

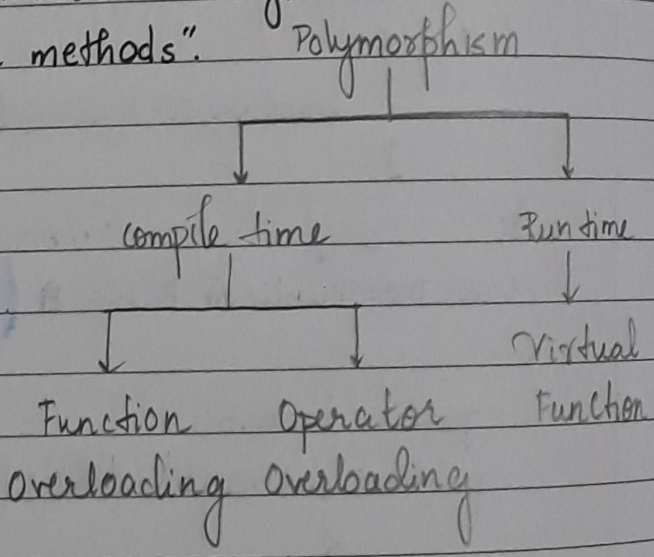



```
class A
{
  public: int a, b;
  public: int add(int a, int b)
  {
    cout<<a+b<<endl;
  }
};
class D1: Virtual Public A
{
  public: int a,b;
  public: int sub(int a, int b)
    {
      cout<<a-b<<endl;
    }
};
class D2: Virtual Public A
{
  public: int a,b;
  public: int mul(int a, int b)
    {
      cout<<a*b<<endl;
    }
};
class D3: Public D1, Public D2
{
  public: void disp()
    {
      cout<<"The result is"<<endl;
    }
};
```

The class A (base class) is made virtual; if not there will be two copies of add() function in class D3 (due to D1 and D2) hence the base class is a virtual class/abstract class

```
int main()
{
    D3 d;
    d.add(10,20);
    d.sub(10,5);
    d.mul(10,10);
    d.disp();
}
```

4. **Polymorphism:**
A single entity consisting of many forms.
Ex: A function with the same name but performing different functions. (function overloading)
"one interface, multiple methods".



- **Function overloading:**

```
Ex: class A
    {
        public: int a, b;
public: virtual int add(int a, int b)
            {
                cout<< a+b << endl;
            }
        virtual void disp()
            {
                cout << "Result is" << endl;
            };
```

```
class B : public A
{
  public: int add(int c, int d)
          {
            cout << c+d << endl;
          }
          void disp()
          {
            cout << "Result" << endl;
          }
};
int main()
{
    A a;          // object
    A *p;
    p = &a;                          → this operator
    p -> add(10,20); // class A
    p -> disp();
    B b;
    p = &b;
    p -> add(10,20); // class B   (if base class is not virtual
    p -> disp();                    this will be class A function)
}
```

NOTE:

Function overloading: It is the process of using the same name for two or more functions. For overloading each redefination of the function must either use different types of parameters or a different number of parameters.

- operator Overloading

operator overloading is the method by which we can change the function of some specific operators to do some ~~specific~~ different task.

syntax:

```
return_type class_name :: operator op (Argument list)
{
    function_body;
}
```

a. Binary Operator Overloading

there should be one argument to be passed. It is overloading of an operator operating on two operands.

Ex:

```
class A
{
    int im;
    int em;
    public: A (int inmark, int extmark)
            {
                im = inmark;
                em = extmark;
            }
    A operator +(A a)
    {
        A temp;
        temp.im = im + a.im;
        temp.em = em + a.em;
        return temp;
    }
    void disp()
    {
        cout << im << em << endl;
    }
};
```

Keerthana Ashok

```
int main()
{
    A a(10,20), b(20, 30);
    A c=a+b;
    c.disp()
    return 0;
}
```

b. Unary Operator Overloading

It is overloading of an operator operating on one operand.

Ex:
```
class A
{
    int im;
    int em;
    public : A(int immark, int extmark)
    {
        im = intmark;
        em = extmark;
    }
    void operator ++()
    {
        ++im;
        ++em;
    }
    void disp()
    {
        cout<<im<<em<<endl;
    }
};
int main()
{
    A a(10,20);
```

Keerthana Ashok

```
++a;
a.disp()
return 0;
}
```

- write a program where pre increment operator overloading performs post decrement operation.

```
Class A
{
  int im;
  int em;
  public: A (int inmark, int extmark)
        {
          im = inmark;
          em = extmark;
        }
        A operator ++()
        {
          im--;
          em--;
        }
        void disp()
        {
          cout << im << em << endl;
        }
};
int main()
{
  A a(10,20);
  ++a;
  a.disp()
  return 0;
}
```

- **write a program to overload assignment operator:**

```
class A
{
  int im;
  int em;
  public: A (int inmark, int extmark)
        {
          im = inmark;
          em = extmark;
        }
        A operator = (A a)
        {
          im = a.im;
          em = a.em;
        }
        void disp()
        {
          cout<<im<<em<<endl;
        }
};
int main()
{
   A a(10,20);
   A b;
   b = a;
   a.disp()
   b.disp()
   return 0;
}
```

- Friend Function

Ex:
```
class A
{
    int a,b; // private
    public: void add()
            {
                a=10, b=20;
                cout << a+b << endl;
            }
    friend void sub(A a)
};
void sub(A a)
{
    a= 10, b=5; // accessed as it is friend function
    cout << a.a-a.b << endl;
}
int main()
{
    A a;
    a.add();
    sub(a);
}
```

- write a program to accept two numbers and find the average and largest of these numbers using friend function:

```
class comp
{
    int A,B;
    public: comp(int a, int b)
            {
                A=a;
```

Keerthana Ashok

```
        B=b;
    }
friend int avg (comp x);
friend int larg (comp x);
};
int avg (comp x)
{
   return ((x.A + x.B)/2);
}
int larg (comp x)
{
    if (x.A > x.B)
      return x.A;
    else
      return x.B;
}
int main()
{
   comp h(10,20);
   cout << "avg is" << avg(h) << endl;
   cout << "larg is" << larg(h) << endl;
   return 0;
}
```

* **Function Template :**

syntax:
```
template < class typename>
returntype function_name (typename parameter)
{
    function body;
}
```

Ex:
```
#include <iostream>
template <class T>
T max(T x, T y)
{
  return((x>y)?x:y);
}
int main()
{
  cout<<max(100,200)<<endl;
  cout<<max(10.0,20.5)<<endl;
  cout<<max('a','b')<<endl;
}
```

```
int max(int x, int y)
{
  return((x>y)?x:y);
}
float max(float x, float y)
{
  return((x>y)?x:y);
}
char max(char x, char y)
{
  return((x>y)?x:y);
}
```

- Write a program to define a function template to swap the contents of two data items of class type int, float and double:

```
#include <iostream>
template <class T>
T swap(T x, T y)
{
  T temp;
  temp = x;
  x = y;
  y = temp;
}
int main()
{
  cout<<swap(10,20)<<endl;
  cout<<swap(10.5,6.3)<<endl;
}
```

Page No.
Date

Using classes

```
#include "iostream"
using namespace std;
template <class T>
class numbers
{
    T n1, n2, total;
  public: void getdata();
          void sum;
};
template <class T>
void numbers <T>:: getdata()
{
    cout << "Enter two numbers" << endl;
    cin >> n1 >> n2;
}
template <class T>
void numbers <T>:: sum()
{
   total = n1 + n2;
   cout << total << endl;
}
int main()
{
   numbers <int> iob;
   numbers <float> fob;
   iob.getdata();
   iob.sum();
   fob.getdata();
   fob.sum();
}
```

Keerthana Ashok

UNIT - 03

Linked List

* **New and Delete Operator:**

C++ provides two dynamic allocation operators:

- **new operator**: It allocates memory and returns a pointer to the start of it. It is used to increase the size of the datatype and the address is stored in the pointer.
- **delete operator**: It frees memory that was previously allocated i.e., it is used for deallocation of memory.

These two operators are used to allocate and free memory at run time.

- write a program to allocate dynamic memory for an array of 5 elements and deallocate them.

```
#include <iostream>
using namespace std;
int main()
{
    int *p;
    *p = new int[5];
    p[1] = 5;
    p[2] = 10;
    p[3] = 15;
    p[4] = 20;
    p[5] = 25;
    cout<<p[1]<<p[2]<<p[3]<<p[4]<<p[5]<<endl;
    delete [5] p;
    return 0;
}
```

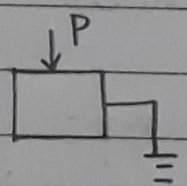
* **Linked Lists:**

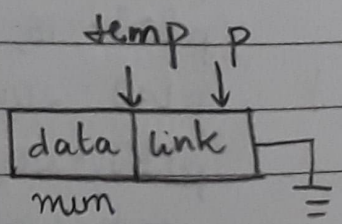
A linked list is a linear datastructure in which the elements are not stored at contiguous memory locations. The elements are linked in a linked list using pointers. A linked list consists of nodes where each node contains a data field and a reference (link) to the node in the list.

There are three different types of linked lists:

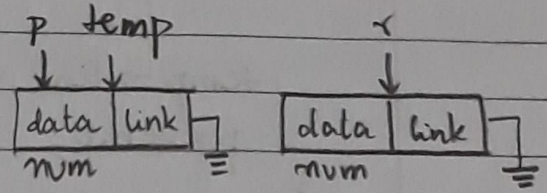
- **Singly Linked List:**

```
class sll
{
    struct node
    {
        int data;
        node *link;
    }*p;
(or) (node *p;)
};
public: sll()
        {
            p=NULL;
        }
int create (int num)
{
    node *temp,*r;
    if (p=NULL)
    {
        temp = new node;
        temp → data = num;
        temp → link = NULL;
        p= temp;
    }
```

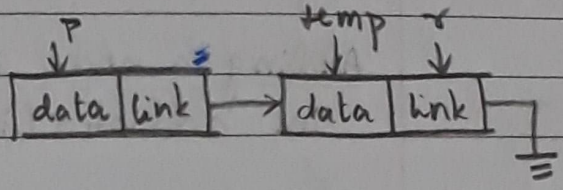




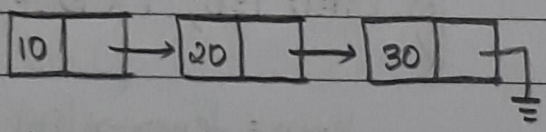
Keerthana Ashok




```
else
{
    r = new node;
    r->data = num;
    r->link = NULL;
    temp->link = r;
    temp = temp->link;
  }
}
```

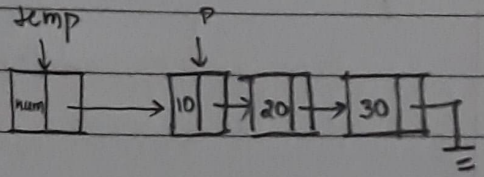



```
void disp()
{
    node *temp = p;
    while (temp != NULL)
    {
        cout << temp->data << endl;
        temp = temp->link;
    }
  }
};
int main()
{
    ll o;
    o.create(10);
    o.create(20);
    o.create(30);
    o.disp();
}
```

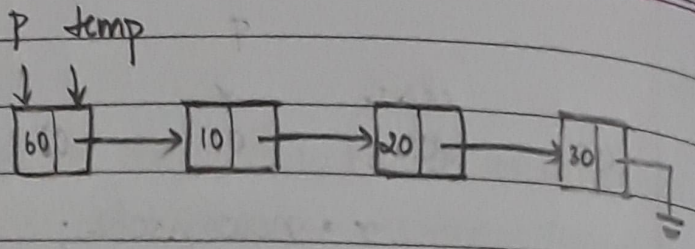


Adding a node at the beginning of the linked list

```
int addatbeg(int num)
{
    node *temp;
    temp = new node;
    temp->data = num;
```

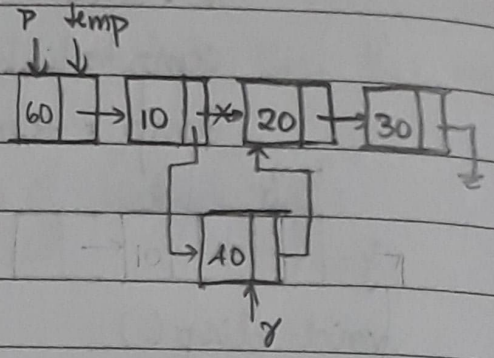


```
    temp -> link = p;
    p = temp;
}
```



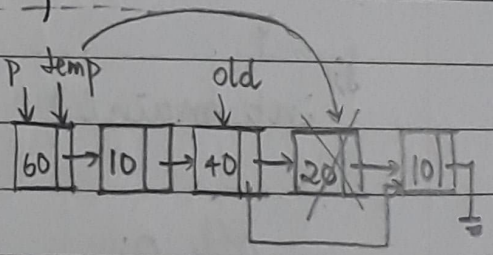
Adding a node in an intermediate position

```
int addinter(int num, int loc)
{
    node *temp = p, *r;
    for (int i=1; i<loc-1; i++)
    {
        temp = temp -> link;
    }
    r = new node;
    r -> data = num;
    r -> link = temp -> link;
    temp -> link = r;
}
```



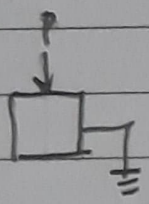
Deleting a node from a particular position

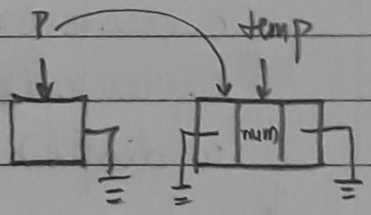
```
int delinter(int loc)
{
    node *temp = p, *old;
    for (int i=1; i<loc-1; i++)
    {
        temp = temp -> link;
    }
    old = temp;
    temp = temp -> link;
    old -> link = temp -> link;
    delete temp;
}
```

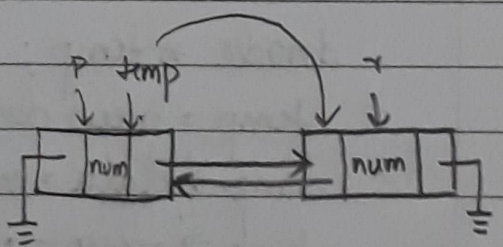


- Double Linked List:

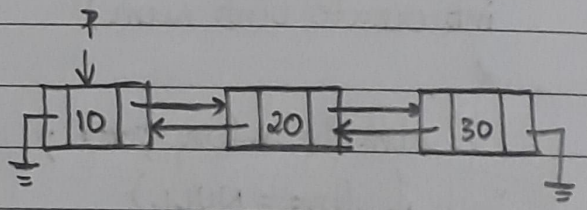
```
class dll
{
    struct dnode
    {
        int data;
        dnode *prev;
        dnode *next;
    };
    dnode *p;
    public: dll()
            {
                p = NULL;
            }
    int create(int num)
    {
        dnode *temp, *r;
        if (p == NULL)
        {
            temp = new dnode;
            temp->data = num;
            temp->prev = NULL;
            temp->next = NULL;
            p = temp;
        }
        else
        {
            r = new dnode;
            r->data = num;
            r->next = NULL;
            r->prev = temp;
            temp->next = r;
```





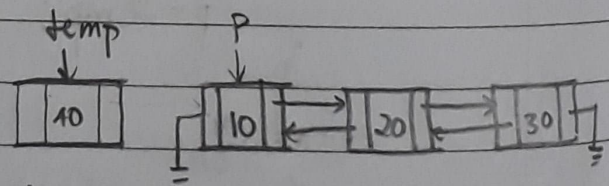


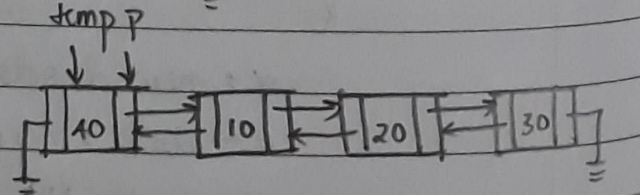
```
        temp = temp -> next;
    }
}
void disp()
{
    dnode *temp = p;
    while (temp != NULL)
    {
        cout << temp -> data << endl;
        temp = temp -> next;
    }
}
};
int main()
{
    dll o;
    o.create(10);
    o.create(20);
    o.create(30);
    o.disp()
}
```



Adding a node at the beginning of the linked list:

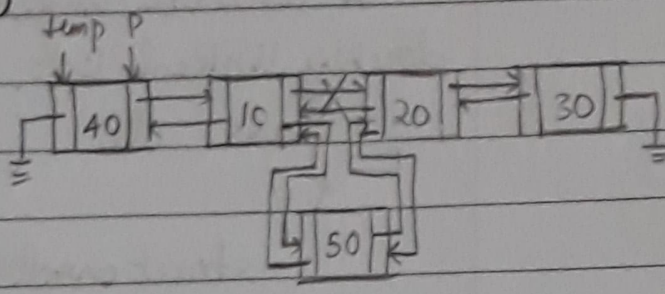
```
int addbeg(int num)
{
    dnode *temp;
    temp = new dnode;
    temp -> data = num;
    temp -> prev = NULL;
    temp -> next = p;
    p -> prev = temp;
    p = temp;
}
```





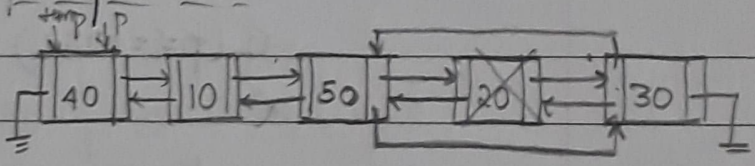
Adding a node at the intermediate position

```
int addinter(int num, int loc)
{
    node *temp = p, *r, *q;
    for (int i=1 ; i<loc-1; i++)
    {
        temp = temp -> next;
    }
    q = temp -> next;
    r = new node;
    r -> data = num;
    r -> prev = temp;
    r -> next = q;
    q -> prev = r;
    temp -> next = r;
}
```



Deleting a node from a particular position

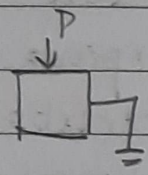
```
void del(int loc)
{
    node *temp=p, *old;
    for (int i=1 ; i<loc-1 ; i++)
    {
        temp = temp -> next;
    }
    old = temp;
    temp = temp -> next;
    old -> next = temp -> next;
    temp -> next -> prev = old -> next -> prev;
    delete temp;
}
```

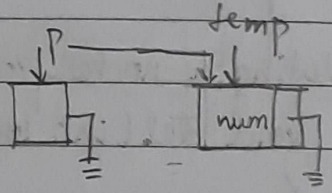


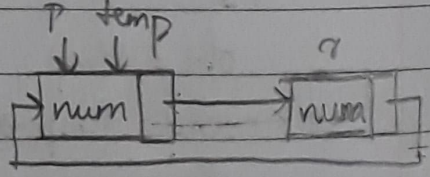
- Circular Linked List:

A circular linked list can be singly linked list or a doubly linked list.

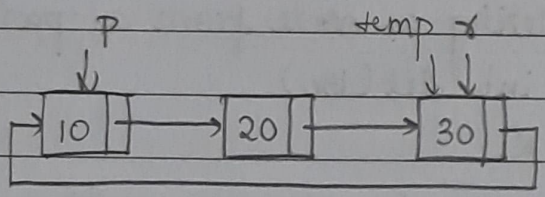
```
class cll
{
    struct cnode
    {
      int data;
      cnode *link;
    }*p,*r;
    public: cll()
            {
                p = NULL;
            }
    int create(int num)
    {
      cnode *temp;
      if(p==NULL)
      {
        temp = new cnode;
        temp -> data = num;
        temp -> link = NULL;
        p = temp;
      }
      else
      {
        r = new cnode
        r -> data = num;
        r -> link = p;
        temp -> link = r;
        temp = temp -> link;
      }
}
```





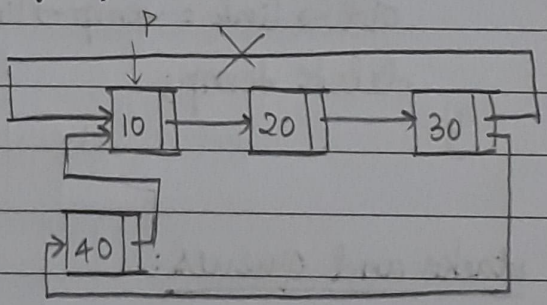


```
void disp()
{
  cnode *temp=p;
  do
  {
    cout<<temp->data<<endl;
    temp=temp->link;
  }while(temp!=p)
}
};
int main()
{
  cll o;
  o.create(10);
  o.create(20);
  o.create(30);
  o.disp();
}
```



Adding a node at the beginning of the linked list

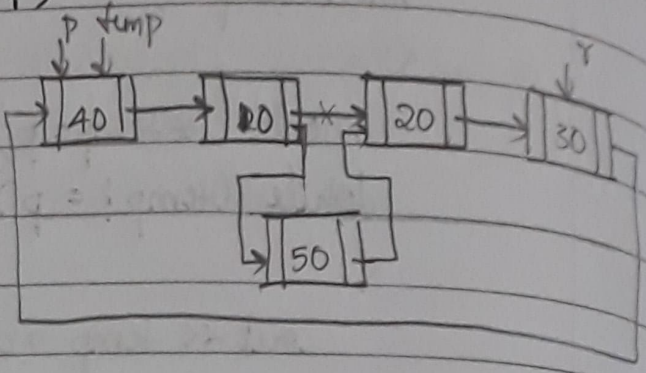
```
int addbeg(int num)
{
  cnode *temp;
  temp=new cnode;
  temp->data=num;
  temp->link=r->link;
  r->link=temp;
  p=temp;
}
```



Adding a node at the intermediate position of the linked list:

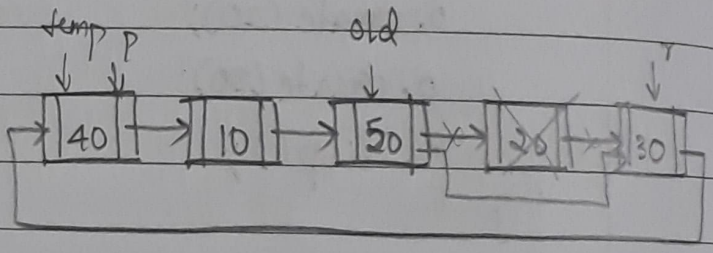
```
int addinter(int num, int loc)
{
  cnode *temp=p,*r;
```

```
for (int i=1; i<loc-1; i++)
{
    temp = temp -> link;
}
r = new cnode;
r -> data = num;
r -> link = temp -> link;
temp -> link = r;
}
```



Deleting a node from a particular position:

```
int del(loc)
{
    cnode *temp=p, *old;
    for (int i=1; i<loc-1; i++)
    {
        temp = temp -> link;
    }
    old = temp;
    temp = temp -> link;
    old -> link = temp -> link;
    delete temp;
}
```



* Stacks and Queues:

- Stack: stack is collection of elements that follows the Last In First Out (LIFO), which means element which is inserted most recently will be removed first.

A stack has a restriction that insertion and deletion of elements can be done from only one end of the stack which is the top end. The element at the top position is called top element. Insertion of element is called PUSH and deletion is called POP.

- Queue: Queue is a data structure that follows the First In First Out (FIFO) where the element added first in the queue will be the one to be removed first. Elements are always added to the back and removed from the front.

STACK:

```
#include <iostream>
using namespace std;
class stack
{
   struct snode
   {
      int data;
      snode *link;
   };
   snode *top;
   public: stack()
           {
              top = NULL;
           }
   int push(int data)
   {
      snode *temp;
      temp = new snode;
      temp -> data = data;
      temp -> link = top;
      top = temp;
   }
   int pop()
   {
      if (top == NULL)
      cout << "stack is empty" << endl;
```

```
        else
        {
            cout<<"The popped elements are:"<<endl;
            cout << top->data << endl;
            top= top->link;
        }
    }
    void disp()
    {
        snode *ptr;
        if (top==NULL)
        cout <<" stack is empty" <<endl;
        else
        {
            ptr = top;
            while (ptr!==NULL)
            {
                cout << ptr->data <<endl;
                ptr = ptr-> link;
            }
        }
    }
};
int main()
{
    stack o;
    o.disp();
    o.push(10);
    o.push(20);
    o.push(30);
    o.pop();
    o.disp();
}
```

output:

```
stack is empty.
The popped elements are:
30
20
10
```

QUEUE:

```
#include <iostream>
using namespace std;
class queue
{
  struct qnode
  {
    int data;
    qnode *link;
  };
  qnode *front, *rear, *temp;
  public queue()
        {
          front = NULL;
          rear = NULL;
        }
  void in(int data)
  {
    if (rear == NULL)
    {
      rear = new qnode;
      rear->link = NULL;
      rear->data = data;
      front = rear;
    }
    else
    {
      temp = new qnode;
      rear->link = temp;
      temp->data = data;
      temp->link = NULL;
      rear = temp;
  } }
```

```
void out()
{
  temp = front;
  if (front == NULL)
  cout << "Queue is empty" << endl;
  else if (temp -> link != NULL)
  {
    temp = temp -> link;
    cout << "Element out from queue is:" << front -> data << endl;
    front = temp;
  }
  else
  {
    cout << "Element out from queue is:" << front -> data << endl;
    front = NULL;
    rear = NULL;
  }
}
void disp()
{
  temp = front;
  if ((front == NULL) && (rear == NULL))
  cout << "Queue is empty" << endl;
  while (temp != NULL)
  {
    cout << temp -> data << " ";
    temp = temp -> link;
  }
  cout << endl;
 }
};
```

```
int main()
{
    queue o; o.disp;
    cout << "Queue created:" << endl;
    o.in(10);
    o.in(20);
    o.in(30);
    o.in(40);
    o.in(50);
    o.disp();
    o.out();
    cout << "Queue after one deletion:" << endl;
    o.disp();
}
```

Output:

```
Queue is empty
Queue created:
10 20 30 40 50
Element out from queue is: 10
Queue after one deletion:
20 30 40 50
```

UNIT - 04

Trees and Graphs

* **Data structure:**

A data structure is a collection of data type 'values' which are stored and organised in such a way that it allows for efficient access and modification.

There are generally four forms of data structures:

- Linear Data Structure: arrays, lists, stacks and queues
- Tree Data Structure: Binary, heaps, space partitioning
- Hash Data structure: distributed hash table, hash tree
- Graphs: decision, directed, acyclic

* **Tree:**

The concept of a 'tree' in its simplest terms is to represent a hierarchical data. A tree contains "nodes" and each node is connected by a line called an "edge". These lines represent the relationship between the nodes.

The top level node is known as the "root" and a node with no children is a "leaf". If a node is connected to other nodes, then the preceeding node is referred to as the "parent" and nodes following it are "child" nodes.

Tree is a non-linear data structure

- **Relation of tree**

Root: Top most node
Leaf: Has no child
Children: a node following a node (parent)
Parent: a node predecessor of a node
Siblings: Have same parent node
Edges: If a tree has N nodes, then there are N-1 edges

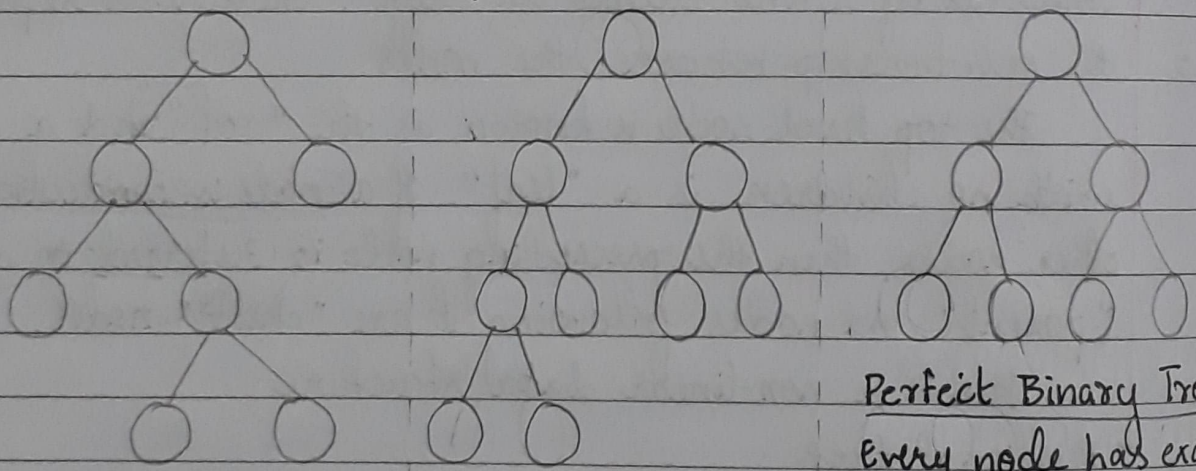
Depth of a node: the length of the path from the node to the root.

Height of a tree: the maximum node depth, i.e., the number of edges in the longest path from the root to a leaf.

- Applications of trees:
  - Storing naturally hierarchial data: File System
  - Organising data for quick search, insertion and deletion: search tree
  - Dictionary Tree
  - Network Routing Algorithm.

★ Binary Tree:

A binary tree is a "rooted tree" and consists of nodes which have at most two children. A node can have only left and right child or only a left child or only a right child. A leaf node has only NULL, ie, no left or right child.



Strict/Proper/2-Binary Tree

Every node should have exactly 2 nodes except the leaves, ie, each node can have 2 or 0 child.

Complete Binary Tree

The last level is completely filled and all the nodes are as left as possible.

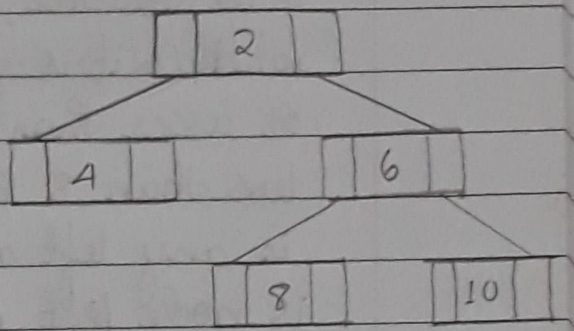
Perfect Binary Tree

Every node has exactly two nodes and all levels are completely filled. It satisfies all the properties of complete and full binary tree

We can implement Binary tree using:

a. Dynamically created nodes: (Linked list representation)

A linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree.



The left pointer has a pointer to the left child of the node and the right pointer has a pointer to the right child of the node. If there are no children for a given node (leaf), then the pointers (left and right) for that node are set to null. The data part contains the actual data of the node.

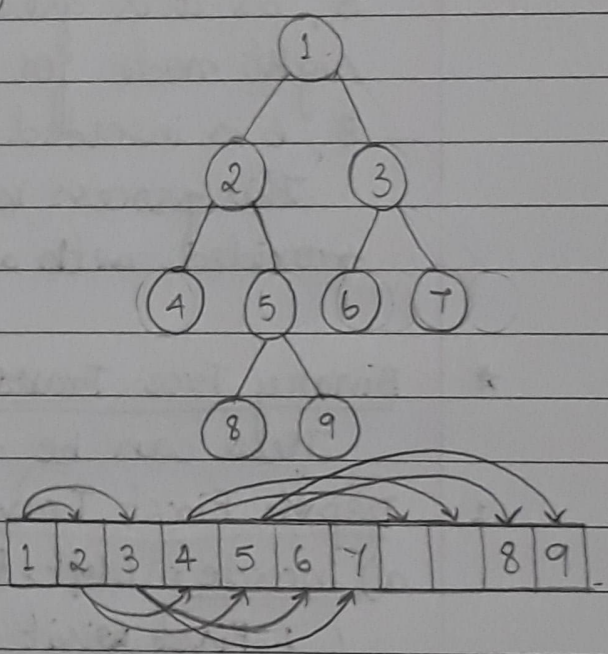
b. Arrays: (Sequential Representation)

An array is used to store a tree data structure. The number of nodes in a tree defines the size of the array. The root node of the tree is stored at the first index of the array.

For a node of index i:

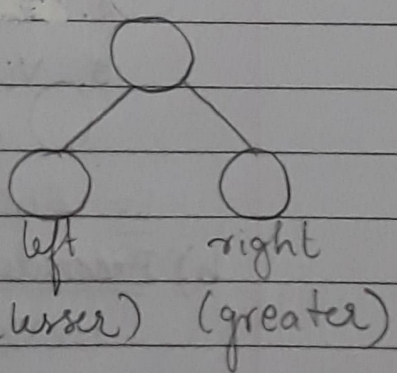
Right child index: $2i+2$

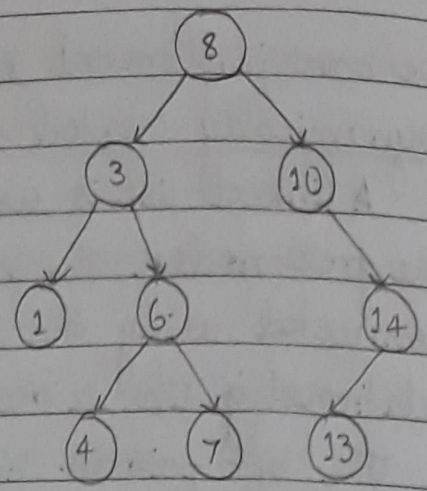
Left child index: $2i+1$



★ Binary Search Tree:

A binary search tree is a "sorted" tree. The left node should always be a lesser number than the right node, and the parent node should be the decider as to whether a child node is placed to the left or the right.





Ex: considering 8 as the root node. If we try inserting 3 to the tree, first it is checked whether it is greater than 8 or lesser than 8. As 3 is less than 8 it checks if there is any left node, as there is no left node here 3 is added as the left node.

Next 6 is added, it is compared with 8. As 6 is less than 8 it is checked if there is any left node. Here 3 is present as the left node. Now 6 is compared with 3, as it is greater than 3, it is checked if there is any right node for 3. Here as there is no right node for 3, 6 is inserted as the right node.

This process is continued until the tree has been provided with all the relevant numbers.

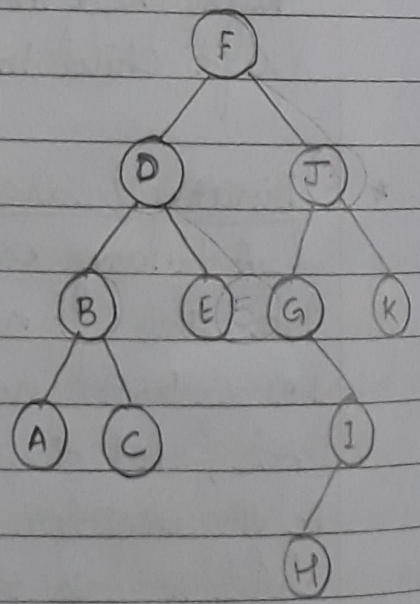
★ **Binary Tree Traversal**

Trees can be traversed in different ways.

1. **Depth First Traversals**

a) **Inorder** (Left, Root, Right)

1. First visit all the nodes in the left subtree
2. Then visit the root node
3. Visit all the nodes in the right subtree



Ex: A B C D E F G H I J K

b) **Preorder** (Root, Left, Right)

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

Ex: FDBACEJGIHK

c) Postorder: (Left, Right, Root)

1. Visit all the nodes in the left subtree.
2. Visit all the nodes in the right subtree.
3. Visit the root node.

Ex: ACBEDHIGKJF

Code:

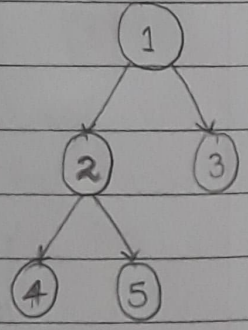
```
#include <iostream>
using namespace std;
class traversal
{
    struct node
    {
        int data;
        node *left, *right;
    };
    public: traversal()
            {
                left = NULL;
                right = NULL;
            }
    void postorder (struct node* n)
    {
        if (n == NULL)
            return;
        postorder (n->left);
        postorder (n->right);
        cout << n-> data << " ";
    }
    void inorder (struct node* n)
    {
        if (n == NULL)
            return;
```

```
    inorder (n->left);
    cout << n->data << " ";
    inorder (n->right);
  }
  void preorder (struct node* n)
  {
    if (n==NULL)
    return;
    cout << n->data << " ";
    preorder (n->left);
    preorder (n->right);
  }
};
int main()
{
    struct node *root = new node(1);
    root->left = new node(2);
    root->right = new node(3);
    root->left->left = new node(4);
    root->left->right = new node(5);
    cout << "Preorder Traversal: \n";
    preorder (root);
    cout << "Inorder Traversal: \n";
    inorder (root);
    cout << "Postorder Traversal: \n";
    postorder (root);
}
```

Preorder Traversal:
1 2 4 5 3

Inorder Traversal:
4 2 5 1 3

Postorder Traversal:
4 5 2 3 1



2. Breadth-First / Level Order Traversals:

A level order traversal is a traversal which always traverses based on the level of the tree. It first traverses the node corresponding to level 0, level 1 and so on, from the root node.

Ex: F D J B E G K A C I H

★ Search an element in Binary Search Tree:

1. Start from root
2. Compare the element with root, if less than root, the recurse for left, else recurse for right.
3. If element to search is found anywhere, return true or else return false.

Code:

```
bool search (node * root, data type)
{
    if (root == NULL)
      return false;
    else if (root -> data == data)
      return true;
    else if (root -> data <= data)
      return search (root -> left, data);
    else
      return search (root -> right, data);
```

★ Delete a node from Binary Search Tree:

When we delete a node, there are three possibilities.

CASE 1. Node to be deleted is a leaf (no child node)

This case is easy to handle, move to the reference of the node and return NULL to the parent's pointer depending upon whether the node is on the left or on the right.

CASE 2. Node to be deleted has only one child:
copy the child to the node and delete the node.

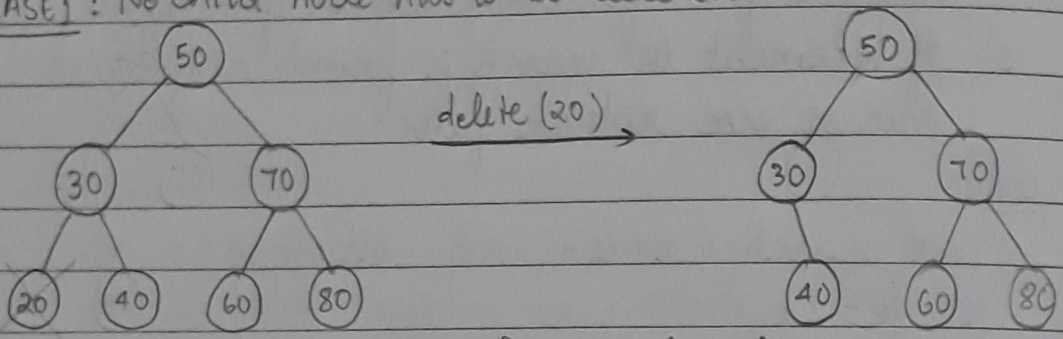
CASE 3. Node to be deleted has two children:
There are two ways to delete:

a. Find minimum in right, copy the value into the node to be deleted. Delete the duplicate from right subtree.

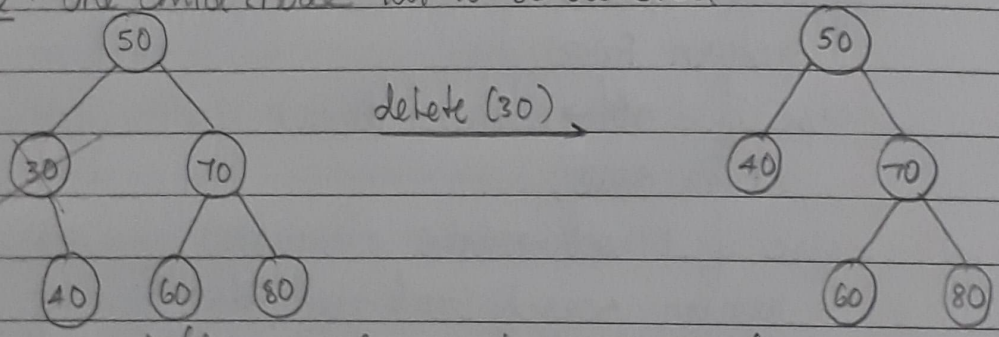
b. Find maximum in left, copy the value into the node to be deleted. Delete the duplicate from left subtree

Ex:

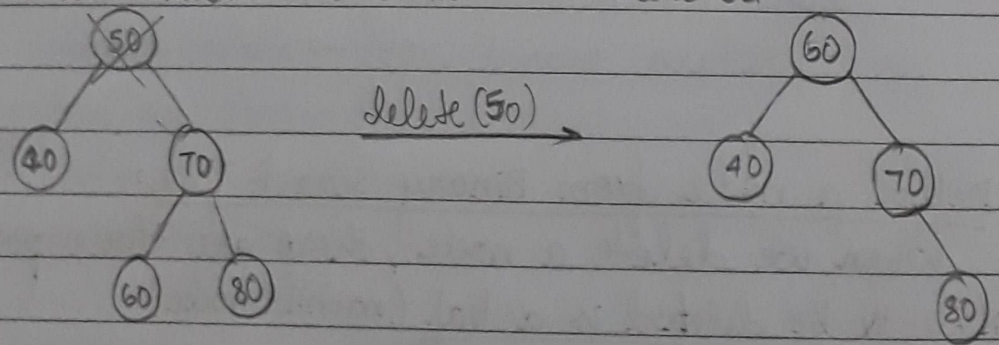
CASE1: No child node has to be deleted



CASE2: One child node has to be deleted



CASE3: Two children node has to be deleted



Page No.:
Date:

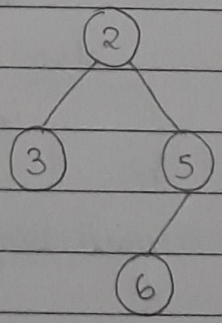
* **Running time of operation:**

Operation	Array	Linked List	Binary search Tree
search	O(log n)	O(n)	O(log n)
Insertion	O(n)	O(1)	O(log n)
Deletion	O(n)	O(n)	O(log n)

(average case)

Q: Write a C++ code to create a tree

```
#include <iostream>
using namespace std;
struct node{
    int data;
    node *left, *right;
};
node* create (int num)
{
    node *q = new node;
    q->data = num;
    q->left = NULL;
    q->right = NULL;
    return q;
}
int main()
{
    node *root = create(2);
    root->left = create(3);
    root->right = create(5);
    root->right->left = create(6);
    cout << root->data;
    cout << root->left->data;
    cout << root->right->data;
    cout << root->right->left->data;
}
```



Output
2356

Q: Write a C++ code to create a binary search tree and hence to search a node.

```
#include <iostream>
using namespace std;
struct node {
    int data;
    node *lt, *rt;
};
node* create (int num)
{
    node *q = new node;
    q->data = num;
    q->lt = q->rt = NULL;
    return q;
}
void insert (node *&root, int num)
{
    if (root == NULL)
        root = create (num);
    else if (num > root->data)
        insert (root->rt, num);
    else
        insert (root->lt, num);
}
void display (node *root)
{
    if (root != NULL)
    {
        display (root->lt);
        cout << root->data;
        display (root->rt);
    }
}
```

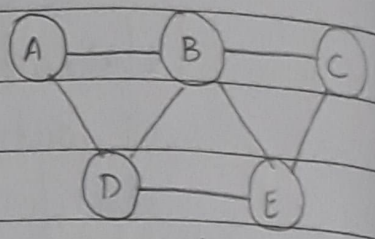
```
bool search (node * & root, int num)
{
    if (root == NULL)
        return false;
    if (root -> data == num)
        return true;
    else if (num <= root -> data)
        return search (root -> lt, num);
    else if (num > root -> data)
        return search (root -> rt, num);
}
int main()
{
    node *root = NULL;
    int n, num, h;
    cout << " Enter the number of nodes " << endl;
    cin >> n;
    for (int i=0; i<n; i++)
    {
        cout << " Enter the number to be inserted " << endl;
        cin >> num;
        insert (root, num);
    }
    display (root);
    cout << " Enter the number to be searched " << endl;
    cin >> h;
    if (search (root, h) == true)
        cout << " Number is found " << endl;
    else
        cout << " Number is not found " << endl;
}
```

* Graphs:

A graph is a non-linear structure. It is a collection of nodes which are also called as "vertices" and "edges" connect two or more vertices.

If two nodes are connected by an edge then they are called adjacent nodes or neighbours.

The number of edges connected to a particulare node is called the degree of the node.



Graph G

vertices: {A, B, C, D, E}

edges: {(A,B), (B,C), (A,D), (B,D), (B,E), (C,E), (D,E)}

The sequence of nodes that we need to follow when we have to travel from one vertex to another in a graph is called the path.

A graph in which the edges are not directed is called an undirected graph. A graph in which the edges have directions associated with them is called a directed graph.

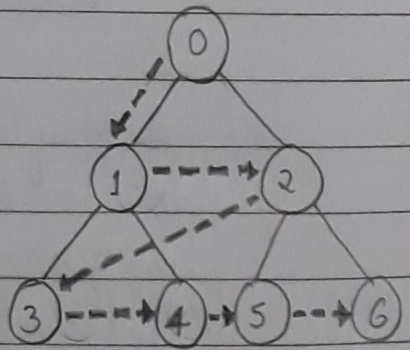
The vertex from which the path initiates is called "Initial Node" and the vertex into which the path terminates is called the "Terminal Node".

Traversal is the technique used to visit every node of the graph of a tree. There are two standard methods of traversals.

1. Breadth First Search : BFS

BFS algorithm is all about searching level by level. Therefore it searches horizontally before moving to the next step.

BFS uses a queue for storing the nodes that it is visiting. Hence it is a vertex based algorithm.



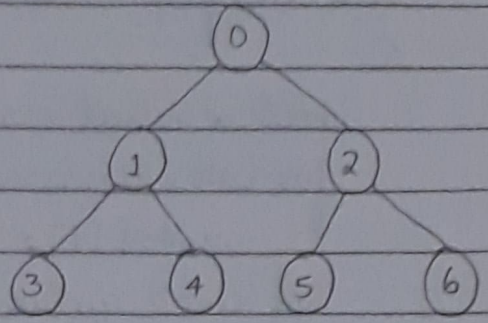
BFS: 0 1 2 3 4 5 6

queue: 1 2 3 4 5 6

2. Depth First Search : DFS

In DFS algorithm, the searching is done from the root to the leaf following one path. Therefore it is a vertical searching.

DFS uses stack while traversing the node. Hence it is a edge based-algorithm

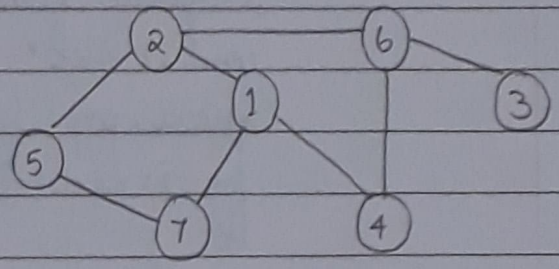


DFS: 0 1 3 4 2 5 6

stack: 1 3 4 2 5 6

Q: Write a c++ code for BFS and DFS of the given graph

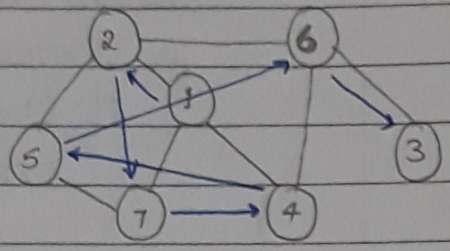
- Breadth First Search




```
# include <iostream>
# include <list>
using namespace std;
class BFSgraph
{
    int vert; // number of vertices
    list <int> *next; // pointer to an array containing
                      adjacency lists
    public:
        BFSgraph (int vert); // constructor
        void add (int vert, int w); // add an edge from vertex v to w
        void BFS (int s); // starting node
};
BFSgraph :: BFSgraph (int vert){
{
    this -> vert = vert;
    next = new list <int> [vert];
}
void BFSgraph :: add (int vert, int w)
{
    next [vert]. push_back (w); // add w to v's list
}
```

```
void BFSgraph :: BFS (int s)
{
    bool * visited = new bool [vert];
    for (int i=0; i<vert ; i++)
        visited [i] = false;
    list <int> queue;  // queue to hold BFS traversal sequence
    visited [s] = true ;  // mark the current node as
    queue.push_back(s);  visited and enqueue it.
    list <int> :: iterator i ;  // to get all adjacent vertices
    while (!queue.empty())
    {
        s = queue.front (); // dequeue the vertex
        cout << s << " ";
        queue.pop_front(); // get all adjacent vertices
        for (i = next [s]. begin(); i != next[s]. end (); ++i)
        {
            if (!visited [*i])
            {
                visited [*i] = true;
                queue.push_back (*i);
            }
        }
    }
}
int main()
{
    BFSgraph v(8);
    v.add (1,2);
    v.add (1,7);
    v.add (1,4);
    v.add (2,5);
    v.add (4,6);
```

```
v.add(6,3);
v.add(5,7);
cout << "Following is Breadth First search \n"
     << "(starting from vertex 1) \n";
v.BFS(1);
return 0;
}
```



BFS: 1 2 7 4 5 6 3

queue: ~~2 7 4 5 6 3~~

Output:

Following is Breadth First search
(starting from vertex 1)
1 2 7 4 5 6 3

- Depth First Search

```
#include <iostream>
#include <list>
using namespace std;
class DFSgraph
{
    int vert;    // number of vertices
    list<int> *next;    // adjacency list
    void DFS(int vert, bool visited[]);
  public:
    DFSgraph(int vert);    // class Constructor
    void add(int vert, int w);    // function to add an edge to graph
    void vis(int vert);
};
DFSgraph::DFSgraph(int vert)
{
    this->vert = vert;
    next = new list<int>[vert];
}
void DFSgraph::add(int vert, int w)
{
    next[vert].push_back(w);    // add w to vert's list
}
```

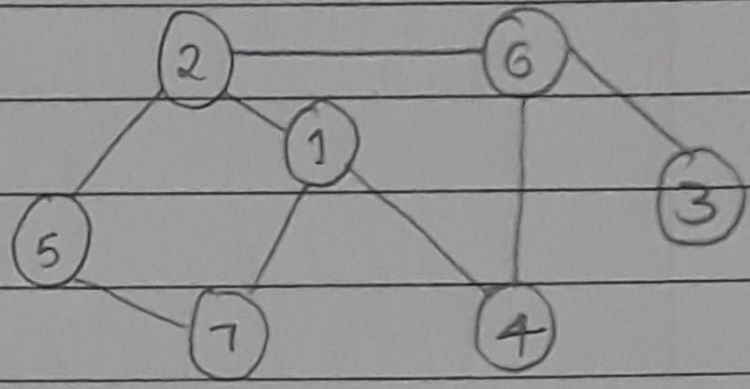
```
void DFSgraph :: DFS(int vert, bool visited[])
{
    visited[vert] = true; // current node vert is visited
    cout << vert << " ";
    list<int> :: iterator i; //procurs all adjacent vertices of the node
    for (i = next[vert].begin(); i != next[vert].end(); ++i)
        if (!visited[*i])
            DFS(*i, visited);
}
void DFSgraph :: vis(int vert) // DFS traversal
{
    bool *visited = new bool[vert];
    for (int i=0; i<vert; i++)
        visited[i] = false;
        DFS(vert, visited);
}
int main()
{
    DFSgraph v(8);
    v.add(1,2);
    v.add(1,7);
    v.add(1,4);
    v.add(2,5);
    v.add(2,6);
    v.add(4,6);
    v.add(6,3);
    v.add(5,7);
    cout << "Following is Depth First search \n"
        << " (starting from vertex 1) \n";
    v.vis(1);
    return 0;
}
```

Page No.:
Date:

Output:

Following is Depth First search
(starting from vertex 1)

1 2 5 7 6 3 4



DFS: 1257634

UNIT - 05

Searching

* Linear Search:

Linear search is used to find whether a given number is present in an array and if it is present then at what location it occurs. Each element in the list is compared with the number until it is found or till the list ends

1. Linear Search Method in an unsorted array:

```
#include <iostream>
using namespace std;
const int MAX = 10;
class array
{
  private: int arr[MAX];
           int count;
  public: array();
          void add(int item);
          int search(int item);
};
array:: array()
{
   count=0;
   for(int i=0; i<MAX; i++)
       arr[i]=0;
}
void array:: add(int item)
{
   if(count<MAX)
   {
      arr[count]=item;
```

```
        count++;
    }
    else
        cout << "In Array is full" << endl;
}
int array :: search (int num)
{
    int i;
    for (i=0; i<count; i++)
    {
        if (arr[i] == num)
            break;
    }
    return i;
}
int main()
{
    array a;
    a.add(3);
    a.add(90);
    a.add(11);
    a.add(2);
    a.add(9);
    a.add(13);
    a.add(54);
    a.add(25);
    a.add(17);
    a.add(1);
    int num;
    cout << "\nEnter the number to be searched";
    cin >> num;
    int i = a.search(num);
```

Find '25'

0	1	2	3	4	5	6	7	8	9
3	90	11	2	9	13	54	25	17	1

```
if (i == MAX)
    cout << "\n Number is not present in the array.";
else
    cout << "\n Number is present in position " << i <<
        "in the array." << endl;
}
```

2. Linear search Method in a sorted array:

```
#include <iostream>
using namespace std;
const int MAX = 10;
class array
{
    private:
        int arr[MAX];
        int count;
    public:
        array();
        void add(int item);
        void search(int item);
};
array :: array()
{
    count = 0;
    for (int i=0; i<MAX; i++)
        arr[i] = 0;
}
void array :: add(int item)
{
    if (count < MAX)
    {
        arr[count] = item;
        count++;
    }
```

```
    else
        cout << "Array is full" << endl;
}
void array :: search (int num)
{
    for (int i=0; i<count; i++)
    {
        if (arr[count-1]<num || arr[i]>=num)
        {
            if (arr[i] == num)
            cout << "Number is present in the position " << i <<
                " in the array." << endl;
            else
                cout << "Number is not present in the array." << endl;
            break;
        }
    }
}
int main()
{
    array a;
    a.add (1);
    a.add (2);
    a.add (3);
    a.add (9);
    a.add (13);
    a.add (15);
    a.add (17);
    a.add (25);
    a.add (54);
    a.add (90);
    int num;
```

Page No.:
Date:
YOUVA

```
cout << "Enter number to be searched: ";
cin >> num;
cout << endl;
a.search(num);
}
```

* **Binary search:**

Binary search method is a method to find the required element in a sorted array by repeatedly halving the array and searching in the half.

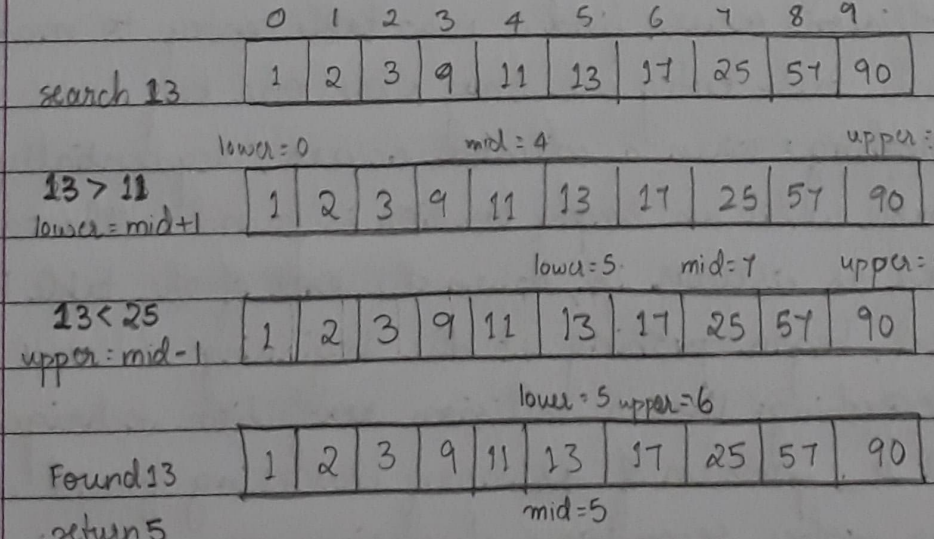
This method is done by starting with the whole array. Then it is halved. If the required data value is greater than the element at the middle of the array, then the upper half of the array is considered. Otherwise, the lower half is considered. This is done continuously until either the required data value is obtained or the remaining array is empty.

- **Binary search**

```
#include <iostream>
using namespace std;
const int MAX=10;
class array
{
  private:
    int arr[MAX];
    int count;
  public:
    array();
    void add(int item);
    void search(int item);
};
```

```
array :: array()
{
    count = 0;
    for (int i=0; i<MAX; i++)
        arr[i] = 0;
}
void array :: add (int item)
{
    if (count < MAX)
    {
        arr [count] = item;
        count++;
    }
    else
        cout << "In Array is full" << endl;
}
void array :: search (int num)
{
    int mid, lower = 0, upper = count - 1, flag = 1;
    for (mid = (lower + upper)/2; low <= upper; mid = (lower + upper)/2)
    {
        if (arr[mid] == num)
        {
            cout << "In the number is at position" << mid <<
                "in the array." << endl;
            flag = 0;
            break;
        }
        if (arr[mid] > num)
            upper = mid - 1;
        else
            lower = mid + 1;
    }
```

```
    if (flag)
        cout<<"\n Element is not present in the array."<<endl;
}
int main()
{
    array a;
    a.add(1);
    a.add(2);
    a.add(3);
    a.add(9);
    a.add(11);
    a.add(13);
    a.add(17);
    a.add(25);
    a.add(57);
    a.add(90);
    int num;
    cout<<"\n Enter the number to be searched: ";
    cin>>num;
    a.search(num);
}
```

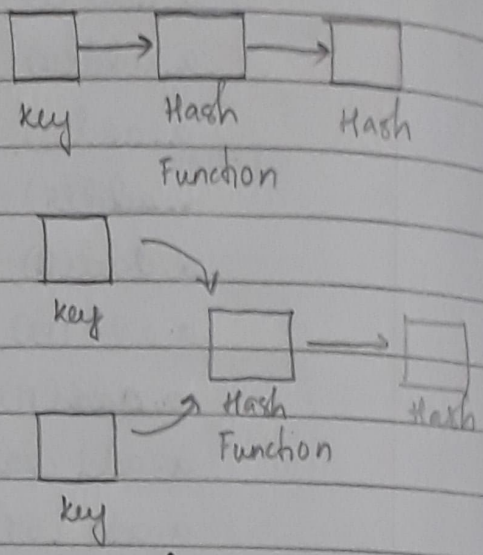


* Hashing:

Hashing is the mechanism of assigning unique code to a variable or attribute using an algorithm to enable easy retrieval.

A hash function is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a hash value or hash.

A good hash function uses a one-way hashing algorithm or in other words, the hash cannot be converted back into the original key.



Two keys can generate the same hash, this phenomenon is known as a collision.

Hashing is most commonly used to implement hash tables. A hash table stores key/value pairs in the form of a list where any element can be accessed using its index.

The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling techniques.

1. Open Addressing:

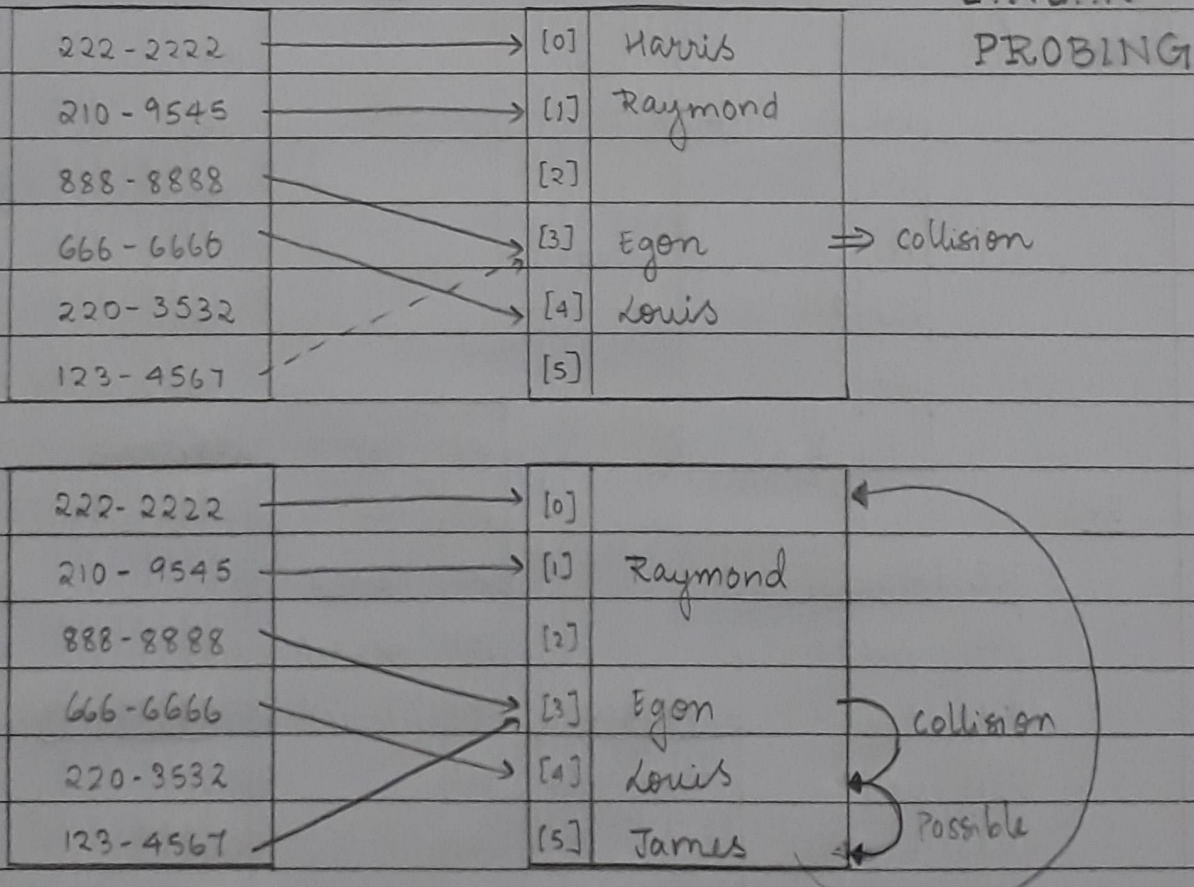
When collisions occur, find a new table entry to make the insertion. Each table entry can only store one key.

- Linear Probing: when a collision occurs, sequentially search the table until an empty location is found. The table is treated as circular i.e., when the end of the table has been probed, begin probing at the beginning
- Quadratic Probing: Uses a collision resolution technique to avoid the problem of primary clustering found with linear probes. Secondary clustering may occur i.e. the initial collision requires a recalculation of a new table entry.

- Double Hashing: when the result of the hash function results in a collision, a second hash function is used. Having a different increment may reduce the overflow problems found with quadratic probing and usually results in less clustering than with linear probing.
- Rehashing: Increasing the size of table. As the hash table gets fuller, the probability of a collision increases. Whenever the table reaches some predetermined percentage utilization the size of the table is increased. The array should not be doubled because the size of the table should be a prime number. Existing elements cannot be simply copied to the new table, all the previous entries are rehashed to new locations based on the new table size.

Ex:

LINEAR PROBING



QUADRATIC PROBING

Ex: $h_i(key) = (h(key) + i^2) \% \text{table size}$

key : 23, 13, 21, 14, 7, 8, 15

table size : 7

$h_0(23) = (23 \% 7) \% 7 = 2$

$h_0(13) = (13 \% 7) \% 7 = 6$

$h_0(21) = (21 \% 7) \% 7 = 0$

$h_0(14) = (14 \% 7) \% 7 = 0$

$h_0(7) = (7 \% 7) \% 7 = 0$

$h_0(8) = (8 \% 7) \% 7 = 1$

$h_0(15) = (15 \% 7) \% 7 = 1$

14 → [0] 21
7 → [1] 14
8 → [2] 23
15 → [3] 15
[4] 7
[5] 8
[6] 13

i=1, i=2, i=2, i=3

Ex:

DOUBLE HASHING

123-4567 → [0] 123-4567
222-2222 → [1]
888-8888 → [2] 222-2222
210-3532 → [3]
[4] 210-3532
[5]
[6]
[7] 888-8888
[8]
[9]

2 steps

8 steps

Ex:

REHASHING

[0]	key = 5
[1]	key = 1
[2]	key = 2
[3]	key = 3
[4]	

Increase table size →

[0]	key = 5	key is in the wrong location
[1]	key = 1	
[2]	key = 2	
[3]	key = 3	
[4]		rehashing →
[5]		
[6]		
[7]		
[8]		
[9]		
[10]		

[0]	
[1]	key = 1
[2]	key = 2
[3]	key = 3
[4]	
[5]	key = 5
[6]	
[7]	
[8]	
[9]	
[10]	

Keerthana Ashok

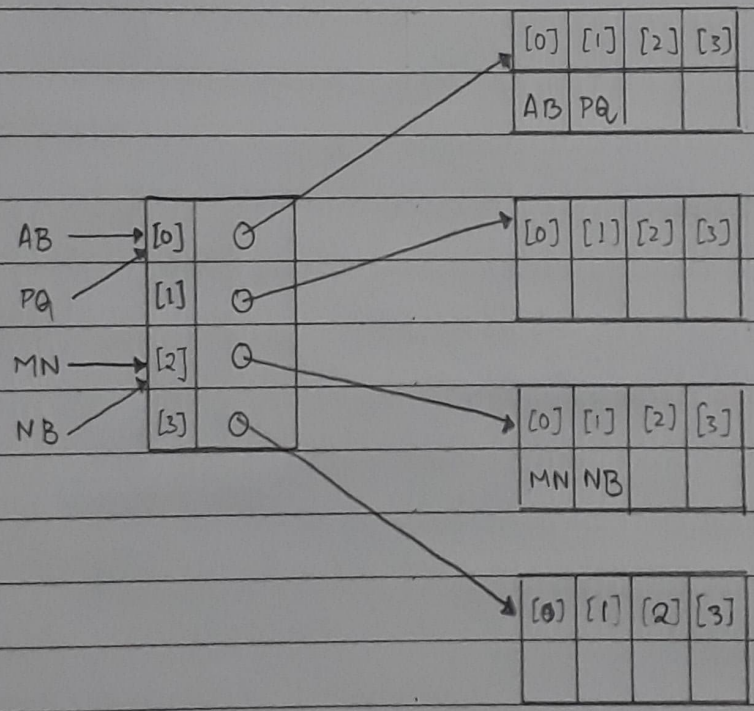
2. Closed Addressing:

When a collision occurs, accomodate the additional key by adding additional keys at the same location in the table. Each table entry can only store multiple keys.

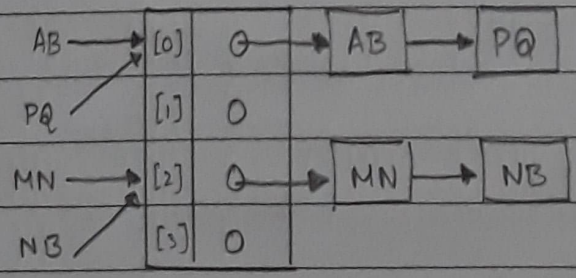
- Implementing each table entry as a Bucket:
  Each hash table entry is a 1D array. If the size of the bucket is too small, the problem with collisions has to be postponed, whereas if it is too large then memory is wasted. Hence rarely used method.
- Separate (External) Chaining:
  Each table entry is a reference to a linked list.

Ex:

Implementing each table entry as a Bucket



Ex: Separate (External) Chaining



* Open Addressing (closed Hashing)
  - Resolves collisions by finding another place in the hash table
  - May be faster in practice because the table does not change in size.
  - Requires a larger hash table

Closed Addressing (Open Hashing)
  - Resolves collisions by inserting additional elements at the same location in the hash table.
  - May require less memory
  - May be slower in practice because of the dynamic memory allocations. More complex as it needs another data structure.

UNIT - 06

Sorting

* **Insertion sort:**

In insertion sort we sort the array by virtually splitting the array into a sorted and an unsorted part where values from the unsorted part are picked and placed at the correct position in the sorted part.

- **code:**

```
#include <iostream>
using namespace std;
const int MAX = 10;
class array
{
    private:
        int arr[MAX];
        int count;
    public:
        array();
        void add(int item);
        void sort();
        void display();
};
array :: array()
{
    count = 0;
    for(int i=0; i<MAX; i++)
        arr[i] = 0;
}
```

```
void array :: add (int item)
{
   if (count < MAX)
   {
      arr[count] = item;
      count++;
   }
   else
      cout << "Array is full." << endl;
}
void array :: sort()
{
   int temp, j;
   for (int i=1; i<count ; i++)
   {
      temp = arr[i];
      j = i;
      while (j>0 && arr[j-1] > temp)
      {
         arr[j] = arr[j-1];
         j--;
      }
      arr[j] = temp;
   }
}
void array :: display()
{
   for (int i=0; i< count; i++)
      cout << arr[i] << "\t";
   cout << endl;
}
```

Page No.
Date: YOUVA

```
int main()
{
    array a;
    a.add(25);
    a.add(17);
    a.add(10);
    a.add(31);
    a.add(13);
    a.add(6);
    a.add(23);
    a.add(18);
    a.add(2);
    a.add(12);
    cout << " Insertion sort.\n" <<endl;
    cout << " Array before sorting : " << endl;
    a.display();
    cout << " Array after sorting : " <<endl;
    a.sort();
    a.display();
}
```

25	17	10	31	13	6	23	18	2	12
17	25	10	31	13	6	23	18	2	12
10	17	25	31	13	6	23	18	2	12
10	17	25	31	13	6	23	18	2	12
10	13	17	25	31	6	23	18	2	12
6	10	13	17	25	31	23	18	2	12
6	10	13	17	23	25	31	18	2	12
6	10	13	17	18	23	25	31	2	12
2	6	10	13	17	18	23	25	31	12
2	6	10	12	13	17	18	23	25	31

* **Selection Sort**

The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part of the array and putting it at the beginning.

- **Code:**

```
#include <iostream>
using namespace std;
const int MAX = 10;
class array
{
    private:
```

```
    int arr[MAX];
    int count;
  public:
    array();
    void add(int item);
    void sort();
    void display();
};
array::array()
{
    count = 0;
    for(int i=0; i<MAX; i++)
        arr[i] = 0;
}
void array::add(int item)
{
    if(count<MAX)
    {
        arr[count] = item;
        count++;
    }
    else
        cout<<"Array is full"<<endl;
}
void array::sort()
{
    int temp;
    for(int i=0; i<=count-2; i++)
    {
        for(int j=i+1; j<=count-1; j++)
        {
```

```
            if(arr[i] > arr[j])
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
void array :: display()
{
    for (int i=0; i < count ; i++)
        cout << arr[i] << "\t";
    cout << endl;
}
int main()
{
    array a;
    a.add (25);
    a.add (17);
    a.add (10);
    a.add (13);
    a.add (6);
    a.add (23)
    a.add (31);
    a.add (18);
    a.add (2);
    a.add (12);
    cout<<" Selection sort. \n" <<endl;
    cout<<" Array before sorting: " <<endl;
    a. display();
    cout <<" Array after sorting: " <<endl;
```

1st pass	25	17	10	13	6	23	31	18	2	12
	17	25	10	13	6	23	31	18	2	12
	10	25	17	13	6	23	31	18	2	12
	6	25	17	13	10	23	31	18	2	12
	2	25	17	13	10	23	31	18	6	12
2nd pass	2	17	25	13	10	23	31	18	6	12
	2	13	25	17	10	23	31	18	6	12
	2	10	25	17	13	23	31	18	6	12
	2	6	25	17	13	23	31	18	10	12
3rd pass	2	6	17	25	13	23	31	18	10	12
	2	6	13	25	17	23	31	18	10	12
	2	6	10	25	17	23	31	18	13	12
4th pass	2	6	10	17	25	23	31	18	13	12
	2	6	10	13	25	23	31	18	17	12
	2	6	10	12	25	23	31	18	17	13
5th pass	2	6	10	12	23	25	31	18	17	13
	2	6	10	12	18	25	31	23	17	13
	2	6	10	12	17	25	31	23	18	13
	2	6	10	12	13	25	31	23	18	17
6th pass	2	6	10	12	13	23	31	25	18	17
	2	6	10	12	13	18	31	25	23	17
	2	6	10	12	13	17	31	25	23	18
7th pass	2	6	10	12	13	17	25	31	23	18
	2	6	10	12	13	17	23	31	25	18
	2	6	10	12	13	17	18	31	25	23
8th pass	2	6	10	12	13	17	18	25	31	23
	2	6	10	12	13	17	18	23	31	25
9th pass	2	6	10	12	13	17	18	23	25	31

```
        a.sort();
        a.display();
    }
```

* **Bubble Sort:**

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.