EC2202 - Data Structures and Object Oriented Programming in C++

LECTURE NOTES

Text Books:
#1. Mark Allen Weiss, "Data Structures and Algorithm Analysis in C", 3rd ed, Pearson Education Asia, 2007.
#2. E. Balagurusamy, “Object Oriented Programming with C++", McGraw Hill Company Ltd., 2007.

Anna University | B.E Electronics and Communication Engineering | Semester III

UNIT 1

PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

Introduction-Tokens-Expressions-control Structures-Functions in C++,classes and objects,constructors and destructors,operators overloading and type conversion

Introduction

Basic concepts of OOPS

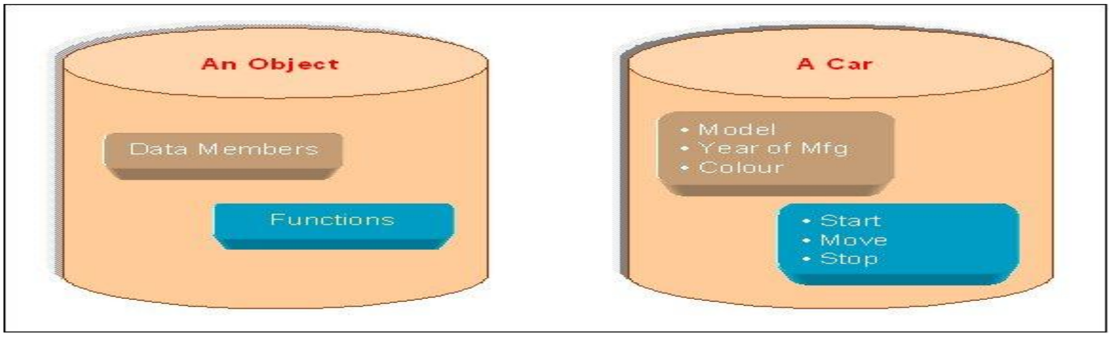
Before starting to learn C++ it is essential that one must have a basic knowledge of the concepts of Object oriented programming. Some of the important object oriented features are namely:

- Objects
- Classes
- Inheritance
- Data Abstraction
- Data Encapsulation
- Polymorphism
- Overloading
- Reusability

In order to understand the basic concepts in C++, the programmer must have a command of the basic terminology in object-oriented programming. Below is a brief outline of the concepts of Object-oriented programming languages:

Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.



After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

> On the average quicksort has O(n log n) complexity, but strong proof of this fact is not trivial and not presented here. Still, you can find the proof in [1]. In worst case, quicksort runs O(n^2) time, but on the most "practical" data it works just fine and outperforms other O(n log n) sorting algorithms

```
void quickSort(int arr[], int left, int right) {

    int i = left, j = right;

    int tmp;

    int pivot = arr[(left + right) / 2];

     /* partition */

    while (i <= j) {

        while (arr[i] < pivot)

            i++;

        while (arr[j] > pivot)

            j--;

        if (i <= j) {

            tmp = arr[i];

            arr[i] = arr[j];

            arr[j] = tmp;

            i++;

            j--;

        }

    };

     /* recursion */
```

Overloading:

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

Introduction to C++

Variable, Constants and Data types in C++

Variables

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore, for example: character _

Rules for defining variable name:

- A variable name can have one or more letters or digits or underscore for example character _.
.
- White space, punctuation symbols or other characters are not permitted to denote variable name. .
- A variable name must begin with a letter.
.
- Variable names cannot be keywords or any reserved words of the C++ programming language.
.
- C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters. For example, the variable name EXFORSYS differs from the variable name exforsys.

Data Types

Below is a list of the most commonly used *Data Types* in C++ programming language:

short int : This data type is used to represent short integer.

int: This data type is used to represent integer.

long int: This data type is used to represent long integer.

float: This data type is used to represent floating point number.

double: This data type is used to represent double precision floating point number.
long double: This data type is used to represent double precision floating point number.
char: This data type is used to represent a single character.

bool: This data type is used to represent boolean value. It can take one of two values: True or False.

Using variable names and data type, we shall now learn how to declare variables.

Declaring Variables:

In order for a variable to be used in C++ programming language, the variable must first be declared. The syntax for declaring variable names is

```
data type variable name;
```

The date type can be int or float or any of the data types listed above. A variable name is given based on the rules for defining variable name (refer above rules).

Example:

```
int a;
```

This declares a variable name a of type int.

If there exists more than one variable of the same type, such variables can be represented by separating variable names using comma.

For instance

```
int x,y,z
```

This declares 3 variables x, y and z all of data type int.

The data type using integers (int, short int, long int) are further assigned a value of **signed** or **unsigned**. Signed integers signify positive and negative number value. Unsigned integers signify only positive numbers or zero.

For example it is declared as

```
unsigned short int a;
signed int z;
```

By default, unspecified integers signify a signed integer.

For example:

```
int a;
```

is declared a signed integer

It is possible to initialize values to variables:

```
data type variable name = value;
```

Example:

```
int a=0;
int b=5;
```

Constants

Constants have fixed value. Constants, like variables, contain data type. Integer constants are represented as decimal notation, octal notation, and hexadecimal notation. Decimal notation is represented with a number. Octal notation is represented with the number preceded by a zero character. A hexadecimal number is preceded with the characters 0x.

Example

80 represent decimal
0115 represent octal
0x167 represent hexadecimal

By default, the integer constant is represented with a number.

The unsigned integer constant is represented with an appended character **u**. The long integer constant is represented with character **l**.

Example:

78 represent int
85u present unsigned int
78l represent long

Floating point constants are numbers with decimal point and/or exponent.
Example

2.1567
4.02e24

These examples are valid floating point constants.

Floating point constants can be represented with f for floating and l for double precision floating point numbers.

Character constants have single character presented between single quotes.

Example

‘c’
‘a’

are all character constants.

Strings are sequences of characters signifying string constants. These sequence of characters are represented between double quotes.

Example:

“Exforsys Training”

is an example of string constant.

Referencing variables

The & operator is used to reference an object. When using this operator on an object, you are provided with a pointer to that object. This new pointer can be used as a parameter or be assigned to a variable.

Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. A stream of these tokens makes up a translation unit.Tokens are usually separated by "white space." White space can be one or more:

- Blanks
- Horizontal or vertical tabs
- New lines
- Formfeeds
- Comments

Expression

An expression is a combination of variables constants and operators written according to the syntax of C language. In C++ every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable.

Control Structures

It is of three types:

1. Sequence structure
2. Selection structure
3. Loop or iteration or repetition structure

Just like C, C++ supports all the control structures of C. The control structures if, and switch are selection structures. The control structures do..while, while, and for are called loop structure.

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

if (condition) statement

where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

The format of while loop is:

while (expression) statement

and its functionality is simply to repeat statement while the condition set in expression is true.

The format of for loop is:

for (initialization; condition; increase) statement;

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.

2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).

3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.

4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
case constant1:
group of statements 1;
break;
case constant2:
group of statements 2;
break;
.
.
.
default:
default group of statements
}
```

It works in the following way:

switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Functions in C++

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

type name (parameter1, parameter2, ...) { statements }

where:

• type is the data type specifier of the data returned by the function.

• name is the identifier by which it will be possible to call the function.

• parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.

• statements is the function's body. It is a block of statements surrounded by braces { }.

Classes And Objects

An Overview about Objects and Classes

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an *object*. A *class* is an extended concept similar to that of *structure* in C programming language, this class describes the data properties alone. In C++ programming language, *class* describes both the properties (data) and behaviors (functions) of objects. *Classes* are not *objects*, but they are used to instantiate *objects*.

Features of Class:

Classes contain details known as member data and member functions. As a unit, the collection of member data and member functions is an object. Therefore, this unit of objects make up a class.

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol, {is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol} and concluded with a colon;.

```
class exforsys
{
  member data;
  member functions;
  ……………
};
```

There are different access specifiers for defining the data and functions present inside a class.

Access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

- private
- public
- protected

- A *private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.
.
- *Public* members are accessible from outside the class.
.
- A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

When defining access specifiers, the programmer must use the keywords: *private*, *public* or *protected* when needed, followed by a semicolon and then define the data and member functions under it.

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
};
```

In the code above, the member *x* and *y* are defined as private access specifiers. The member function sum is defined as a public access specifier.

General Syntax of a class:

General structure for defining a class is:

```
class classname
{
  acess specifier:
  data member;
  member functions;

  acess specifier:
  data member;
  member functions;
};
```

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. Private is the default access level for specifiers. If no access specifiers are identified for members of a class, the members are defaulted to private access.

```
class exforsys
{
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
};
```

In this example, for members x and y of the class *exforsys* there are no access specifiers identified. *exforsys* would have the default access specifier as private.

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Juts as we declare a variable of data type *int* as:

```
int x;
```

Objects are also declared as:

class name followed by object name;

```
exforsys e1;
```

This declares *e1* to be an object of class exforsys.

For example a complete class and object declaration is given below:

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
};

main()
{
  exforsys e1;
  ……………
  ……………
}
```

The object can also be declared immediately after the class definition. In other words the object name can also be placed immediately before the closing flower brace symbol } of the class declaration.

For example

```
class exforsys
{
  private:
  int x,y;
  public:
  void sum()
  {
    ………
    ………
  }
}e1 ;
```

The above code also declares an object *e1* of class exforsys.

It is important to understand that in object-oriented programming language, when a class is created no memory is allocated. It is only when an object is created is memory then allocated.

Function Overloading

A function is overloaded when same name is given to different function. However, the two functions with the same name will differ at least in one of the following.

a) The number of parameters
b) The data type of parameters
c) The order of appearance

These three together are referred to as the **function signature**.

For example if we have two functions :

void foo(int i,char a);
void boo(int j,char b);

Their signature is the same **(int ,char)** but a function

void moo(int i,int j) ; has a signature **(int, int)** which is different.

While **overloading a function**, the return type of the functions need to be the same.

In general functions are overloaded when :
1. Functions differ in function signature.
2. Return type of the functions is the same.

Here s a basic example of **function overloading**

```
#include <iostream>
using namespace std;

class arith {
public:
   void calc(int num1)

{
cout<<"Square of a given number: " <<num1*num1 <<endl;
}

   void calc(int num1, int num2 )

{
cout<<"Product of two whole numbers: " <<num1*num2 <<endl;
}
};
```

```
int main() //begin of main function
{
   arith a;
   a.calc(5);
   a.calc(6,7);
}
```

First the overloaded function in this example is **calc**. If you have noticed we have in our **arith** class two functions with the name **calc**. The fist one takes one integer number as a parameter and prints the square of the number. The second calc function takes two integer numbers as parameters, multiplies the numbers and prints the product. This is all we need for making a successful overloading of a function.

a) we have two functions with the same name : *calc*
b) we have different signatures : *(int)* , *(int, int)*
c) return type is the same : void

The result of the execution looks like this

```
Square of a given number: 25
Product of two whole numbers: 42
```

The result demonstrates the overloading concept. Based on the arguments we use when we call the **calc** function in our code :

```
a.calc(5);
a.calc(6,7);
```

The compiler decides witch function to use at the moment we call the function.

C++ Friend Functions

Need for Friend Function:

As discussed in the earlier sections on access specifiers, when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-memberfunctions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

How to define and use Friend Function in C++:

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in C++:

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.

  .
- It is possible to declare a function as friend in any number of classes.

  .
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.

  .
- A friend function, even though it is not a member function, would have the rights to access the private members of the class.

  .
- It is possible to declare the friend function as either private or public.

  .
- The function can be invoked without the use of an object. The friend function has its argument as objects, seen in example below.

Example to understand the friend function:

```
#include
class exforsys
{
private:
int a,b;
public:
void test()
{
a=100;
b=200;
}
friend int compute(exforsys e1)

//Friend Function Declaration with keyword friend and with the object of class exforsys
to which it is friend passed to it
};
```

```
int compute(exforsys e1)
{
//Friend Function Definition which has access to private data
return int(e1.a+e2.b)-5;
}

main()
{
exforsys e;
e.test();
cout<<"The result is:"<
//Calling of Friend Function with object as argument.
}
```

The output of the above program is

```
The result is:295
```

The function compute() is a non-member function of the class exforsys. In order to make this function have access to the private data a and b of class exforsys , it is created as afriend function for the class exforsys. As a first step, the function compute() is declared as friend in the class exforsys as:

```
friend int compute (exforsys e1)
```

The keyword friend is placed before the function. The function definition is written as a normal function and thus, the function has access to the private data a and b of the class exforsys. It is declared as friend inside the class, the private data values a and b are added, 5 is subtracted from the result, giving 295 as the result. This is returned by the function and thus the output is displayed as shown above.

Constant and volatile member functions

A member function declared with the **const** qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the **volatile** qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be called for a nonvolatile object.

static members

Class members can be declared using the storage class specifier static in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects.

You access a static member by qualifying the class name using the :: (scope resolution) operator. In the following example, you can refer to the static member f() of class type X as X::f() even if no object of type X is ever declared:

```
struct X {
  static int f();
};
int main() {
  X::f();
}
```

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
  CRectangle * prect;
```

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example

#include <iostream>

using namespace std;

class CRectangle {

    int width, height;

  public:

    void set_values (int, int);

    int area (void) {return (width * height);}
```

```
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}
int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() << endl;
  cout << "d[1] area: " << d[1].area() << endl;
  delete[] d;
  delete b;
  return 0;
```

```
}
```

Output:

```
a area: 2

*b area: 12

*c area: 2

d[0] area: 30

d[1] area: 56
```

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

expression	**can be read as**
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Difference between const variables and const object

Constant variables are the variables whose value cannot be changed through out the programme but if any object is constant, value of any of the data members (const or non const) of that object cannot be changed through out the programme. Constant object can invoke only constant function.

Nested classes

A nested class is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

```
class A {
  int x;
  class B { };
  class C {
    // The compiler cannot allow the following
    // declaration because A::B is private:
    //   B b;
    int y;
    void f(A* p, int i) {
    // The compiler cannot allow the following
    // statement because A::x is private:
    //   p->x = i;
    }
  };
  void g(C* p) {
    // The compiler cannot allow the following
    // statement because C::y is private:
    //   int z = p->y;
  }
};
int main() { }
```

The compiler would not allow the declaration of object b because class A::B is private. The compiler would not allow the statement p->x = i because A::x is private. The compiler would not allow the statement int z = p->y because C::y is private.

Local classes

A local class is declared within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;                        // global variable
void f()                        // function definition
{
      static int y;            // static variable y can be used by
                            // local class
      int x;                  // auto variable x cannot be used by
                            // local class
      extern int g();          // extern function g can be used by
                            // local class
      class local             // local class
      {
            int g() { return x; }       // error, local variable x
                                    // cannot be used by g
            int h() { return y; }       // valid,static variable y
            int k() { return ::x; }     // valid, global x
            int l() { return g(); }     // valid, extern function g
      };
}
int main()
{
      local* z;                // error: the class local is not visible
      // ...}
```

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword inline.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()
{
    class local
    {
       int f();             // error, local class has noninline
                          // member function
       int g() {return 0;}   // valid, inline member function
       static int a;         // error, static is not allowed for
                          // local class
       int b;               // valid, nonstatic variable
    };
}
//     . . .
```

An enclosing function has no special access to members of the local class.

Constructors

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

<class name> { arguments};

The default constructor for a class X has the form

X::X()

In the above example the arguments is optional.

The constructor is automatically invoked when an object is created.

The various types of constructors are

1. Default constructors
2. Parameterized constructors
3. Copy constructors

Default Constructor:

This constructor has no arguments in it. Default Constructor is also called as *no argument constructor*.

For example:

```
class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys(); //default Constructor
      ...
};

Exforsys :: Exforsys()
{
   a=0;
   b=0;
}
```

Parameterized Constructor:

A parameterized constructor is just one that has parameters specified in it.

Example:

```
class Exforsys
{
   private:
      int a,b;
   public:
      Exforsys(int,int);// Parameterized constructor
      ...
};

Exforsys :: Exforsys(int x, int y)
{
   a=x;
   b=y;
}
```

Copy constructor:

One of the more important forms of an overloaded constructor is the copy constructor. The purpose of the copy constructor is to initialize a new object with data copied from another object of the same class.

For example to invoke a copy constructor the programmer writes:

```
Exforsys e3(e2);
or
Exforsys e3=e2;
```

Both the above formats can be used to invoke a copy constructor.

For Example:

```
#include <iostream.h>
class Exforsys()
{
   private:
      int a;
   public:
      Exforsys()
      { }
      Exforsys(int w)
   {
      a=w;
   }
   Exforsys(Exforsys& e)
   {
      a=e.a;
      cout<<" Example of Copy Constructor";
   }
   void result()
   {
      cout<< a;
   }
};

void main()
{
   Exforsys e1(50);
   Exforsys e3(e1);
   cout<< "\ne3=";e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is

```
Example of Copy Constructor
e3=50
```

Some important points about constructors:

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.
- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible.

Destructors

What is the use of Destructors

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically called when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilda ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

For example:

```
class Exforsys
{
   private:
      ……………
   public:
      Exforsys()
      { }
      ~ Exforsys()
      { } }
```

Operator Overloading

Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

Operators can be broadly classified into:

- Unary Operators
- Binary Operators

Unary Operators:

As the name implies takes operate on only one operand. Some unary operators are namely

++	- Increment operator
--	- Decrement Operator
!	- Not operator
-	- unary minus.

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword **operator**.

The general syntax for defining an operator overloading is as follows:

```
return_type classname :: operator operator symbol(argument)
{
…………..
statements;
}
```

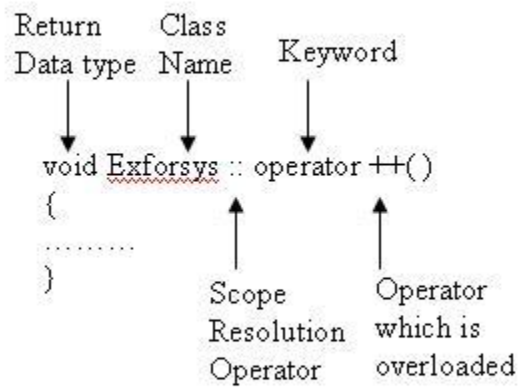
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

- return_type – is the data type returned by the function
- class name - is the name of the class
- operator – is the keyword
- operator symbol – is the symbol of the operator which is being overloaded or defined for new functionality
- **::** - is the scope resolution operator which is used to use the function definition outside the class.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys
{
   private:
   ……….
   public:
   void operator ++( );
   …………
};
```

The important steps involved in defining an operator overloading in case of unary operators are namely:

- ➢ Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
- ➢ If the function is a member function then the number of arguments taken by the operator member function is none.

- If the function defined for the operator overloading is a friend function then it takes one argument.

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
   private:
   int x;
   public:
   Exforsys( ) { x=0; }      //Constructor
   void display();
   void Exforsys ++( );          //overload unary ++
};

void Exforsys :: display()
{
   cout<<”\nValue of x is: “ << x;
}

void Exforsys :: operator ++( ) //Operator Overloading for operator ++
                                          defined
{
   ++x;
}

void main( )
{
   Exforsys e1,e2;        //Object e1 and e2 created
   cout<<”Before Increment”
   cout <<”\nObject e1: ”<<e1.display();
   cout <<”\nObject e2: ”<<e2.display();
   ++e1;  //Operator overloading applied
   ++e2;
   cout<<”\n After Increment”
   cout <<”\nObject e1: ”<<e1.display();
   cout <<”\nObject e2: ”<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
```

```
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1
```

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator Overloading – Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
class Exforsys
{
private:
int x;
int y;

public:
Exforsys()                  //Constructor
{ x=0; y=0; }

void getvalue( )            //Member Function for Inputting Values
{
cout << “\n Enter value for x: “;
cin >> x;
cout << “\n Enter value for y: “;
cin>> y;
}

void displayvalue( )          //Member Function for Outputting Values
{
cout <<”value of x is: “ << x <<”; value of y is: “<<y
}

Exforsys operator +(Exforsys);
};

Exforsys Exforsys :: operator + (Exforsys e2)
//Binary operator overloading for + operator defined
{
int x1 = x+ e2.x;
int y1 = y+ e2.y;
return Exforsys(x1,y1);
}

void main( )
{
Exforsys e1,e2,e3;            //Objects e1, e2, e3 created
cout<<\n”Enter value for Object e1:”;
e1.getvalue( );
cout<<\n”Enter value for Object e2:”;
e2.getvalue( );
e3= e1+ e2;                   //Binary Overloaded operator used
cout<< “\nValue of e1 is:”<<e1.displayvalue();
cout<< “\nValue of e2 is:”<<e2.displayvalue();
cout<< “\nValue of e3 is:”<<e3.displayvalue();
}
```

The output of the above program is:

```
Enter value for Object e1:
Enter value for x: 10
Enter value for y: 20
Enter value for Object e2:
Enter value for x: 30
Enter value for y: 40
Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60
```

In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

The important aspect is the statement:

```
e3= e1 + e2;
```

The binary overloaded operator '+' is used. In this statement, the argument on the left side of the operator '+', e1, is the object of the class Exforsys in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' . Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

Some operators cannot be overloaded:
Scope resolution operator denoted by ::
Member access operator or the dot operator denoted by .
Conditional operator denoted by ?:
Pointer to member operator denoted by .*

Operator Overloading through friend functions

```
// Using friend functions to
// overload addition and subtarction
// operators
#include <iostream.h>
class myclass
{
  int a;
  int b;
public:
  myclass(){}
  myclass(int x,int y){a=x;b=y;}
  void show()
  {
    cout<<a<<endl<<b<<endl;
  }
  // these are friend operator functions
  // NOTE: Both the operans will be be
  // passed explicitely.
  // operand to the left of the operator
  // will be passed as the first argument
  // and operand to the right as the second
  // argument
  friend myclass operator+(myclass,myclass);
  friend myclass operator-(myclass,myclass);
};
myclass operator+(myclass ob1,myclass ob2)
{
  myclass temp;
  temp.a = ob1.a + ob2.a;
  temp.b = ob1.b + ob2.b;
  return temp;
}
myclass operator-(myclass ob1,myclass ob2)
{
  myclass temp;
  temp.a = ob1.a - ob2.a;
  temp.b = ob1.b - ob2.b;
  return temp;
}
void main()
{
  myclass a(10,20);
  myclass b(100,200);
  a=a+b;
  a.show();
}
```

Overloading the Assignment Operator (=)

We know that if we want objects of a class to be operated by common operators then we need to *overload them*. But there is one operator whose operation is automatically crested by C++ for every class we define, it is the assignment operator '='.

Actually we have been using similar statements like the one below previously

```
ob1=ob2;
```

where ob1 and ob2 are objects of a class.

The simple program below illustrates how it can be done. Here we are defining two similar classes, one with the default assignment operator (created automatically) and the other with the overloaded one. Notice how we could control the way assignments are done in that case.

```
// Program to illustrate the
// overloading of assignment
// operator '='
#include <iostream.h>
// class not overloading the
// assignment operator
class myclass
{
  int a;
  int b;
public:
  myclass(int, int);
  void show();
};
myclass::myclass(int x,int y)
{
  a=x;
  b=y;
}
void myclass::show()
{
  cout<<a<<endl<<b<<endl;
}
// class having overloaded
// assignment operator
class myclass2
{
  int a;
  int b;
public:
  myclass2(int, int);
  void show();
  myclass2 operator=(myclass2);
};
myclass2 myclass2::operator=(myclass2 ob)
{
  // -- do something specific --
  // this is just to illustrate
  // that when overloading '='
  // we can define our own way
  // of assignment
  b=ob.b;
  return *this;
 };
myclass2::myclass2(int x,int y)
{
  a=x;
  b=y;
}
void myclass2::show()
{
  cout<<a<<endl<<b<<endl;
}
// main
void main()
{
  myclass ob(10,11);
  myclass ob2(20,21);
  myclass2 ob3(100,110);
  myclass2 ob4(200,210);

  // does a member-by-member copy
  // '=' operator is not overloaded
  ob=ob2;
  ob.show();
  // does specific assignment as
  // defined in the overloaded
  // operator definition
  ob3=ob4;
  ob3.show();
}
```

Type Conversions

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

There are two ways of achieving the type conversion namely:

Automatic Conversion otherwise called as **Implicit Conversion**
Type casting otherwise called as **Explicit Conversion**

Automatic Conversion otherwise called as Implicit Conversion

This is not done by any conversions or operators. In other words value gets automatically converted to the specific type in which it is assigned.

```
#include <iostream.h>
void main()
{
short x=6000;
int y;
y=x;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

Type casting otherwise called as Explicit Conversion

Explicit conversion can be done using type cast operator and the general syntax for doing this is

```
datatype (expression);
```

Here in the above datatype is the type which the programmer wants the expression to gets changed as

In C++ the type casting can be done in either of the two ways mentioned below namely:

- C-style casting
- C++-style casting

The C-style casting takes the syntax as

```
(type) expression
```

The C++-style casting takes the syntax as

```
type (expression)
```

Let us see the concept of type casting in C++ with a small example:

```
#include <iostream.h>
void main()
{
int a;
float b,c;
cout<< “Enter the value of a:”;
cin>>a;
cout<< “n Enter the value of b:”;
cin>>b;
c = float(a)+b;
cout<<”n The value of c is:”<<c;
}
```

The output of the above program is

```
Enter the value of a: 10
Enter the value of b: 12.5
The value of c is: 22.5
```

In the above program ‘a’ is declared as integer and b and c are declared as float. In the type conversion statement namely

```
c = float(a)+b;
```

The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable c with value as 22.5

Explicit Constructors

The keyword explicit is a Function Specifier." The explicit specifier applies only to constructors. Any time a constructor requires only one argument either of the following can be used to initialize the object. The reason for this is that whenever a constructor is created that takes one argument, it also implicitly creates a conversion from the type of that argument to the type of the class. A constructor specified as explicit will be used only when an initialization uses the normal constructor syntax, Data (x). No automatic conversion will take place and Data = x will not be allowed. Thus, an explicit constructor creates a "nonconverting constructor."

Example:

```
class Data
{

explicit Data(float x); // Explicit constructor
{ }

};
```

Implicit Constructors

If a constructor is not stated as explicit, then it is by default an implicit constructor.

UNIT II

ADVANCED OBJECT ORIENTED PROGRAMMING

Inheritance,Extending,classes,pointers,Virtual functions and polymorphism,File Handling templates,Exception handling,Manipulating strings

Inheritance

Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

For example, a programmer can create a *base* class named fruit and define *derived* classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

This concept of *Inheritance* leads to the concept of *polymorphism*.

Features or Advantages of Inheritance:

- ***Reusability:***

  Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

- ***Saves Time and Effort:***

  The above concept of reusability achieved by inheritance saves the programmer time and effort, because the main code written can be reused in various situations as needed.

- ***Increases Program Structure which results in greater reliability.***
- ***Polymorphism***

General Format for implementing the concept of Inheritance:

class derived_classname: access specifier baseclassname

For example, if the *base* class is *exforsys* and the derived class is sample it is specified as:

```
class sample: public exforsys
```

The above makes sample have access to both *public* and *protected* variables of base class *exforsys*. Reminder about public, private and protected access specifiers:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
  .
- Public members and variables are accessible from outside the class.
  .
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

Inheritance Example:

```
class exforsys
{
private:
int x;



public:
exforsys(void) { x=0; }
void f(int n1)
{
x= n1*5;
}

void output(void) { cout<<x; }
};

class sample: public exforsys
{
public:
sample(void) { s1=0; }

void f1(int n1)
{
s1=n1*10;
}

void output(void)
{
```

```
exforsys::output();
cout << s1;
}

private:
int s1;
};

int main(void)
{
sample s;
s.f(10);
s.output();
s.f1(20);
s.output();
}
```

The output of the above program is

```
50
200
```

In the above example, the derived class is sample and the base class is *exforsys*. The *derived* class defined above has access to all *public* and *private* variables. *Derived* classes cannot have access to base class *constructors* and *destructors*. The derived class would be able to add new member functions, or variables, or new constructors or new destructors. In the above example, the derived class sample has new member function f1( ) added in it. The line:

sample s;

creates a derived class object named as s. When this is created, space is allocated for the data members inherited from the base class *exforsys* and space is additionally allocated for the data members defined in the derived class *sample*.

The *base* class constructor *exforsys* is used to initialize the base class data members and the *derived* class *constructor* sample is used to initialize the data members defined in *derived* class.

The access specifier specified in the line:

```
class sample: public exforsys
```

Public indicates that the *public* data members which are inherited from the *base* class by the derived class sample remains *public* in the *derived* class.

A derived class inherits every member of a base class except:

- its constructor and its destructor
- its friends
- its operator=() members
- class sample: public exforsys

Types of Inheritance

There are five different inheritances supported in C++:

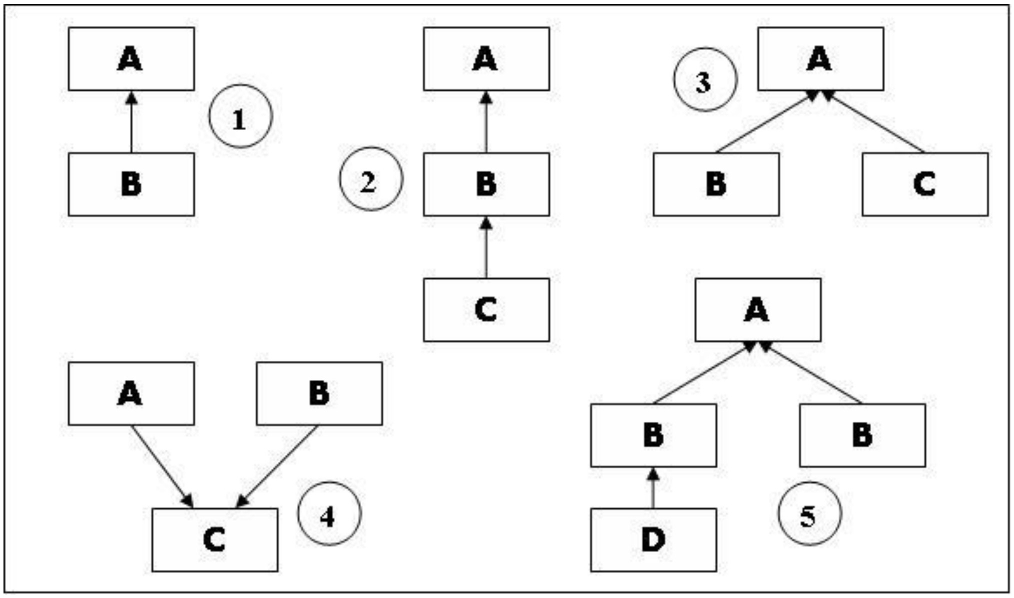
(1) Simple / Single

(2) Multilevel

(3) Hierarchical

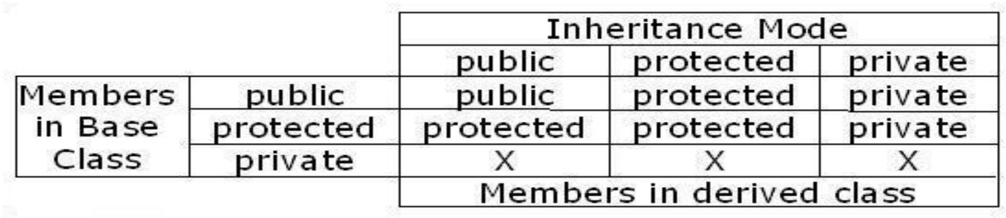
(4) Multiple

(5) Hybrid



Accessibility modes and Inheritance

We can use the following chart for seeing the accessibility of the members in the Base class (first class) and derived class (second class).



		Inheritance Mode		
		public	protected	private
Members in Base Class	public	public	protected	private
	protected	protected	protected	private
	private	X	X	X
		Members in derived class		

Here X indicates that the members are not inherited, i.e. they are not accessible in the derived class.

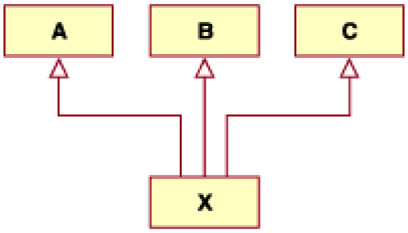
Multiple inheritance

We can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

In the following example, classes A, B, and C are direct base classes for the derived class X:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

The following inheritance graph describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:

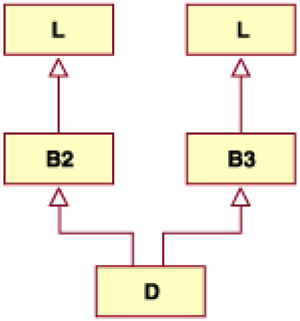


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ };                    // direct base class

class D : public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:



```
class L { /* ... */ };                  // indirect base class
```

class B2 : public L { /* ... */ };

class B3 : public L { /* ... */ };

class D : public B2, public B3 { /* ... */ }; // valid

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

B3::L.

we can also avoid this ambiguity by using the base specifier virtual to declare a base class.

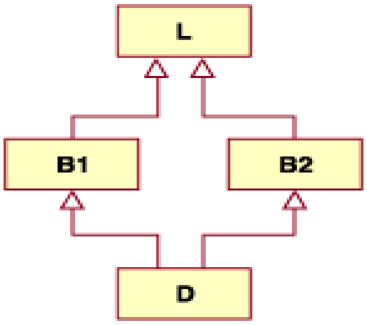
Extending Classes

Virtual base classes

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as virtual to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifiers in the base lists of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

For example:



class L { /* ... */ }; // indirect base class

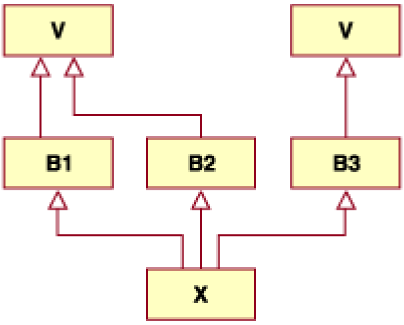
class B1 : virtual public L { /* ... */ };

class B2 : virtual public L { /* ... */ };

class D : public B1, public B2 { /* ... */ }; // valid

Using the keyword virtual in this example ensures that an object of class D inherits only one subobject of class L.

A derived class can have both virtual and nonvirtual base classes. For example:



class V { /* ... */ };

class B1 : virtual public V { /* ... */ };

class B2 : virtual public V { /* ... */ };

class B3 : public V { /* ... */ };

class X : public B1, public B2, public B3 { /* ... */

};

In the above example, class X has two subobjects of class V, one that is shared by classes B1 and B2 and one through class B3.

Abstract classes

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You can declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

class AB {

```
public:

  virtual void f() = 0;

};
```

Function AB::f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
class A {

  virtual void g() { } = 0;

};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
class A {
  virtual void f() = 0;
};
class A {
  virtual void f() { }
};
// Error:
// Class A is an abstract class
// A g();
// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);
int main() {
// Error:
// Class A is an abstract class
//   A a;
   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

Class A is an abstract class. The compiler would not allow the function declarations A g() or void h(A), declaration of object a, nor the static cast of b to type A.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
  virtual void f() = 0;
};
class D2 : public AB {
  void g();
};
int main() {
  D2 d;
}
```

The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function f()from AB. The compiler will allow the declaration of object d if you define function D2::g().

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
class A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

The default constructor of A calls the pure virtual function direct() both directly and indirectly (through indirect()).

The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

Pointers

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

The address that locates a variable within memory is what we call a *reference* to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```

This would assign to ted the address of variable andy, since when preceding the name of the variable andy with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

Virtual Functions And Polymorphism

Polymorphism

Polymorphism is the phenomenon where the same message sent to two different objects produces two different set of actions. Polymorphism is broadly divided into two parts:

- *Static polymorphism* – exhibited by overloaded functions.
- *Dynamic polymorphism* – exhibited by using late binding.

Static Polymorphism

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called *early binding*. The term *early binding* stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
 void add(int , int);

void add(float, float);
```

When the *add()* function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term *late binding* refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Static Vs Dynamic Polymorphism

- Static polymorphism is considered more efficient, and dynamic polymorphism more flexible.
- Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.
- As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.
- To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility.

Introduction to Virtual Functions

Polymorphism, one of the three main attributes of an OOP language, denotes a process by which different implementations of a function can be accessed by the use of a single name. Polymorphism also means "one interface, multiple methods."

C++ supports polymorphism both at run-time and at compile-time. The use of overloaded functions is an example of compile-time polymorphism. Run-time polymorphism can be achieved by the use of both derived classes and *virtual functions.*

Pointers to Derived Types

We know that pointer of one type may not point to an object of another type. You'll now learn about the one exception to this general rule: a pointer to an object of a base class can also point to any object derived from that base class.

Similarly, a reference to a base class can also reference any object derived from the original base class. In other words, a base class reference parameter can receive an object of types derived from the base class, as well as objects within the base class itself.

Virtual Functions

How does C++ handle these multiple versions of a function? Based on the parameters being passed, the program determines at run-time which version of the virtual function should be the recipient of the reference. It is the type of object being pointed to, not the type of pointer, that determines which version of the virtual function will be executed!

To make a function *virtual,* the virtual keyword must precede the function declaration in the base class. The redefinition of the function in any derived class does not require a second use of the virtual keyword. Have a look at the following sample program to see how this works:

```
#include<iostream.h>
using namespace std;
class bclass {
public:
   virtual void whichone() {
      cout << "bclass\n";
   }
};
class dclass1 : public bclass {
public:
   void whichone() {
      cout << "dclass1\n";
   }
};
class dclass2 : public bclass {
public:
   void whichone() {
      cout << "dclass2\n";
   }
};
int main()
{
   bclass Obclass;
   bclass *p;
   dclass1 Odclass1;
   dclass2 Odclass2;
   // point to bclass
   p = &Obclass;
   // access bclass's whichone()
   p->whichone();
   // point to dclass1
   p = &Odclass1;
   // access dclass1's whichone()
   p->whichone();
   // point to dclass2
```

```
    p = &Odclass2;
    // access dclass2's whichone()
    p->whichone();
    return 0;
}
```

The output from this program looks like this:

```
bclass

dclass1

dclass2
```

Notice how the type of the object being pointed to, not the type of the pointer itself, determines which version of the virtual whichone() function is executed.

Virtual Functions and Inheritance

Virtual functions are inherited intact by all subsequently derived classes, even if the function is not redefined within the derived class. So, if a pointer to an object of a derived class type calls a specific function, the version found in its base class will be invoked. Look at the modification of the above program. Notice that the program does not define the whichone() function in d class2.

```
#include
using namespace std;
class bclass {
public:
    virtual void whichone() {
        cout << "bclass\n";
    }
};
class dclass1 : public bclass {
public:
    void whichone() {
        cout << "dclass1\n";
    }
};
class dclass2 : public bclass {
};
int main()
{
    bclass Obclass;
    bclass *p;
    dclass1 Odclass1;
    dclass2 Odclass2;
```

```
   p = &Obclass;
   p->whichone();
   p = &Odclass1;
   p->whichone();
   p = &Odclass2;
   // accesses dclass1's function
   p->whichone();
   return 0;
}
```

The output from this program looks like this:

```
bclass

dclass1

bclass
```

Templates

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

template <class identifier> function_declaration;

template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>

myType GetMax (myType a, myType b) {

 return (a>b?a:b);

}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.
To use this function template we use the following format for the function call:
function_name <type> (parameters);
For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;

GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.
Here is the entire example:

```
// function template

#include <iostream>

using namespace std;

template <class T>

T GetMax (T a, T b) {

  T result;

  result = (a>b)? a : b;

  return (result);

}

int main () {

  int i=5, j=6, k;

  long l=10, m=5, n;

  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);

  cout << k << endl;

  cout << n << endl;

  return 0;

}
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.
In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.
As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.
In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;

GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II

#include <iostream>

using namespace std;

template <class T>
```

```
T GetMax (T a, T b) {

  return (a>b?a:b);

}

int main () {

  int i=5, j=6, k;

  long l=10, m=5, n;

  k=GetMax(i,j);

  n=GetMax(l,m);

  cout << k << endl;

  cout << n << endl;

  return 0;

}
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;

long l;

k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
```

```
T GetMin (T a, U b) {

  return (a<b?a:b);

}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;

long l;

i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>

class mypair {

    T values [2];

  public:

    mypair (T first, T second)

    {

      values[0]=first; values[1]=second;

    }

};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;
template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)              100
      {a=first; b=second;}
    T getmax ();
};
template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;

  return retval;

}

int main () {

  mypair <int> myobject (100, 75);

  cout << myobject.getmax();

  return 0;

}
```

Notice the syntax of the definition of member function getmax:

```
template <class T>

T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.
For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization

#include <iostream>

using namespace std;

// class template:
```

```
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};
// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;

  cout << mychar.uppercase() << endl;

  return 0;

}
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an emptytemplate<> parameter list. This is to explicitly declare it as a template specialization.
But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };

template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.
When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template

#include <iostream>

using namespace std;
```

```
template <class T, int N>

class mysequence {

   T memblock [N];

 public:

   void setmember (int x, T value);

   T getmember (int x);

};

template <class T, int N>

void mysequence<T,N>::setmember (int x, T value) {

 memblock[x]=value;

}

template <class T, int N>

T mysequence<T,N>::getmember (int x) {

 return memblock[x];}

int main () {

 mysequence <int,5> myints;

 mysequence <double,5> myfloats;

 myints.setmember (0,100);

 myfloats.setmember (3,3.1416);

 cout << myints.getmember(0) << '\n';

 cout << myfloats.getmember(3) << '\n';

 return 0;

}
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

template <*class* T=*char*, *int* N=10> *class* mysequence {..};

We could create objects using the default template parameters by declaring:

mysequence<> myseq;

Which would be equivalent to:

mysequence<*char*,10> myseq;

Exception Handling

Ways of handling errors in C++

- Errors can be dealt with at place error occurs
  - easy to see if proper error checking implemented
  - harder to read application itself and see how code works
- Exception handling
  - makes clear, robust, fault-tolerant programs
  - C++ removes error handling code from "main line" of program
- Common failures
  - **new** not allocating memory
  - out of bounds array subscript
  - division by zero
  - invalid function parameters

- Exception handling - catch errors before they occur
  - deals with synchronous errors (i.e., divide by zero)
  - does not deal with asynchronous errors - disk I/O completions, mouse clicks - use interrupt processing
  - used when system can recover from error
    - exception handler - recovery procedure
  - typically used when error dealt with in different place than where it occurred
  - useful when program cannot recover but must shut down cleanly
- Exception handling should not be used for program control
  - not optimized, can harm program performance

Exception handling

- Exception handling improves fault-tolerance
  - easier to write error-processing code
  - specify what type of exceptions are to be caught

- Most programs support only single threads
  - techniques in this chapter apply for multithreaded OS as well (Windows NT, OS/2, some UNIX)

- Exception handling another way to return control from a function or block of code

When Exception Handling Should Be Used

- Error handling should be used for
  - processing exceptional situations
  - processing exceptions for components that cannot handle them directly
  - processing exceptions for widely used components (libraries, classes, functions) that should not process their own exceptions
  - large projects that require uniform error processing

Other Error-Handling Techniques

- Use **assert**
  - if assertion **false**, the program terminates
- Ignore exceptions
  - use this "technique" on casual, personal programs - not commercial!
- Abort the program
  - appropriate for nonfatal errors give appearance that program functioned correctly
  - inappropriate for mission-critical programs, can cause resource leaks
- Set some error indicator
  - program may not check indicator at all points the error could occur

Other Error-Handling Techniques can’t

- Test for the error condition
  - issue an error message and call **exit**
  - pass error code to environment

- **setjump** and **longjump**
  - in **<csetjmp>**
  - jump out of deeply nested function calls back to an error handler.
  - dangerous - unwinds the stack without calling destructors for automatic objects (more later)

- specific errors
  - some have dedicated capabilities for handling them
  - if **new** fails to allocate memory **new_handler** function executes to deal with problem

Exception Handling: try, throw, catch

- A function can **throw** an exception object if it detects an error
  - object typically a character string (error message) or class object
  - if exception handler exists, exception caught and handled
  - otherwise, program terminates
- Format
  - enclose code that may have an error in **try** block
  - follow with one or more **catch** blocks
    - each **catch** block has an exception handler
  - if exception occurs and matches parameter in **catch** block, code in catch block executed
  - if no exception thrown, exception handlers skipped and control resumes after catch blocks
  - **throw** point - place where exception occurred
    - control cannot return to **throw** point

Manipulating strings

C++ strings allow to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cin >> name;
  // read string until the next separator
  // (space, tab, newline)

getline (cin, name);
  // read a whole line into the string name

if (name == "")
{
   cout << "You entered an empty string, "
       << "assigning default\n";
   name = "John";
}
```

C++ strings also provide many string manipulation facilities. The simplest string manipulation that is commonly used is concatenation, or addition of strings. In C++, we can use the + operator to concatenate (or "add") two strings, as shown in the example below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
   // result now contains "hello world"
```

Notice that neither **s1** nor **s2** are modified! The operation reads the values and produces a result corresponding to the concatenated strings, but doesn't modify the two original strings.

The += operator can also be used. In that case, one string is appended to another one, as shown in the following example:

```
string result;
string s1 = "hello";
  // without the extra space at the end
string s2 = "world";
result = s1;
result += ' ';
  // append a space at the end
result += s2;
```

Now, result contains the string **"hello world"**.

The following example shows how to create a string that contains the full name from first name and last name (e.g., firsname = "John", lastname = "Smith", fullname = "Smith, John"):

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string firstname, lastname, fullname;

   cout << "First name: ";
   getline (cin, firstname);
   cout << "Last name: ";
   getline (cin, lastname);

   fullname = lastname + ", " + firstname;
   cout << "Fullname: " << fullname << endl;
   return 0;
}
```

UNIT – III

DATA STRUCTURES & ALGORITHMS

Algorithm, Analysis, Lists, Stacks and queues, Priority queues-Binary Heap-Application, Heaps–hashing-hash tables without linked lists

Algorithm Analysis

An **algorithm** is a step- by- step procedure for solving a problem in a finite amount of time.

Big-Oh Notation

Given functions $\boldsymbol{f}(\boldsymbol{n})$ and $\boldsymbol{g}(\boldsymbol{n})$, we say that $\boldsymbol{f}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{g}(\boldsymbol{n}))$ if there are positive constants $\boldsymbol{c}$ and $\boldsymbol{n0}$ such that

$$\boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{cg}(\boldsymbol{n}) \text{ for } \boldsymbol{n} \geq \boldsymbol{n0}$$

Example: $2\boldsymbol{n} + 10$ is $\boldsymbol{O}(\boldsymbol{n})$

- $2\boldsymbol{n} + 10 \leq \boldsymbol{cn}$
- $(\boldsymbol{c} - 2)\, \boldsymbol{n} \geq 10$
- $\boldsymbol{n} \geq 10/(\boldsymbol{c} - 2)$
- Pick $\boldsymbol{c} = 3$ and $\boldsymbol{n0} = 10$

Abstract Data Types (ADT)

Abstract Data Types (ADT) is a mathematical model together with the possible set of operations.

E.g. . List ADT, Tree ADT, Stack ADT, Set ADT and so on.

For example in list ADT we have the operations such as insert, delete, and find, count and so on.

The Set ADT the operations are Union, Intersection, size, complement and so on.

The basic idea is that the implementation of these operations is written once, further needs of this ADT we can call the appropriate function.

List ADT

A list is a sequence of Zero or more elements of a given type. It can be of the form A_1, A_2, A_3, …, A_n , Where the size of the list in ‘n’. If the size is 0 then the list is called as empty list.

For any list except the empty list the element A_{i+1} follows the element A_i (i<n) and that A_{i-1} precedes A_i (i>1)

The First element of the list is A_1 and the last element of the list is A_n. A_1 has no predecessor and A_n has no successor.

We will not define the predecessor of A_1 or the successor of A_n. The position of the element A_i in a list is i.

Operations

PrintList – Print the elements of the list in the order of occurrence

Find – Returns the position of the first occurrence of the element in the list.

Insert – Insert an element into the list at the specified position.

Delete – Delete an element from the list

MakeEmpty – Empty the list.

Array-based Implementation

In this implementation, the list elements are stored in contiguous cells of an array. All the list operation can be implemented by using the array.

Insertion

Insertion refers to the operation of adding another element to the list at the specified position.

If an element is inserted at the end of the array, if there is a space to add the element then insertion can be done easily. If we want to insert an element in the middle of the array then half of the elements must be moved downwards to new location to accommodate the new element and retain the order of the element. If an element is inserted at the beginning of the array then the entire array elements can be moved downward one step to make space for new element.

Routine to insert an element at the specified position

```
void insert(int *a , int pos, int num)

{

        int i;

        for(i=max-1;i>=pos;i--)

                a[i]=a[i-1];

        a[i]=num;
}
```

The running time for insertion operation as O(n).

Deletion

Deletion refers to the operation of removing an element from the array.

Deleting the element from the end of the array can be done easily. Deleting the first element of the array requires shifting all elements in the list up one. Deleting the other elements requires half of the list needs to be moved.

Routine for Delete an element from the specified position

```
void del(int *a ,int pos)

{

        int i;

        for(i=pos;i<max;i++)

                a[i-1]=a[i];

        a[i]=0;

}
```

Delete the first occurrence of the specified element

```
void  del(int *a,int d)

{

        int i,pos;

        pos=-1;

        for(i=0;i<max;i++)

        {

                if(a[i] = =d)

                {
```

```
                pos=i;
                break;
            }
        }
        if(pos==-1)
            printf(“\n Element not  found in the list”);
        else
        {
            for(i=pos;i<max;i++)
                a[i-1]=a[i];
            a[i-1]=0;
        }
}
```

Search

This operation is used to check whether the given element is present in the list or not

```
/*Search a given element */
void search(int *a,int d);
{
        int i,pos;
        pos =-1;
        for(i=0;i<max;i++)
        {
            if(a[i]==d)
            {
                pos =i;
                break;
            }
        }
        if(pos==-1)
            printf(“\n Element not found in the list”);
        else
            printf(“\n Element found at position %d”,pos);
}
```

Display

Display all elements in the array.

```
/* display the list elements */
void display(int *a)
{
    int i;
    printf(“\n List………”);
    for(i=0;i<max;i++)
        printf(“%d\t”,a[i]);
}
```

Disadvantages of Array Implementation’

Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high overestimate, which waste considerable space.

Insertion and deletion operations are expensive, because insertion at the beginning of the array requires pushing the entire array elements one step downwards. As like the deleting the first element of the array requires, shifting all elements up one position. So the worst case operation requires the computation time O(n).

Linked List Implementation

Definition

Linked list consist of a series of structures, which are not necessarily in adjacent in memory. The structure is called as Node.

Each structure (node) contains

- Element
- Pointer to a structure containing its successor. – It is called as Next Pointer

The Last cell's Next pointer point to NULL.A single node is represented as follows.

Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circular Linked List.

Single Linked List

A singly linked list is a list in which each node contains only one link field pointing to the next node in the list.

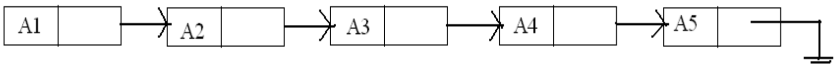
A node in this type of linked list contains two types of fields.

Data – This holds the list element

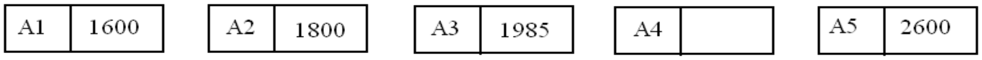
Next – Pointer to the next node in the list

Data	Next Ptr

E.g



The actual representation of the above list is shown below.



The list contains five structures. The structures are stored at memory locations 800, 1600, 1800, 1985, 2600 respectively. The Next pointer in the first structure contains the value 1600.

Basic linked List Operations

The basic operations to be performed on linked lists are as

- Creation - Create a linked list
- Insertion - insert a new node at the specified position
- Deletion - delete the specified node
- Traversing - to display every node information
- Find - Search a particular data

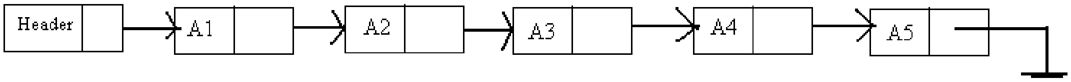
Implementation

For easy implementation of all linked list operation a sentinel node is maintained to point the beginning of the list. This node is sometimes referred to as a header or dummy node. To access the list, we must know the address of the header Node.

To insert a new node at the beginning of the list, we have to change the pointer of the head node. If we miss to do this we can lose the list. Likewise deleting a node from the front of the list is also a special case, because it changes the head of the list.

To solve the above mentioned problem, we will keep a sentinel node.

A linked list with header representation is shown below.



Type Declaration

typedef struct Node *PtrToNode;

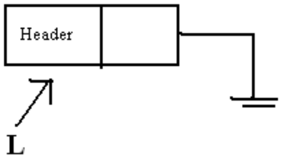
typedef PtrToNode Position;

typedef PtrToNode List;

```
struct Node

{

        ElementType Element;

        Position Next;

}
```

Creating Linked List

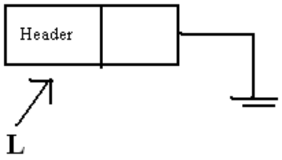
The malloc( ) function is used to allocate a block of memory to a node in a linked list. The create function is used to create a dummy header node. It is shown in the below figure.



```
List Create( )

{

        List L;

                L=(struct Node *)malloc(sizeof(struct Node));

        L->Element=0;

        L->Next=NULL;

        return L;

}
```

Routine to check whether the list is empty

The empty list is shown below.



This function return true if the list is empty otherwise return fals.

```
int IsEmpty (List L) /*Returns 1 if L is empty */
{
        return L->Next = = NULL;
}
```

Routine to check whether the current position is last

This function returns true if P is the last position in List L.

```
int IsLast (Position P, List L) /* Returns 1 is P is the last position in L */
{
        return  P->Next = = NULL;
}
```

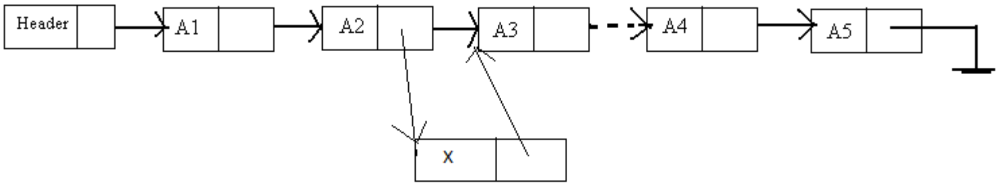
Find Routine

This function returns the position of the element X in the List L.

```
Position Find (int X, List L)
{
        /*Returns the position of X in L; NULL if X is not found */
        Position P;
        P = L ->next;
        while (P! = NULL && P->Element ! = X)
                P = P->Next;
        return P;
}
```

Routine to Insert an Element in an Linked List

This function is used to insert an element X into the list after the position X. The insert action can be shown below



```
void Insert (ElementType  X, List L, Position P)

/* Insert after the position P*/

{

        Position Tmpcell;

        Tmpcell = (struct Node*)malloc (size of (Struct Node));

        If (Tmpcell == NULL)

                Printf("Error! No Space in memory");

        else

        {

                Tmpcell->Element = X;

                Tmpcell->Next = P->Next;

                P->Next = Tmpcell;

        }

}
```

Find Previous Routine

The FindPrevious routine returns the position of the predecessor node.

```
Position FindPrevious (int x, List L)

{
        /* Returns the position of the predecessor */
        Position P;
        P = L;
        while (P->Next ! = NULL && P->Next ->Element ! = x)
                P = P->next;
        return P;
}
```

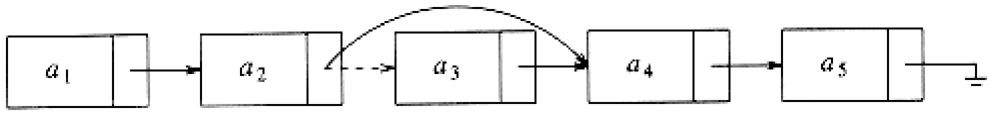
Find Next Routine

This function returns the position of the successor node

```
Position FindNext (int x, List L)
{
        /*Returns the position of its successor */
        P = L->next;
        while (P->Next! = NULL && P->Element ! = x)
                P = P->next;
        return P->next;
}
```

Routine to Delete an Element from the List

This function deletes the first occurrence of an element from the linked list.This can be shown in below figure.



```
void Delete(int x, List L)
```

```
{
        /* Delete the first occurence of x from the List */
        position P, temp;
        P = Findprevious (x,L);
        If (!IsLast(P,L))
        {
                temp = P->Next;
                P ->Next = temp->Next;
                Free (temp);
        }
}
```

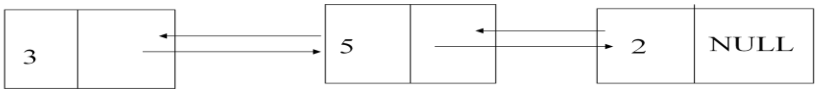
Routine to Delete the list

This function is used to release the memory allocated for the linked list.

```
void DeleteList (List L)
{
        nextptr  P, Temp;
        P = L ->Next;
        L->Next = NULL;
        while (P! = NULL)
        {
                temp = P->next
                free (P);
                P = temp;
} }
```

Doubly Linked List

A node contains pointers to previous and next element**.** One can move in both directions.



Advantages:

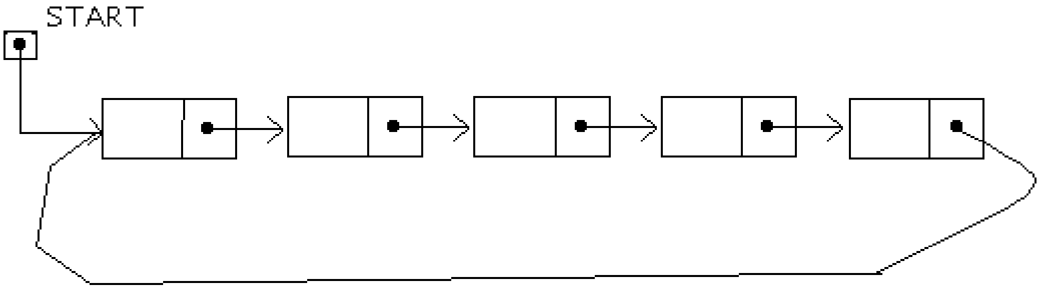
1. It is more efficient.

Disadvantages of Doubly Linked list over Single Linked list

1. The location of the preceding node is needed. The two-way list contains this information, whereas with a one-way list we must traverse the list.
2. A two-way list is not much more useful than a one-way list except in special circumstances.

Circular List

The last node points to the first one.



Cursor based Implementation of List

If linked lists are required and pointers are not available, then an alternate implementation must be used. The alternate method we will describe is known as a *cursor* implementation.

The two important items present in a pointer implementation of linked lists are

1. The data is stored in a collection of structures. Each structure contains the data and a pointer to the next structure.

2. A new structure can be obtained from the system's global memory by a call to *malloc* and released by a call to *free.*

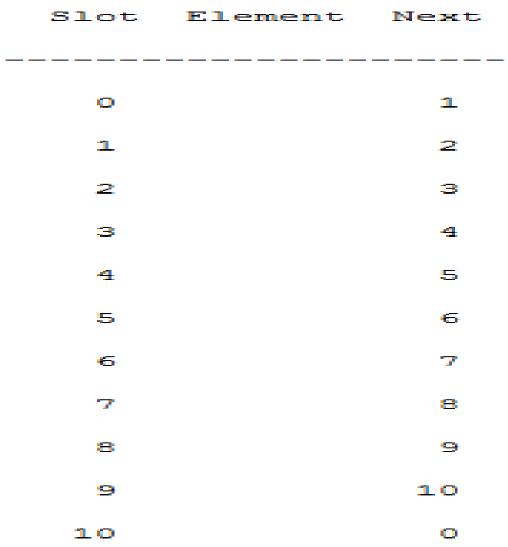
Our cursor implementation must be able to simulate this. The logical way to satisfy condition 1 is to have a global array of structures. For any cell in the array, its array index can be used in place of an address.

We must now simulate condition 2 by allowing the equivalent of *malloc* and *free* for cells in the *CURSOR_SPACE* array. To do this, we will keep a list (the *freelist*) of cells that are not in any list. The list will use cell 0 as a header.

A value of 0 for *next* is the equivalent of a *pointer*. The initialization of *CURSOR_SPACE* is a straightforward loop, which we leave as an exercise.

To perform an *malloc*, the first element (after the header) is removed from the freelist.

Declarations for cursor implementation of linked lists



Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

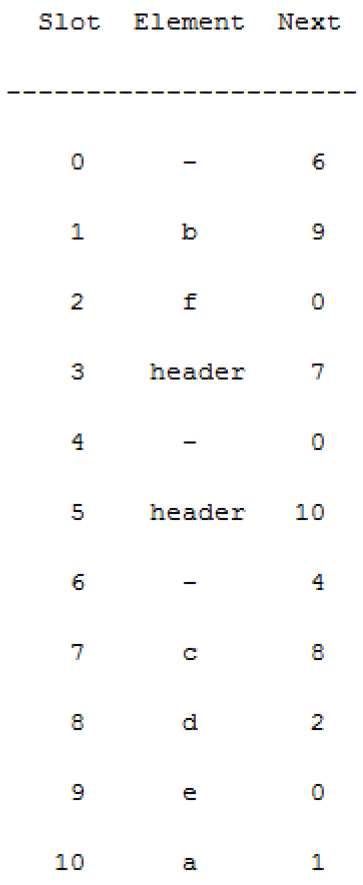
To perform a *free*, we place the cell at the front of the freelist.

Position cursor_alloc(void)

{

```
        position p;

        p = CURSOR_SPACE[O].next;

        CURSOR_SPACE[0].next = CURSOR_SPACE[p].next;

        return p;

}

Void cursor_free( position p)

{

        CURSOR_SPACE[p].next = CURSOR_SPACE[O].next;

        CURSOR_SPACE[O].next = p;

}
```

Example of a cursor implementation of linked lists



Slot	Element	Next
0	-	6
1	b	9
2	f	0
3	header	7
4	-	0
5	header	10
6	-	4
7	c	8
8	d	2
9	e	0
10	a	1

```
int is_empty( LIST L )  /* using a header node */

{

          return( CURSOR_SPACE[L].next == 0

}


int is_last( position p, LIST L)  /* using a header node */

{

          return( CURSOR_SPACE[p].next == 0

}

Position find( element_type x, LIST L) /* using a header node */

{

          position p;


          p = CURSOR_SPACE[L].next;

          while( p && CURSOR_SPACE[p].element != x )

                    p = CURSOR_SPACE[p].next;

          return p;

}

void delete( element_type x, LIST L )

{

          position p, tmp_cell;


          p = find_previous( x, L );
          if( !is_last( p, L) )

          {

                    tmp_cell = CURSOR_SPACE[p].next;

                    CURSOR_SPACE[p].next = CURSOR_SPACE[tmp_cell].next;

          cursor_free( tmp_cell );

          }

}

void insert( element_type x, LIST L, position p )

{

          position tmp_cell;

          tmp_cell = cursor_alloc( )

          if( tmp_cell ==0 )

                    fatal_error("Out of space!!!");

          else

          {

                    CURSOR_SPACE[tmp_cell].element = x;

                    CURSOR_SPACE[tmp_cell].next = CURSOR_SPACE[p].next;

                    CURSOR_SPACE[p].next = tmp_cell;

          }

}
```

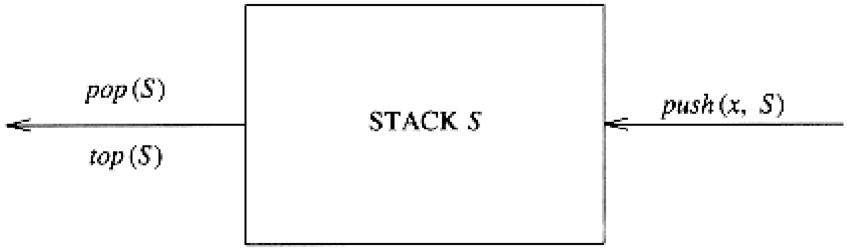
Stack ADT

A *stack* is a list with the restriction that *inserts* and *deletes* can be performed in only one position, namely the end of the list called the *top*.

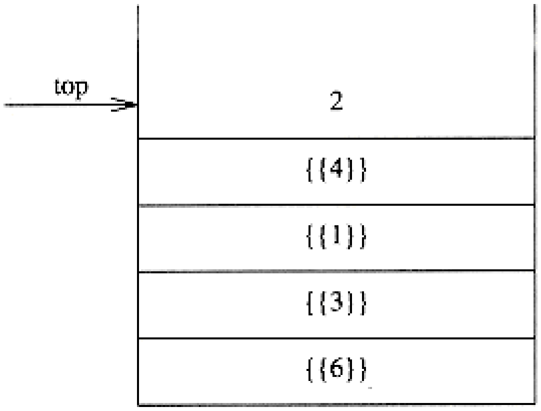
The fundamental operations on a stack are

- ***push***, which is equivalent to an insert,
- ***pop***, which deletes the most recently inserted element.

Stack Model



Example



Implementation of Stack

Linked List Implementation of Stacks

We perform a *push* by inserting at the front of the list. We perform a *pop* by deleting the element at the front of the list. A *top* operation merely examines the element at the front of the list, returning its value.

pop on an empty stack or a *push* on a full stack will overflow the array bounds and cause a crash.

```
typedef struct node *node_ptr;

struct node

{

element_type element;
node_ptr next;
};
typedef node_ptr STACK;

int is_empty( STACK S )
{
        return( S->next == NULL );
}
STACK create_stack( void )
{
        STACK S;
        S = (STACK) malloc( sizeof( struct node ) );
        if( S == NULL )
                fatal_error("Out of space!!!");
        return S;
}
void make_null( STACK S )
{
        if( S != NULL )
                S->next = NULL;
        else
                error("Must use create_stack first");
}
```

```
void push( element_type x, STACK S )
{
        node_ptr tmp_cell;
        tmp_cell = (node_ptr) malloc( sizeof ( struct node ) );
        if( tmp_cell == NULL )
                fatal_error("Out of space!!!");
        else
        {
                tmp_cell->element = x;
                tmp_cell->next = S->next;
                S->next = tmp_cell;
        }
}
element_type top( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->next->element;
}
Void pop( STACK S )
{
        node_ptr first_cell;
```

```
          if( is_empty( S ) )

                    error("Empty stack");

          else

          {

                    first_cell = S->next;

                    S->next = S->next->next;

                    free( first_cell );

          }

}
```

Array Implementation of Stacks

If we use an array implementation, the implementation is trivial. Associated with each stack is the top of stack, *tos*, which is -1 for an empty stack (this is how an empty stack is initialized).

To push some element *x* onto the stack, we increment *tos* and then set *STACK*[*tos*] = *x*, where *STACK* is the array representing the actual stack.

To pop, we set the return value to *STACK*[*tos*] and then decrement *tos*. Of course, since there are potentially several stacks, the *STACK* array and *tos* are part of one structure representing a stack.

```
struct stack_record

{

          unsigned int stack_size;

          int top_of_stack;

          element_type *stack_array;

};

typedef struct stack_record *STACK;

#define EMPTY_TOS (-1) /* Signifies an empty stack */
```

```
STACK create_stack( unsigned int max_elements )
{
        STACK S;
        if( max_elements < MIN_STACK_SIZE )
                error("Stack size is too small");
        S = (STACK) malloc( sizeof( struct stack_record ) );
        if( S == NULL )
                fatal_error("Out of space!!!");
        S->stack_array = (element_type *)
                malloc( sizeof( element_type ) * max_elements );
        if( S->stack_array == NULL )
                fatal_error("Out of space!!!");
        S->top_of_stack = EMPTY_TOS;
                /*S->stack_size = max_elements;
        return( S );
}
void dispose_stack( STACK S )
{
        if( S != NULL )
        {
        free( S->stack_array );
        free( S );
        }
}

int is_empty( STACK S )

{

        return( S->top_of_stack == EMPTY_TOS );

}

Void make_null( STACK S )

{

        S->top_of_stack = EMPTY_TOS;

}

Void push( element_type x, STACK S )

{

        if( is_full( S ) )

                error("Full stack");

else

S->stack_array[ ++S->top_of_stack ] = x;

}

element_type top( STACK S )

{

        if( is_empty( S ) )

                error("Empty stack");

        else

                return S->stack_array[ S->top_of_stack ];

}
```

```
void pop( STACK S )

{

        if( is_empty( S ) )

                error("Empty stack");

        else

                S->top_of_stack--;

}


element_type pop( STACK S )

{

        if( is_empty( S ) )

                error("Empty stack");

        else

                return S->stack_array[ S->top_of_stack-- ];

}
```

Applications of Stack

Infix to Postfix Conversion

The stack is used to convert the infix expression to postfix expression.

Infix

In Infix notation, the arithmetic operator appears between the two operands to which it is being applied.

For example: - A / B + C

Postfix

The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation. ((A/B) + C)

For example: - AB / C +

Algorithm

1. Read the infix expression one character at a time until we reach the end of input

   a) If the character is an operand, place it on to the output.
   b) If the character is a left parenthesis, push it onto the stack.
   c) If the character is a right parenthesis, pop all the operators from the stack until we encounters a left parenthesis, discard both the parenthesis in the output.
   d) If the character is an operator, then pop the entries from the stack until we find an entry of lower priority (never pop (‘‘). The push the operator into the stack.

2. Pop the stack until it is empty, writing symbols onto the output.

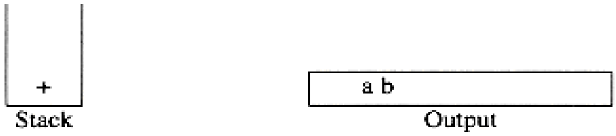
Example

Suppose we want to convert the infix expression

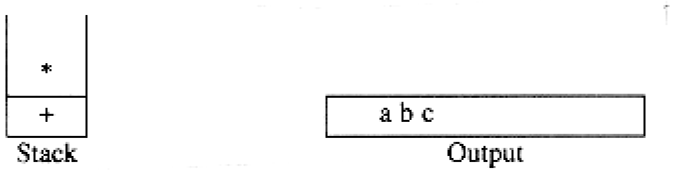
a+b*c+(d*e+f)*g

into postfix expression.

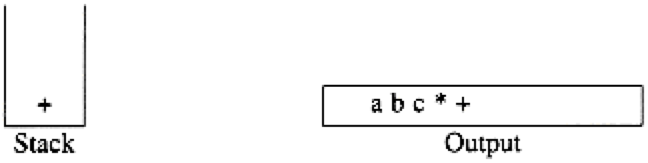
First, the symbol *a* is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next *b* is read and passed through to the output. The state is as follows:



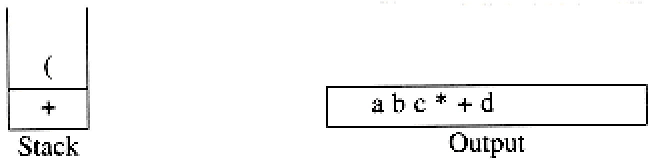
Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, *c* is read and output. Thus far, we have



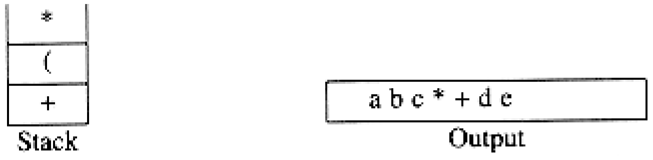
The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output, pop the other '+', which is not of *lower* but equal priority, on the stack, and then push the '+'.



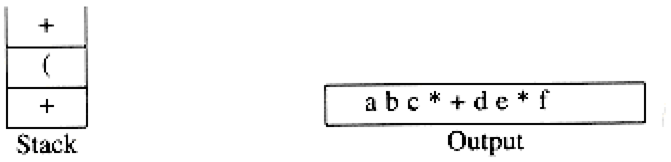
The next symbol read is an '(', which, being of highest precedence, is placed on the stack. Then *d* is read and output.



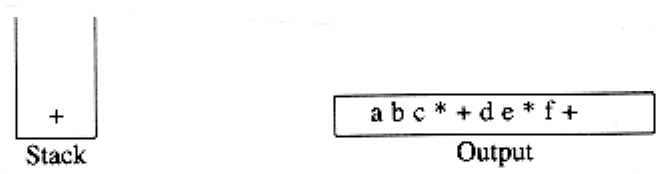
We continue by reading a '*'. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, *e* is read and output.



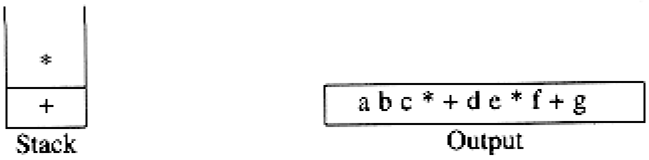
The next symbol read is a '+'. We pop and output '*' and then we push '+'. Then we read and output



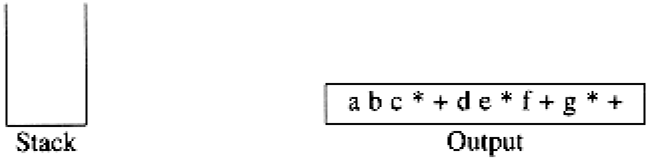
Now we read a ')', so the stack is emptied back to the '('. We output a '+'.



We read a '*' next; it is pushed onto the stack. Then *g* is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.

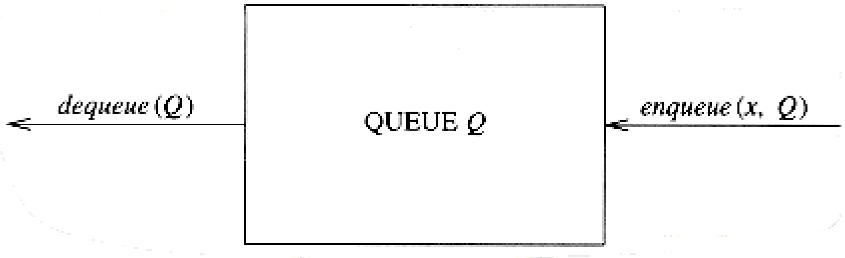


Queue ADT

Queue is an ordered collection of elements in that insertion is done at one end called rear, whereas deletion is performed at the other end called front.

The basic operations on a queue are

- enqueue, which inserts an element at the end of the list (called the rear)
- dequeue, which deletes (and returns) the element at the start of the list (known as the front).



Array Implementation of Queues

For each queue data structure, we keep an array, *QUEUE*[], and the positions *q_front* and *q_rear*, which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, *q_size*. All this information is part of one structure, and as usual, except for the queue routines themselves, no routine should ever access these directly.

To *enqueue* an element *x*, we increment *q_size* and *q_rear*, then set *QUEUE*[*q_rear*] = *x*.

To *dequeue* an element, we set the return value to *QUEUE*[*q_front*], decrement *q_size*, and then increment *q_front*.

There is one potential problem with this implementation. After 10 enqueues, the queue appears to be full, since *q_front* is now 10, and the next *enqueue* would be in a nonexistent position. However, there might only be a few elements in the queue, because several elements may have already been dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The simple solution is that whenever *q_front* or *q_rear* gets to the end of the array, it is wrapped around to the beginning. The following figure shows the queue during some operations. This is known as a *circular array* implementation.

Routine

```
struct queue_record

{

        unsigned int q_max_size;  /* Maximum # of elements */

                /* until Q is full */

        unsigned int q_front;

        unsigned int q_rear;

        unsigned int q_size;      /* Current # of elements in Q */

        element_type *q_array;

};
```

```
typedef struct queue_record * QUEUE;
int is_empty( QUEUE Q )
{
        return( Q->q_size == 0 );
}
Void make_null ( QUEUE Q )
{
        Q->q_size = 0;
        Q->q_front = 1;
        Q->q_rear = 0;
}
unsigned int succ( unsigned int value, QUEUE Q )
{
        if( ++value == Q->q_max_size )
        value = 0;
        return value;
}
void enqueue( element_type x, QUEUE Q )
{
        if( is_full( Q ) )
                error("Full queue");
        else
        {
                Q->q_size++;

                Q->q_rear = succ( Q->q_rear, Q );

                Q->q_array[ Q->q_rear ] = x;

           }

}
```

Application of Queue

- Telephone Call Processing
- Queuing Theory

Priority Queue(binary heap)

Heaps have two properties, namely,

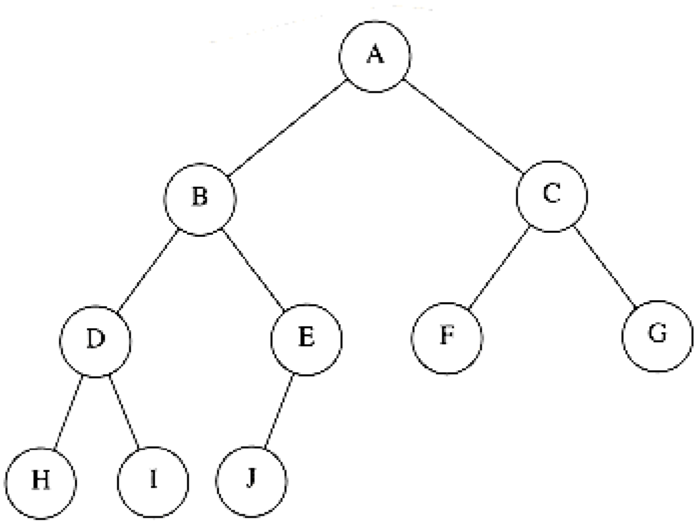
i) Structure property

ii) Heap order property.

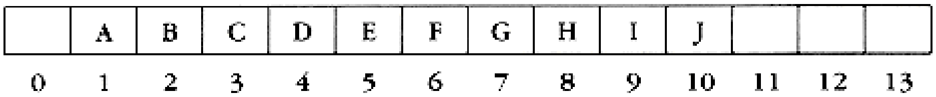
Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree.

Example



Array Implementation



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child (2i + 1), and the parent is in position *i*/2 .

Heap Order property

In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the obvious exception of the root (which has no parent).

```
struct heap_struct

{

        /* Maximum # that can fit in the heap */

    unsigned int max_heap_size;

        /* Current # of elements in the heap */

    unsigned int size;

    element_type *elements;

};
```

typedef struct heap_struct *PRIORITY_QUEUE;

Basic Heap Operation

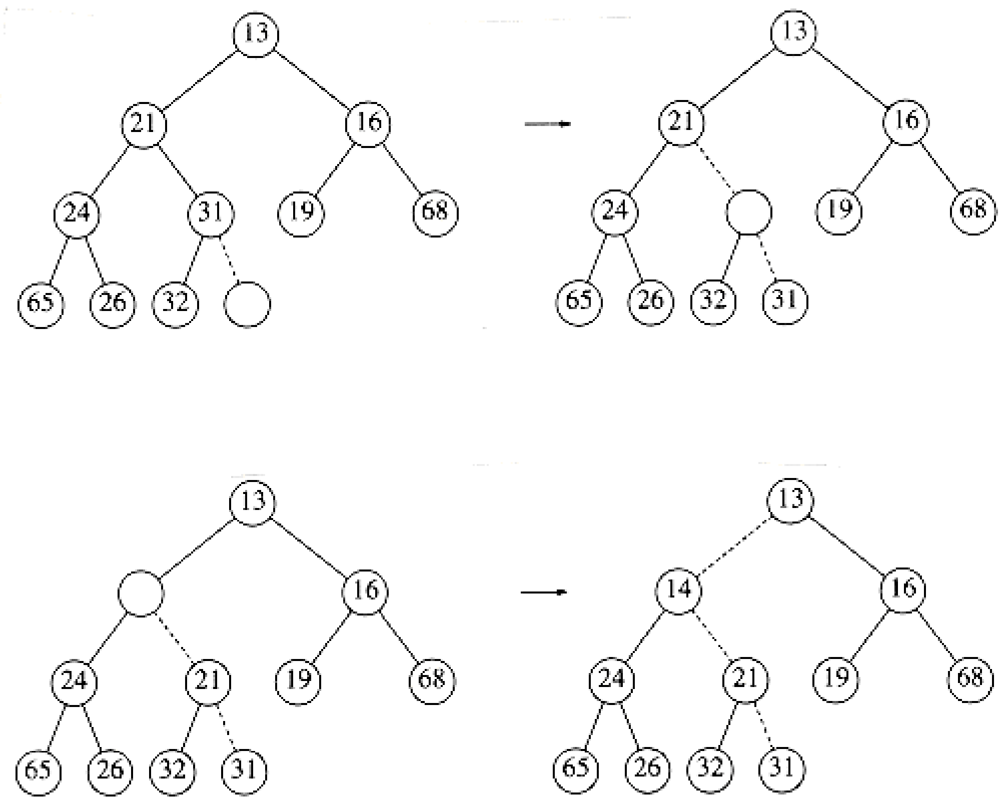
i) Insert
ii) Delete_min

Insert

To insert an element x into the heap, we create a hole in the next available location, since otherwise the tree will not be complete. If x can be placed in the hole without violating heap order, then we do so and are done.

Otherwise we slide the element that is in the whole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the hole. This general strategy is known as a percolate up; the new element is percolated up the heap until the correct location is found.

Insert 14



Routine

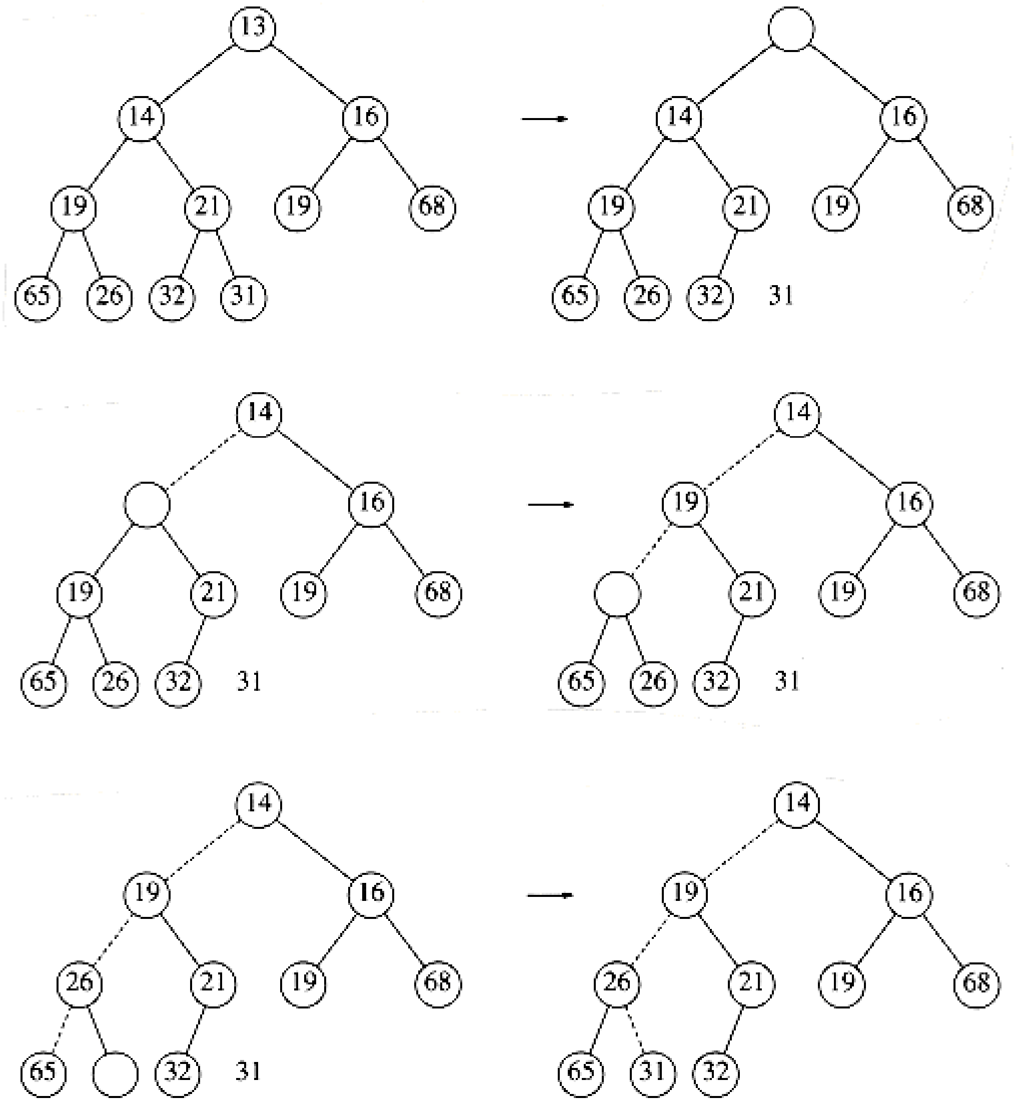
Void insert(element_type x, PRIORITY_QUEUE H)

```
{
        unsigned int i;
        if( is_full( H ) )
                error("Priority queue is full");
        else
        {
                i = ++H->size;
                while( H->elements[i/2] > x )
                {
                        H->elements[i] = H->elements[i/2];
                        i /= 2;
                }
                H->elements[i] = x;
        }
}
```

DeleteMin

Delete_mins are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element *x* in the heap must move somewhere in the heap. If *x* can be placed in the hole, then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until *x* can be placed in the hole. Thus, our action is to place *x* in its correct spot along a path from the root containing *minimum* children.

Example



Routine

```
element_type delete_min( PRIORITY_QUEUE H )

{

        unsigned int i, child;

        element_type min_element, last_element;

        if( is_empty( H ) )

        {

                error("Priority queue is empty");
```

```
    return H->elements[0];

    }

    min_element = H->elements[1];

    for( i=1; i*2 <= H->size; i=child )

    {

            /* find smaller child */

        child = i*2;

        if( ( child != H->size ) &&

            ( H->elements[child+1] < H->elements [child] ) )

          child++;

            /* percolate one level */

        if( last_element > H->elements[child] )

            H->elements[i] = H->elements[child];

        else

            break;

    }

    H->elements[i] = last_element;

    return min_element;

}
```

Heap applications

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.

- Selection algorithms: Finding the min, max, both the min and max, median, or even the *k*-th largest element can be done in linear time (often constant time) using heaps.[6]

- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

Full and almost full binary heaps may be represented in a very space-efficient way using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at position n would be at positions 2n and 2n+1 in a one-based array, or 2n+1 and 2n+2 in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by swapping elements which are out of order. As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

One more advantage of heaps over trees in some applications is that construction of heaps can be done in linear time using Tarjan's algorithm.

Hashing

Hashing is a technique used for performing insertion, deletion and finds in a constant average time.

Hash table

The hash table data structure is an array of some fixed size, containing the keys. The table size is generally denoted by the variable TableSize.

Each key is mapped into some numbers in the range o to TableSize-1 and placed in the appropriate cell.

0	
1	
2	
3	John 15467 2
4	Phil 67889 8
5	
6	Dave 14567 1
7	Mary 12345 6
8	
9	

In the above example, John hashes to 3 ,Phil hashes to 4, Dave hashes to 6 and Mary hashes to 7.

The main problem in hashing as to choose the hash function

Hash Function

There are two important properties to choose a Hash function.

- The function should be very easy and quick to compute.
- The function should give two different indices for two different key values.

There are so many methods to calculate the hash value

Method1

It is the simplest hash function. If the Input keys are integer, then we cal calculate the Index as

Key mod TableSize

```
typedef unsigned int  Index

Index Hash(int key , int TableSize)

{

        return key%TableSize;

}
```

Method2:

If the keys are strings , we have to use this hash function . This method adds the ASCII value of all the characters in the string.

```
Index Hash(char *key , int TableSize)

{

        Unsigned int Hahval=0;

        while(*key !='\0')

                Hashval +=*key++;
        return Hashval%TableSize;
}
```

Method 3:

This hash function assumes that the key has at least two characters plus the NULL terminator. This function examines only the first three characters.

The first character in the key is multiplied with the value 27 and the next character in the key is multiplied with the value 27^2 (729). The value 27 represents the number of letters in English alphabet.

```
Index Hash(char *key , int TableSize)
{
        return (key[0]*27+key[1]*729+key[2])%TableSize;
}
```

This function is easy to compute and it is not appropriate if the hash table is very large.

Method 4:

This function uses all the character in the key and can be expected to distribute well.

$$\sum key[keysize-i-1].32^i$$

The above code computes a polynomial function of 32 by using Horner's rule.

Ex

Hk =K2 + 32K1+322K0

This can be written as

Hk =((K0*32)+K1)*32+k2

Instead of multiplying 32 we have to use the bit-shift operator.

Routine

```
Index Hash(char *key , int TableSize)

{

        Unsigned int Hahval=0;



        while(*key !='\0')

                Hashval +=(Hashval<<5)+*key++;

        return Hashval%TableSize;

}
```

If the keys are very long, the hash function will take too long to compute.

UNIT – IV

NONLINEAR DATA STRUCTURES

Trees-Binary trees, search tree ADT, AVL trees, Graph Algorithms-Topological sort, shortest path algorithm network flow problems-minimum spanning tree - Introduction to NP - completeness.

Introduction

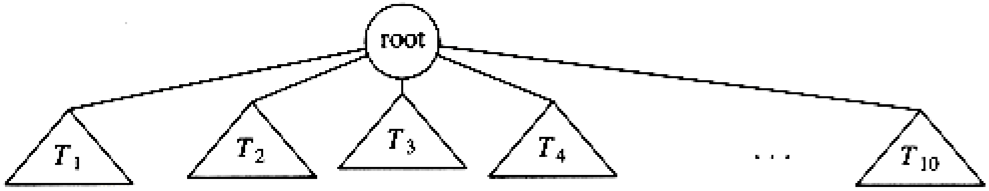
A **tree** is a nonlinear data structure. It is mainly used to represent data containing a hierarchical relationship between elements. Example: records, family trees and tables of contents.

Tree ADT

Definition

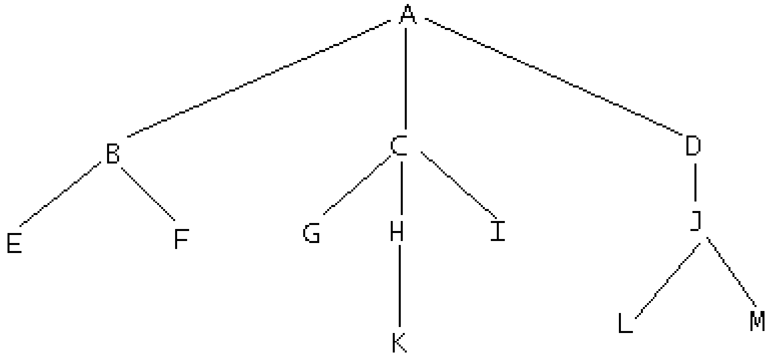
A tree is a finite set of one or more nodes such that there is a specially designated node called the Root, and zero or more non empty sub trees T_1, T_2....T_k, each of whose roots are connected by a directed edge from Root R.

Generic Tree



EXAMPLE:

Figure shows a tree T with 13 nodes, A, B, C, D, E, F, G, H, I, J, K, L, M



The root of a tree T is the node at the top, and the children of a node are ordered from left to right. Accordingly, A is the root of T, and A has three children; the first child B, the second child C and the third child D.

Observe that:

a. The node C has three children.

b. Each of the nodes B and K has two children.

c. Each of the nodes D and H has only one child.

d. The nodes E, F, G, K, I, M and L have no children.

The last groups of nodes, those with no children, are called **terminal nodes.**

Tree Terminology

Root

- The node at the top of the tree is called the root.
- There is only one root in a tree.

Parent

- If there is an edge from node *R* to node *M*, then *R* is a Parent of *M*.
- Any node (except the root) has exactly one edge running upward to another node. The node above it is called the **parent** of the node.
- Parents, grandparents, etc. are ancestors.

Child

- If there is an edge from node *R* to node *M*, then *M* is a **child** of *R*.
- Any node may have one or more lines running downward to other nodes. The nodes below a given node are called its **children**.

Ancestor

- If there is an path from node n1 to node n2, then n1 is an ancestors of n2.

- Parents, grandparents, etc. are ancestors.

Decendants

- If there is a path from node n1 to node n2, then n2 is a **child** of n1.
- Children, grandchildren, etc. are descendants

Sibilings

- Children of the same parent are called as *siblings*.

Leaf and Internal node

- A node that has no children is called a leaf node or simply a leaf. There can be only one root in a tree, but there can be many leaves.
- A node (apart from the root) that has children is an internal node or non-leaf node.

Path

- A path from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …,n_k such that n_i, is the parent of n_{i+1} for $1\leq i \geq k$,. The length of the path is $k – 1$.
- The length of the path is the number of edges on the path
- There is path of length zero from every node to itself.
- In a tree there is exactly one path from the Root to each node.

Levels

The level of a particular node refers to how many generations the node is from the root. If the root is assumed to be on level 0, then its children will be on level 1, its grandchildren will be on level 2, and so on.

Depth

- The depth of a node n_i is the length of the unique path from the Root to n_i.

- The Root is at depth 0.

Height

- The height of a node n_i is the length of the longest path from n_i to leaf.

- All leaves are at height 0.
- The height of the tree is equal to height o the Root.

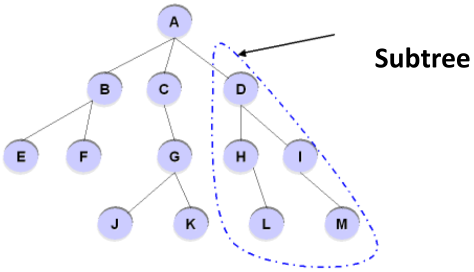
Degree of a node

The degree of a node is the number of subtrees of the node.

The node with degree 0 is a leaf or terminal node**.**

Subtree

- A node's subtree contains all its descendants.



Degree

- The number of subtrees of a node is called its degree.
- The degree of the tree is the maximum degree of any node in the tree.

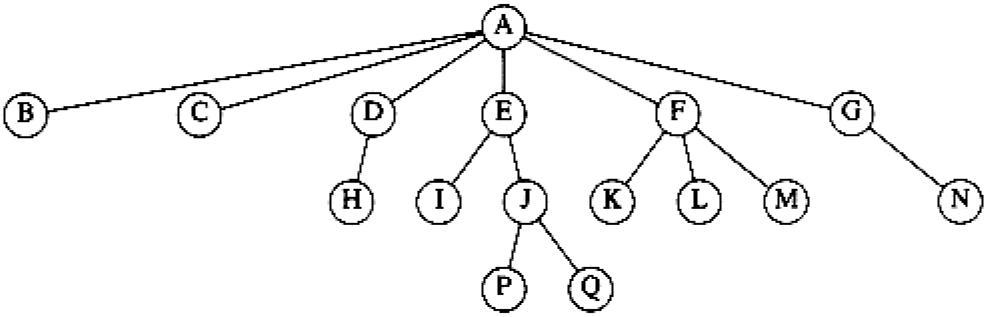
Implementation of Trees

Left child right sibling data structures for general trees

The best way to implement a tree is linked list. For that each node can have the data, a pointer to each child of the node. But the number of children per node can vary so greatly and is not known in advance, so it is infeasible to make the children direct links in the data structure, because there would be too much wasted space. The solution is simple: Keep the children of each node in a linked list of tree nodes. so the structure of the general tree contains the 3 fields

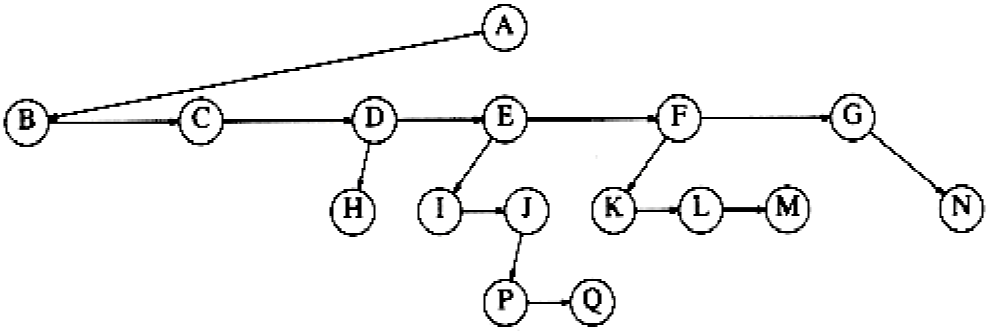
- Element
- Pointer to the Left Child
- Pointer to the Next Sibling

General Tree



Left Child/Right Sibling representation of the above tree.

In this representation, the Arrow that point downward are FirstChild pointers. Arrow that go left to right are NextSibling pointers.



Node Declaration

```
typedef struct TreeNode *PtrToNode;

struct TreeNode

{

        ElementType    Element;

        PtrToNode   FirstChild;

        ptrToNode  NextSibiling;

};
```

Binary Tree ADT

Definition

A **binary tree** is a tree in which each node has at most two children.

Example

The below shows that a binary tree consists of a root and two subtrees, T_l and T_r,

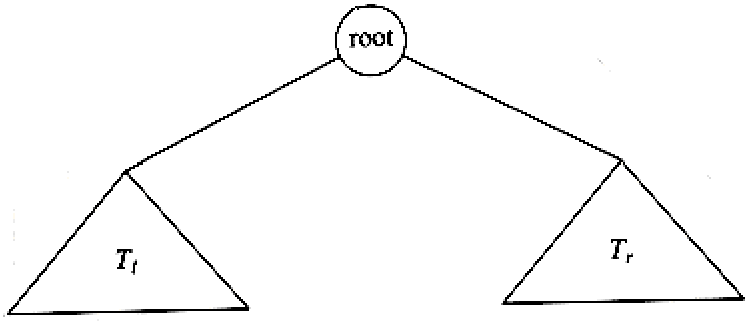


Figure: Generic Binary Tree

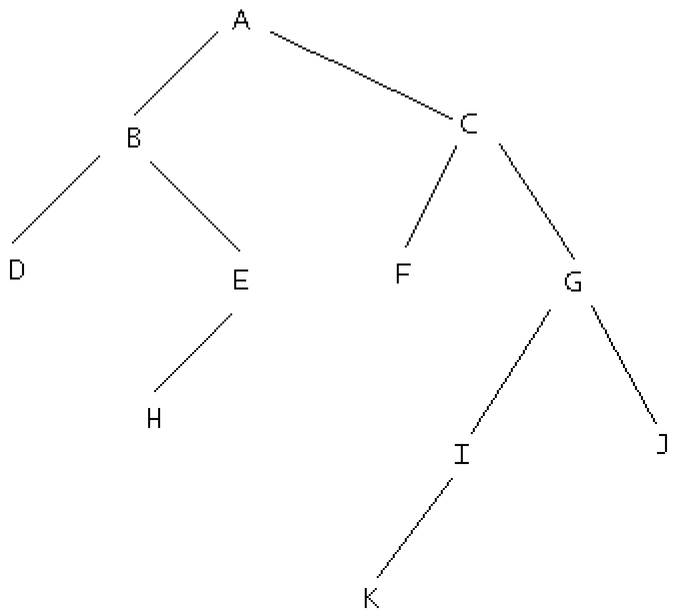


Figure: binary tree

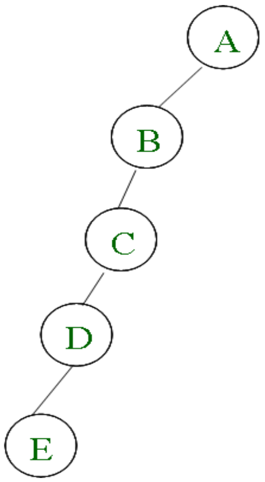
In the above figure

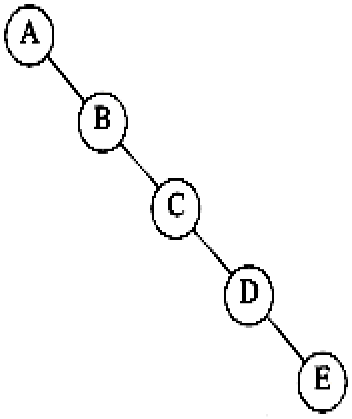
- Tree ,T consists of 11 nodes, represented by the letters **A** through **K.**
- The root of **the Tree** is the node **A .**
- B is a left child and **C** is a right child of the node **A**.
- The left subtree of the root **A** consists of the nodes **B, D, E** and **H**, and the right sub tree of the root **A** consists of the nodes **C, F, G, I, J, and K.**
  - Any node **N** in a binary tree **T** has 0, 1 or 2 successors.
  - The nodes **A, B, C** and **G** have two successors, the nodes **E** and **I** have only one successor, and the nodes **D, H, F, J** and **K** have no successors. The nodes with no successors are called **terminal nodes.**

Skewed Binary Tree

Skewed Binary tree is a binary tree in which all nodes other than the leaf node have only either the left or right child. If it has only a left child it is called as left skewed binary tree.

Left Skewed binary Tree Right Skewed binary Tree





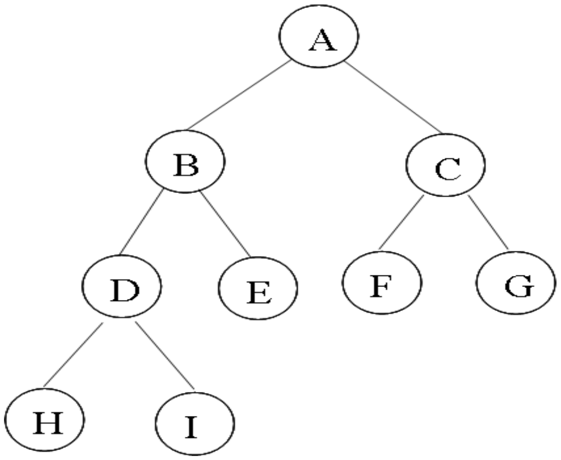
Binary Tree Representations

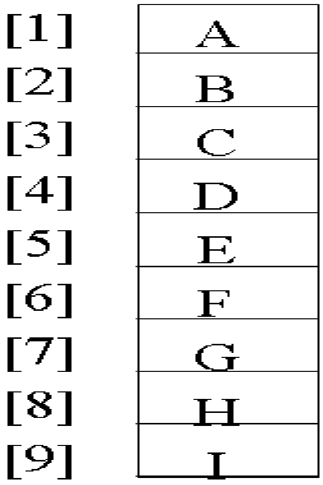
Array Representation (Sequential Representation)

The elements in the tree are represented using arrays. For any element in position i, the left child is in position 2i, the right child is in position (2i + 1), and the parent is in position (i/2).

Example

Binary Tree **Array Representation**

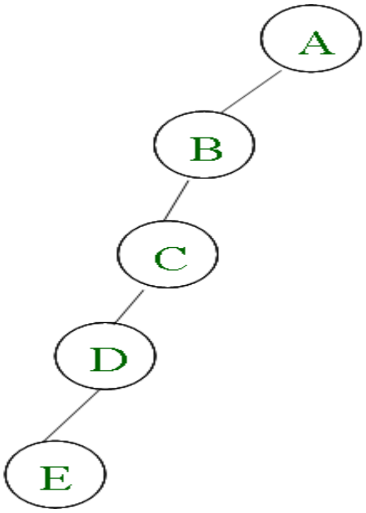




[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Example 2:

Binary Tree



Array Representation

[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

Disadvantages

- Wastage of space
- Insertion/deletion is a tedious process.

Linked Representation

A binary tree every node has atmost two children, so we can keep direct pointers to the children's. Every node in the tree structure can have 3 fields

- Element
- Pointer to the left subtree
- Pointer to the right subtree

It can be shown below

Element	left	right

Binary Tree Node Declaration

```
typedef struct TreeNode *PtrToNode;

typedef  struct PtrToNode Tree;

struct TreeNode

{

        ElementType    Element;

        Tree   Left

        Tree  Right;

};
```

Tree Traversals

Tree traversal is a process of moving through a tree in a specified order to process each of the nodes. Each of the nodes is processed only once (although it may be visited more than once). Usually, the traversal process is used to print out the tree.

There are three standard ways of traversing a binary tree T with root R.

- Preorder
- inorder
- postorder

PreOrder: (also called Element-Left-Right Traversal)

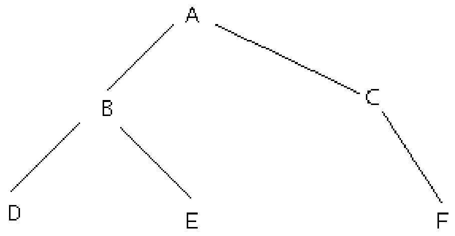
1. Process the root R

2. Traverse the left sub tree of R in preorder

3. Traverse the right subtree of R in preorder

EXAMPLE

Consider the binary tree T.



Observe that A is the root, that its left subtree consists of nodes B, D and E and that its right sub tree consists of nodes C and F.

The preorder traversal of T processes A, traverses left subtree and traverses right subtree. However, the pre order traversal of left subtree processes the root B and then D and E, and the preorder traversal of right subtree processes the root C and then F. Hence **ABDECF** is the **preorder traversal** of T.

Routine

```
void preorder (Tree T)

{

        if (T!= NULL)           /* Base case, if t is null, do nothing */

        {

                printf(“%c”, T → Element);   /* Print the value of the root */

                preorder (T → Left);            /* Traverse the left sub tree in in-order */

                preorder (T → Right);                   /* Traverse the right sub tree in in-
                order */ } }
```

Inorder: (also called Left-Element-Right Traversal)

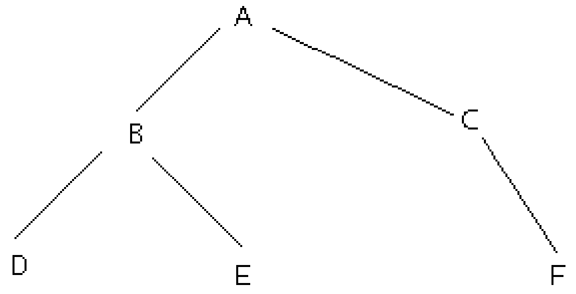
1. Traverse the left subtree of R in inorder.

2. Process the root R

3. Traverse the right subtree of R in inorder.

EXAMPLE

Consider the binary tree T.



Observe that A is the root, that its left subtree consists of nodes B, D and E and that its right sub tree consists of nodes C and F.

The inorder traversal of T traverses left subtree, processes A and traverses right subtree. However, the in order traversal of left subtree processes D, B and then E, and the inorder traversal of right subtree processes C and then F. Hence **DBEACF** is the **inorder traversal** of T.

Routine

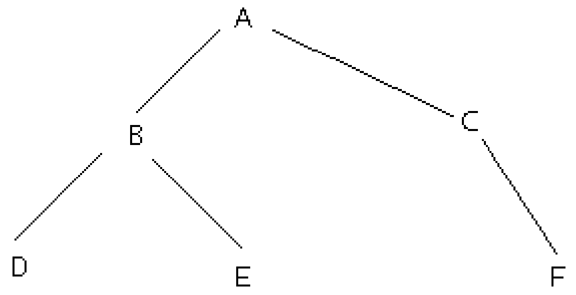
```
void inorder (Tree T)
{
        if (T!= NULL)           /* Base case, if p is null, do nothing */
        {
                inorder (T→ Left);              /* Traverse the left sub tree in in-order */
                printf(“%c”, T →Element);   /* Print the value of the root */
                inorder (T → Right);            /* Traverse the right sub tree in in-order */ }
}
```

Postorder: (also called Left-Right-Node Traversal)

1. Traverse the left subtree of R in postorder.
2. Traverse the right subtree of R in postorder.
3. Process the root R.

EXAMPLE

Consider the binary tree T.



Observe that A is the root, that its left subtree consists of nodes B, D and E and that its right sub tree consists of nodes C and F.

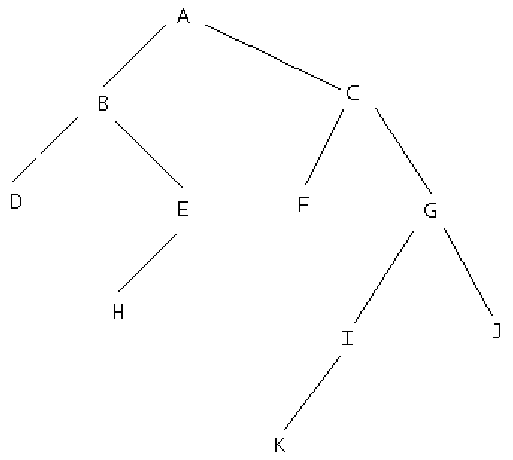
The postorder traversal of T traverses left subtree, traverses right subtree, and processes A. However, the postorder traversal of left subtree processes D, E and then B, and the postorder traversal of right subtree processes F and then C. Hence, **DEBFCA** is the **postorder traversal** of T.

Routine

```
void postorder (Tree T)

{

        if (T!= NULL)           /* Base case, if p is null, do nothing */

        {

                postorder (T → Left);                /* Traverse the left sub tree in in-order */

                postorder (T→ Right);          /* Traverse the right sub tree in in-order */

                printf(“%c”, T→ Element);    /* Print the value of the root */



        }
```

}

Example:



The preorder traversal of T is **A B D E H C F G I K J.**

The Inorder traversal of T is **D B H E A F C K I G J**

The Postorder traversal is **D H E B F K I J G C A.**

Expression Tree

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be travesed by inorder, preorder and postorder traversal.

Example

Constructing an Expression Tree

This algorithm is used to construct the expression tree from the given infix expression.

1. Convert the infix expression to postfix expression.
2. Create an Empty stack.
3. Read one symbol at a time from the postfix expression.
4. If the symbol is an operand

    i) create a one – node tree
    ii) Push the tree pointer on to the stack.

5. If the symbol is an operator

i) pop two pointers from the stack namely T_1 and T_2

ii) form a new tree with root as the operator and T_2 as a left child and T_1 as a right child.

iii) A pointer to this new tree is then pushed onto the stack.

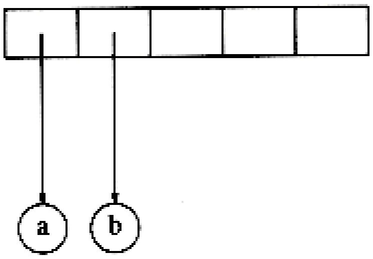
6. Repeat steps 3 to 5 until the end of input is reached

Example : -

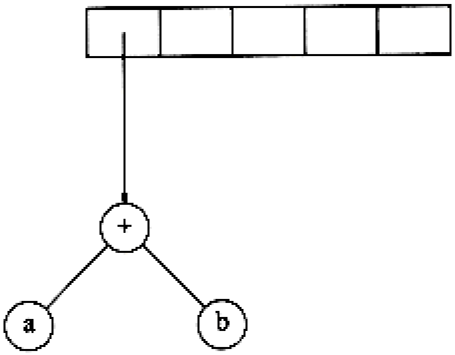
Consider an example, suppose the input is (a + b) * c * (d + e)

First we have to convert it into Postfix Expression. The Postfix Expression as a b + c d e + * *

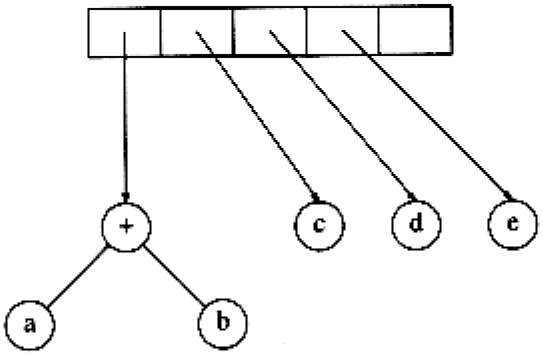
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*



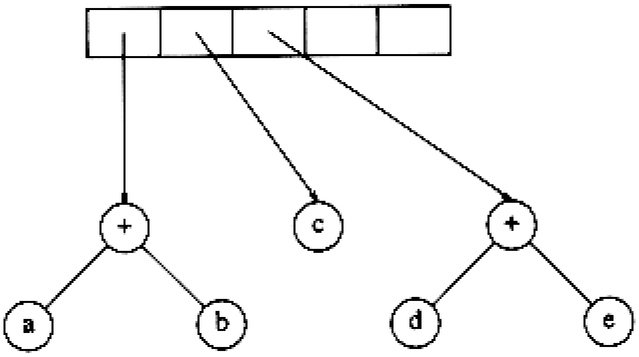
Next `+' symbol is read, so two pointers are popped, a new tree is formed and a pointer to this is pushed on to the stack.



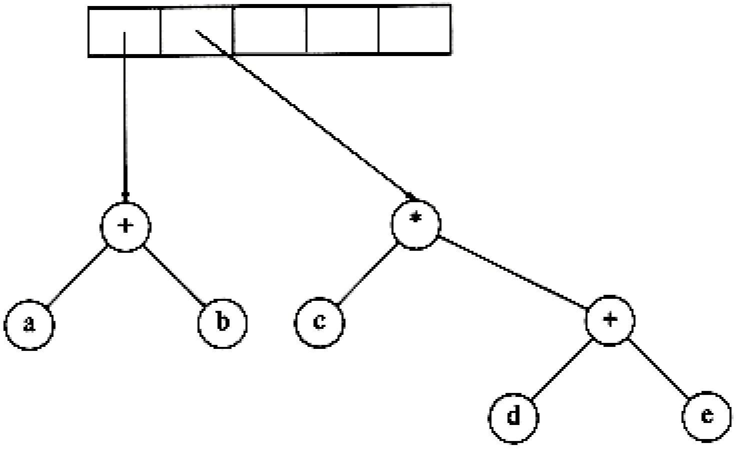
Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



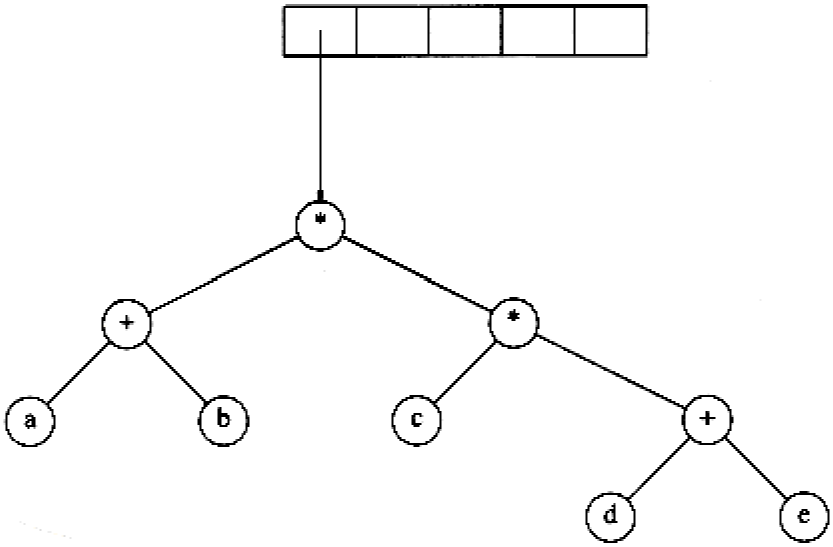
Now a '+' is read, so two trees are merged.



Next '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol '*' is read, two trees are merged, and a pointer to the final tree is left on the stack.



Applications of Binary Tree

1. Maintaining Directory Structure

Most of the Operating System the directory is maintained as a tree structure.

2. Expression Tree

A Compiler uses a binary tree to represent an arithmetic expression. Expression Tree is a binary tree in which the leaf nodes are operands and the non-leaf nodes are operators.

3. Binary Search Tree

This special form of binary search tree improves searching. Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

4. Threaded Binary Tree

A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.

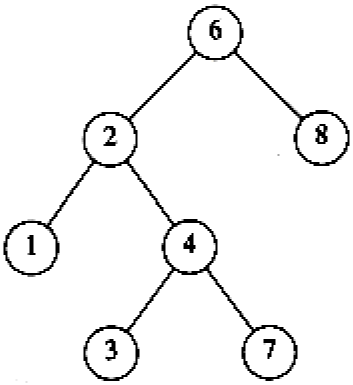
Binary Search Tree

Definition: -

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

A binary tree is a tree in which no node can have more than two children.

Example



Implementation of Binary Search Tree

Binary Search Tree can be implemented as a linked data structure in which each node with three fields. The three fields are the Element , pointer to the left child and pointer to the right child.

Binary Search Tree Declaration

```
typedef struct TreeNode  * SearchTree;

Struct TreeNode

{

        int Element ;

        SearchTree Left;

        SearchTree Right;

};
```

MakeEmpty

This operation is mainly used for initialization.

ROUTINE TO MAKE AN EMPTY TREE :-

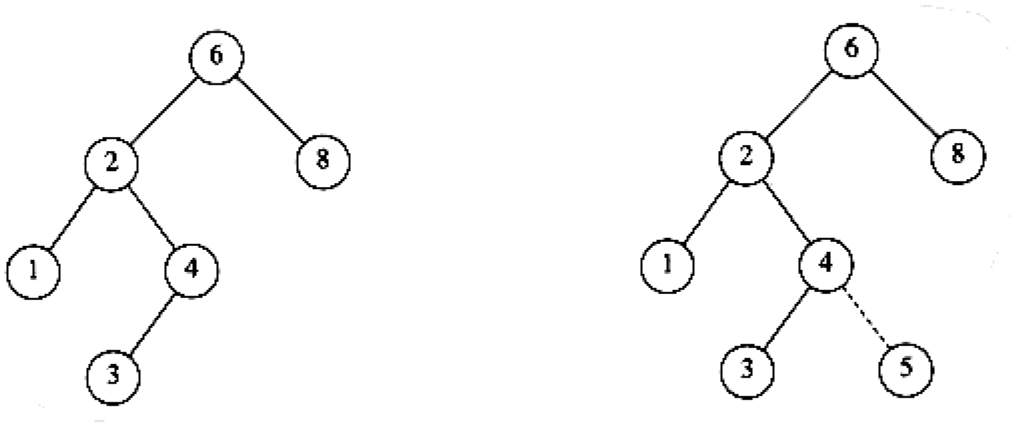
```
SearchTree MakeEmpty (SearchTree T)
{
        if (T! = NULL)
        {
                MakeEmpty (T->left);
                MakeEmpty (T-> Right);
                free (T);
        }
        return NULL ;
}
```

Insert

To insert the element X into the tree,

1. Check with the root node T
2. If it is less than the root, Traverse the left subtree recursively until it reaches the T-> left equals to NULL. Then X is placed in T->left.
3. If X is greater than the root. Traverse the right subtree recursively until it reaches the T-> right equals to NULL. Then x is placed in T->Right.

Binary Search Tree before and after inserting 5



ROUTINE TO INSERT INTO A BINARY SEARCH TREE

```
SearchTree Insert (int X, searchTree T)
{
        if (T = = NULL)
        {
                T = malloc (size of (Struct TreeNode));
                if (T! = NULL) // First element is placed in the root.
                {
                        T->Element = X;
                        T-> left = NULL;
                        T ->Right = NULL;
                }
        }
        else if (X < T ->Element)
                T ->left = Insert (X, T    left);
        else if (X > T     Element)

                T ->Right = Insert (X, T     Right);

        return T;

}
```

Find : -

The Find operation is used to return the pointer of the node that has key X, or NULL if there is no such node. It can be done as follows

- Check whether the root is NULL if so then return NULL.
- Otherwise, Check the value X with the root node value (i.e. T ->Element)

    If X is equal to T -> Element, return T.

    If X is less than T -> Element, Traverse the left of T recursively.

    If X is greater than T -> Element, traverse the right of T recursively.

ROUTINE FOR FIND OPERATION

```
int Find (int X, SearchTree T)

{

        if (T = = NULL)

                Return NULL ;

        if (X < T ->Element)

                return Find (X, T ->left);

        else if (X > T ->Element)

                return Find (X, T ->Right);
        else
                return T; // returns the position of the search element.
}
```

Find Min :

This operation returns the position of the smallest element in the tree.

To perform FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

RECURISVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)

{

        if (T = = NULL);

                return NULL ;

        else if (T->left = = NULL)

                return T;

        else

                return FindMin (T->left);

}
```

NON - RECURSIVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)

{

        if (T! = NULL)

                while (T->Left ! = NULL)

                        T = T->Left ;

        return T;

}
```

FindMax

FindMax routine return the position of largest elements in the tree. To perform a FindMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)

{

        if (T = = NULL)

                return NULL ;

        else if (T->Right = = NULL)

                return T;

        else FindMax (T->Right);

}
```

NON - RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)

{

        if (T! = NULL)

                while (T->Right ! = NULL)

                        T = T->Right ;

        return T ;

}
```

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

CASE 1: Node to be deleted is a leaf node (ie) No children.

CASE 2: Node with one child.

CASE 3: Node with two children.

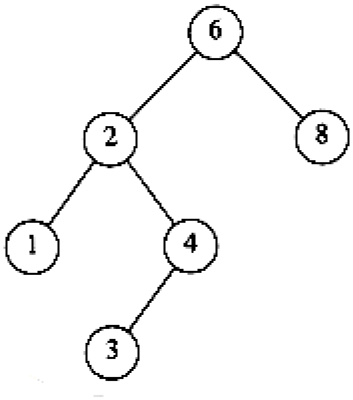
CASE 1: Node with no children (Leaf node)

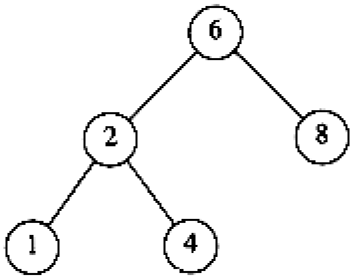
If the node is a leaf node, it can be deleted immediately.

Example

To Delete 3

Before Deletion **After Deleting 3**





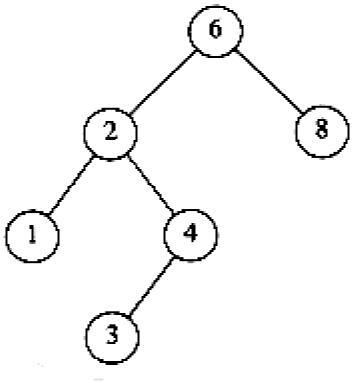
CASE 2 : - Node with one child

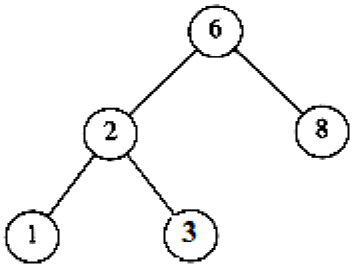
If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 4

To delete 4, the pointer currently pointing the node 4 is now made to to its child node 3.

Before Deletion **After Deleting 4**





Case 3 : Node with two children

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of its right subtree and recursively delete that node.

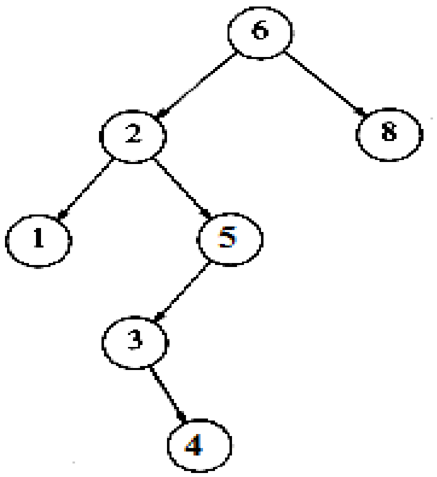
Example :

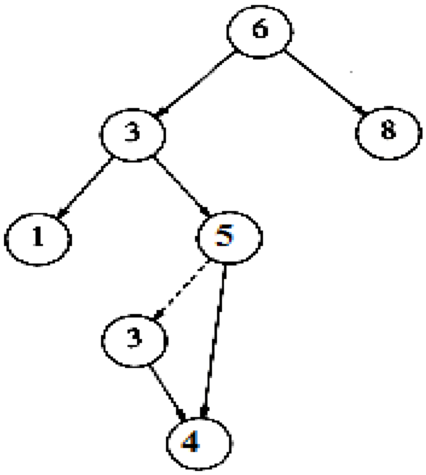
To Delete 2 :

The minimum element at the right subtree is 3.Now the value 3 is replaced in the position of 2.

Next delete 3 in the right subtree. Node 3 has only one child. So to delete 3, the pointer currently pointing the node 3 is now made to to its child node 4.

Before Deletion **After Deleting 2 (two children**
node)





DELETION ROUTINE FOR BINARY SEARCH TREES

```
SearchTree Delete (int X, searchTree T)

{

        int Tmpcell ;



        if (T = = NULL)

                Error ("Element not found");
```

```
    else if (X < T ->Element) // Traverse towards left
        T ->Left = Delete (X, T->Left);
    else if (X > T ->Element) // Traverse towards right
        T ->Right = Delete (X, T ->Right);
    else            // Found Element to be deleted
    if (T ->Left && T ->Right)   // Two children
    {                            // Replace with smallest data in right subtree
        Tmpcell = FindMin (T ->Right);
        T ->Element = Tmpcell ->Element ;
        T ->Right = Delete (T ->Element; T ->Right);
    }
    else // one or zero children
    {
        Tmpcell = T;
        if (T ->Left = = NULL)
            T = T ->Right;
        else if (T->Right = = NULL)
            T = T ->Left ;
        free (TmpCell);
    }
    return T;
}
```

<u>**AVL Tree : - (Adelson - Velskill and Landis)**</u>

An AVL tree is a binary search tree except that for every node in the tree, the height of the left and right subtrees can differ by atmost 1.The height of the empty tree is defined to be - 1.

A balance factor is the height of the left subtree minus height of the right subtree. For an AVL tree all balance factor should be +1, 0, or -1. If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced by making either single or double rotations.

An AVL tree causes imbalance, when any one of the following conditions occur.

Case 1 : An insertion into the left subtree of the left child of node.

Case 2 : An insertion into the right subtree of the left child of node.

Case 3 : An insertion into the left subtree of the right child of node.

Case 4 : An insertion into the right subtree of the right child of node.

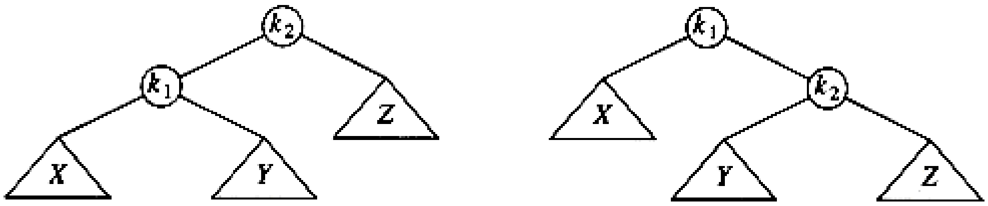
These imbalances can be overcome by

1. Single Rotation

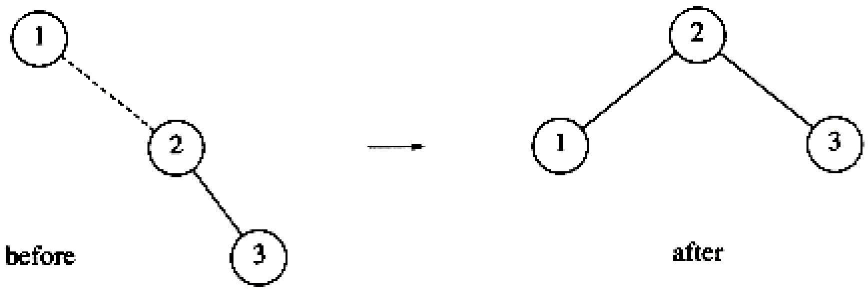
2. Double Rotation.

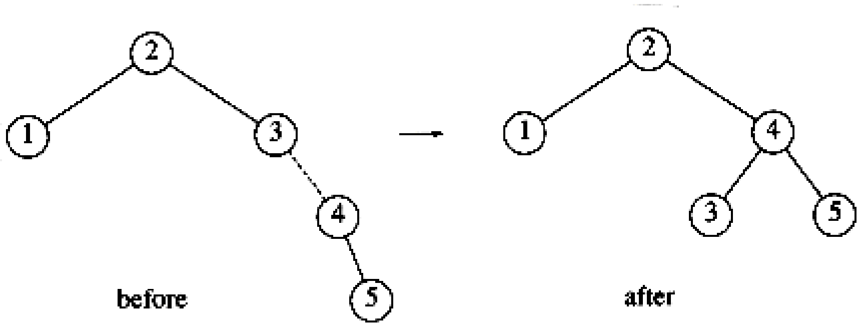
Single Rotation

Single Rotation is performed to fix case 1 and case 4.

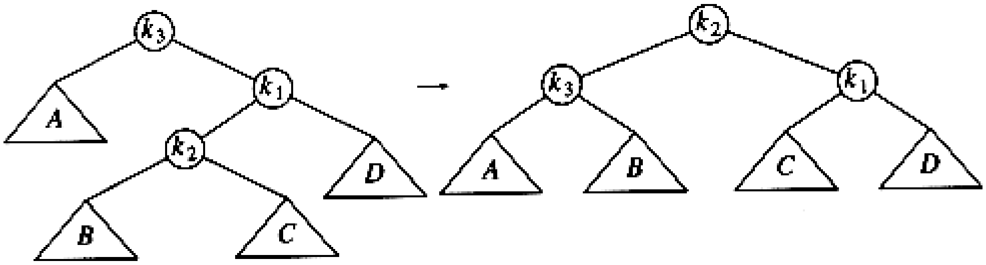


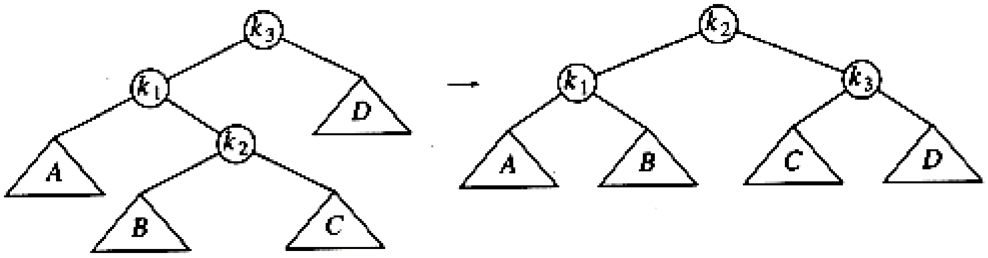
Example



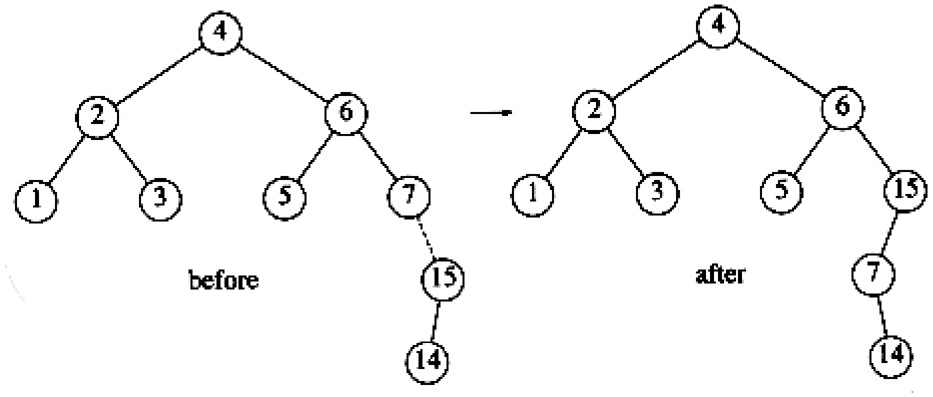


Double Rotation





Example



```
struct AvlNode
{
        ElementType Element;
        AvlTree  Left;
        AvlTree  Right;
        int      Height;
};
AvlTree MakeEmpty( AvlTree T )
{
        if( T != NULL )
        {
           MakeEmpty( T->Left );
           MakeEmpty( T->Right );
           free( T );
        }
        return NULL;
}
static int Height( Position P )
{
        if( P == NULL )
           return -1;
        else
           return P->Height;
}
Position Find( ElementType X, AvlTree T )
{
        if( T == NULL )
           return NULL;
        if( X < T->Element )
           return Find( X, T->Left );
        else
        if( X > T->Element )
           return Find( X, T->Right );
        else
           return T;
}
Position FindMin( AvlTree T )
```

```
{
        if( T == NULL )
           return NULL;
        else
        if( T->Left == NULL )
           return T;
        else
           return FindMin( T->Left );
}
Position FindMax( AvlTree T )
{
        if( T != NULL )
           while( T->Right != NULL )
              T = T->Right;
              return T;
}

static Position SingleRotateWithLeft( Position K2 )
 {
        Position K1;

        K1 = K2->Left;
        K2->Left = K1->Right;
```

```
            K1->Right = K2;

            K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) +1;

            K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;

            return K1;  /* New root */

  }

static Position SingleRotateWithRight( Position K1 )

{

            Position K2;

            K2 = K1->Right;

            K1->Right = K2->Left;

            K2->Left = K1;

            K1->Height = Max( Height( K1->Left ), Height( K1->Right ) )+ 1;

            K2->Height = Max( Height( K2->Right ), K1->Height ) + 1;

            return K2;  /* New root */

 }

static Position DoubleRotateWithLeft( Position K3 )

{

            /* Rotate between K1 and K2 */

            K3->Left = SingleRotateWithRight( K3->Left );

            /* Rotate between K3 and K2 */

            return SingleRotateWithLeft( K3 );

      }

static Position DoubleRotateWithRight( Position K1 )
```

```
{
        /* Rotate between K3 and K2 */
        K1->Right = SingleRotateWithLeft( K1->Right );
        /* Rotate between K1 and K2 */
        return SingleRotateWithRight( K1 );
}
AvllTree Insert( ElementType X, AvlTree T )
{
        if( T == NULL )
        {
          /* Create and return a one-node tree */
          T = malloc( sizeof( struct AvlNode ) );
          if( T == NULL )
            FatalError( "Out of space!!!" );
          else
          {
            T->Element = X; T->Height = 0;
            T->Left = T->Right = NULL;
          }
        }
        Else if( X < T->Element )
            {
          T->Left = Insert( X, T->Left );
```

```
            if( Height( T->Left ) - Height( T->Right ) == 2 )
                if( X < T->Left->Element )
                    T = SingleRotateWithLeft( T );
                else
                    T = DoubleRotateWithLeft( T );
        }
        else if( X > T->Element )
        {
            T->Right = Insert( X, T->Right );
            if( Height( T->Right ) - Height( T->Left ) == 2 )
                if( X > T->Right->Element )
                    T = SingleRotateWithRight( T );
                else
                    T = DoubleRotateWithRight( T );
        }
        /* Else X is in the tree already; we'll do nothing */
        T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
        return T;
}
```

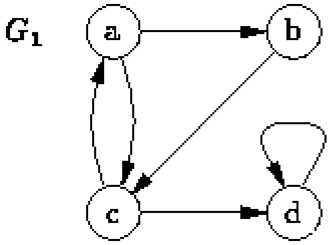
Graph Algorithms

Definition

A graph G = (V, E) consists of a set of vertices, V, and set of edges E. Each edge is a pair (v, w) where v, w εV. Vertices are referred to as nodes and the arc between the nodes are referred to as Edges.

Directed Graph (or) Digraph

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph. If (v, w) is a directed edge then (v, w) # (w, v).



The above graph comprised of four vertices and six edges:

V={a, b, c, d}

E={(a,b)(a,c)(b,c)(c,a)(c,d)(d,d)}

Adjacent Vertex

Vertex w is adjacent to v, if and only if there is an edge from vertex v to w (i.e. (v,w) ε E.

In the above graph vertex c is adjacent to and vertex b is adjacent to a and so on.

Undirected Graph

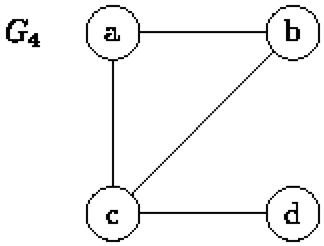
An undirected graph is a graph, which consists of undirected edges. If (v, w) is an undirected edge then (v,w) = (w, v).

consider the undirected graph G=(V1,E1) comprised of four vertices and four edges:

V1={a, b, c, d}

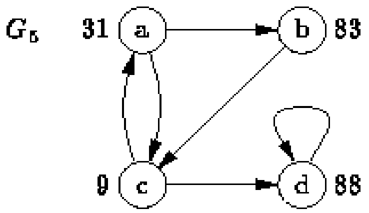
E1={{a, b}{a, c}{b .c},{c, d}}

The graph can be represented *graphically* as shown below

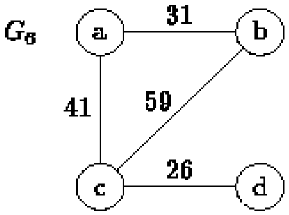


Weighted Graph

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.



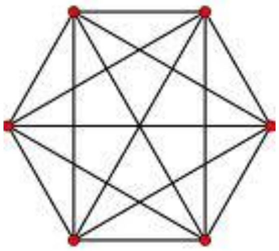
directed graph with labeled vertices



undirected graph with labeled edges

Complete Graph

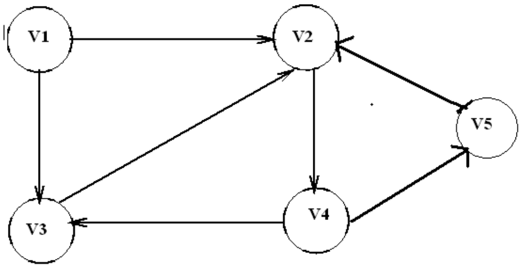
A complete graph is a graph in which there is an edge between every pair of vertices. A complete graph with n vertices will have n (n - 1)/2 edges.



The above graph contains 6 vertices and 15 edges.

Path

A path in a graph is a sequence of vertices w1,w2,w3, … , wn such that (wi , wi+1) ε E for 1 ≤ i <N.



The path from vertex v1 to v4 as v1,v2,v4

The path from vertex v1 to v5 as v1, v2.v4,v5.

Path Length

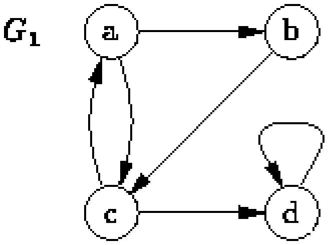
The length of the path is the number of edges on the path, which is equal to N-1, where N represents the number of vertices.

The length of the above path v1 to v5 as 3 . (i.e) (V_1, V_2), (V_2, V4) , (v4,v5).

If there is a path from a vertex to itself, with no edges, then the path length is 0.

Loop

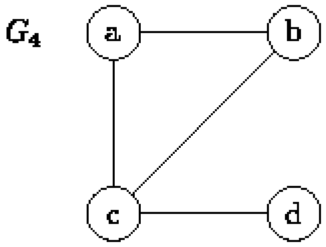
If the graph contains an edge (v, v) from a vertex to itself, then the path v,v is referred to as a loop.



The edge (d,d) is called as loop.

Simple Path

A simple path is a path such that all vertices on the path, except possibly the first and the last are distinct.



In the above graph the path (a ,b ,c, d) is a simple path.

Cycle

A cycle in a graph is a path in which the first and last vertexes are the same.

In the above graph a, b, a is a cycle

A graph which has cycles is referred to as cyclic graph.

Simple cycle

A simple cycle is the simple path of length at least one that begins and ends at the same vertex.

Degree

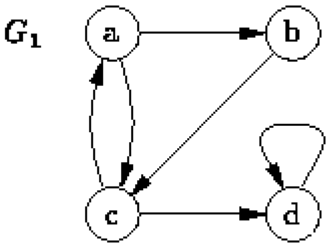
The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as degree (V).

The above graph degree of a vertex c is 3

Indegree and Outdegree

The indegree of the vertex V, is the number of edges entering into the vertex V.

Similarly the out degree of the vertex V is the number of edges exiting from that vertex V.

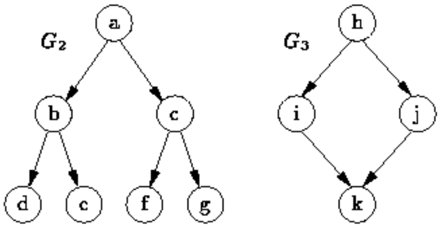


The Indegree of vertex c as 3

The Outdegree of vertex C as 1

Acyclic Graph

A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG (DAG - Directed Acyclic Graph).



Representation of Graph

Graph can be represented by two ways

i) Adjacency Matrix
ii) Adjacency list.

Adjacency Matrix

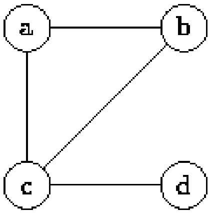
One simple way to represents a graph is Adjacency Matrix. The adjacency Matrix A for a graph G = (V, E) with n vertices is an n x n matrix, such that

A_{ij} = 1, if there is an edge V_i to V_j

A_{ij} = 0, if there is no edge.

Example

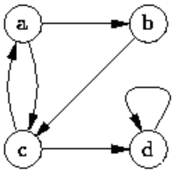
Adjacency Matrix for an Undirected Graph



$$A_1 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 1 & 0 \\ b & 0 & 0 & 1 & 0 \\ c & 1 & 0 & 0 & 1 \\ d & 0 & 0 & 0 & 1 \end{array}$$

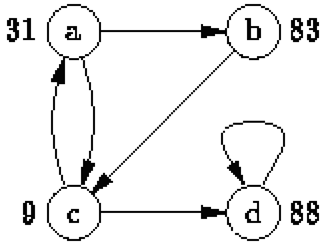
In the matrix (3,4) th data represent the presence of an edge between the vertices c and d.

Adjacency matrix for an directed graph



$$A_4 = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix for an Weighted Graph



$$A_6 = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} \infty & 31 & 41 & \infty \\ 31 & \infty & 59 & \infty \\ 41 & 59 & \infty & 26 \\ \infty & \infty & 26 & \infty \end{bmatrix} \end{array}$$

Advantage

- Simple to implement.

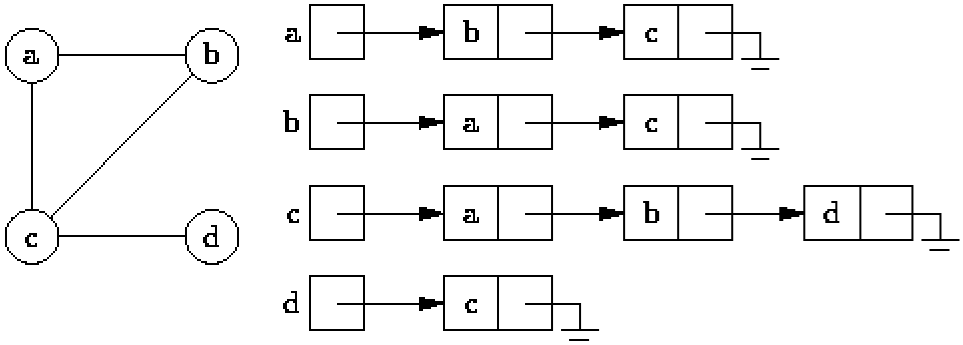
Disadvantage

- Takes O(n^2) space to represents the graph
- It takes O(n^2) time to solve the most of the problems.

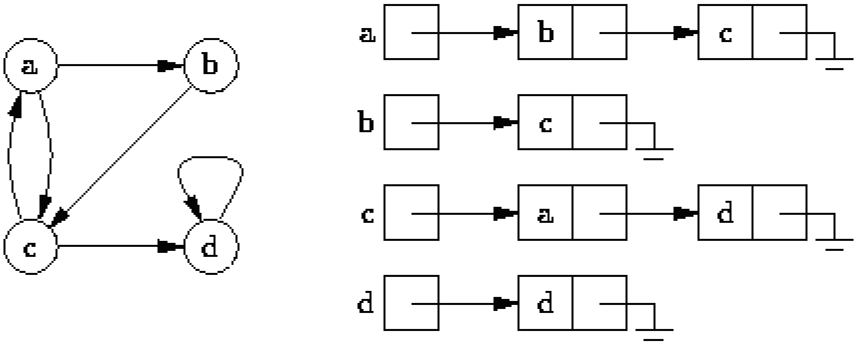
Adjacency List

In this representation for each vertex a list is maintained to keep all its adjacent vertices. It is the standard way to represent graphs

Example



Adjacency List For a Directed Graph



Topological Sort

A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from V_i to V_j, then V_j appears after V_i in the linear ordering.

Topological ordering is not possible. If the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

To implement the topological sort, perform the following steps.

1. Find the indegree for every vertex.
2. Place the vertices whose indegree is `0' on the empty queue.
3. Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.
4. Enqueue the vertex on the queue, if its indegree falls to zero.
5. Repeat from step 3 until the queue becomes empty.
6. The topological ordering is the order in which the vertices dequeued.

Routine to perform Topological Sort

/* Assume that the graph is read into an adjacency matrix and that the indegrees are

```
computed for every vertices and placed in an array (i.e. Indegree [ ] ) */

void Topsort (Graph G)

{

        Queue Q ;

        int counter = 0;

        Vertex V, W ;

        Q = CreateQueue (NumVertex);

        Makeempty (Q);

        for each vertex V

                if (indegree [V] = = 0)

                Enqueue (V, Q);

                while (! IsEmpty (Q))

                {

                        V = Dequeue (Q);

                        TopNum [V] = + + counter;

                        for each W adjacent to V

                                if (--Indegree [W] = = 0)

                                        Enqueue (W, Q);

                }

                if (counter ! = NumVertex)

                        Error (" Graph has a cycle");

        DisposeQueue (Q); /* Free the Memory */

}
```

Shortest Path Algorithm

The Shortest path algorithm determines the minimum cost of the path from source to every other vertex. The cost of a path v1v2…vn is ∑ci,i+1. This is referred to as the weighted path length. The unweighted path length is merely the number of the edges on the path, namely N - 1.

Two types of shortest path problems, exist namely,

1. The single source shortest path problem

2. The all pairs shortest path problem

The single source shortest path algorithm finds the minimum cost from single source vertex to all other vertices. Dijkstra's algorithm is used to solve this problem which follows the greedy technique.

All pairs shortest path problem finds the shortest distance from each vertex to all other vertices. To solve this problem dynamic programming technique known as floyd's algorithm is used.

These algorithms are applicable to both directed and undirected weighted graphs provided that they do not contain a cycle of negative length.

Single Source Shortest Path

Given an input graph G = (V, E) and a distinguished vertex S, find the shortest path from S to every other vertex in G. This problem could be applied to both weighted and unweighted graph.

Unweighted Shortest Path

In unweighted shortest path all the edges are assigned a weight of "1" for each vertex, The following three pieces of information is maintained.

Algorithm for unweighted graph

This algorith uses the following data structures

known

Specifies whether the vertex is processed or not. It is set to `1' after it is processed, otherwise `0'. Initially all vertices are marked unknown. (i.e) `0'.

dv

Specifies the distance from the source `s', initially all vertices are unreachable except for s, whose path length is `0'.

P_v

It is used to trace the actual path. That is the vertex which makes the changes in dv.

To implement the unweighted shortest path, perform the following steps :

1. Assign the source node as `s' and Enqueue `s'.
2. Dequeue the vertex `s' from queue and assign the value of that vertex to be known and then find its adjacency vertices.
3. If the distance of the adjacent vertices is equal to infinity then change the distance of that vertex as the distance of its source vertex increment by `1' and Enqueue the vertex.
4. Repeat from step 2, until the queue becomes empty.

ROUTINE FOR UNWEIGHTED SHORTEST PATH

```
void Unweighted (Table T)

{

        Queue Q;

        Vertex V, W ;

        Q = CreateQueue (NumVertex);

        MakeEmpty (Q);

                /* Enqueue the start vertex s */

        Enqueue (s, Q);

        while (! IsEmpty (Q))

        {

                V = Dequeue (Q);
                T[V].known=True;

                for each w adjacent to v

                        if(T[w].Dist==Infinity)

                        {

                                T[w].Dist = T[V].Dist + 1;

                                T[w].path = v ;

                                Enqueue(W,Q) ;

                        }

        }

        DisposeQueue(Q) ;     /* Free the memory */}
```

Dijkstra's Algorithm

The general method to solve the single source shortest path problem is known as Dijkstra's algorithm. This is applied to the weighted graph G.

Dijkstra's algorithm is the prime example of Greedy technique, which generally solve a problem in stages by doing what appears to be the best thing at each stage. This algorithm proceeds in stages, just like the unweighted shortest path algorithm.

At each stage, it selects a vertex v, which has the smallest dv among all the unknown vertices, and declares that as the shortest path from S to V and mark it to be known. We should set dw = dv + Cvw, if the new value for dw would be an improvement.

Routine For Algorithm

```
Void Dijkstra (Graph G, Table T)

{

        int i ;

        vertex V, W;

        Read Graph (G, T) /* Read graph from adjacency list */
```

```
    /* Table Initialization */
    for (i = 0; i < Numvertex; i++)
    {
        T [i]. known = False;
        T [i]. Dist = Infinity;
        T [i]. path = NotA vertex;
    }
    T [start]. dist = 0;
    for ( ; ;)
    {
        V = Smallest unknown distance vertex;
        if (V = = Not A vertex)
            break ;
        T[V]. known = True;
        for each W adjacent to V
            if ( ! T[W]. known)
            {
                T [W]. Dist = Min [T[W]. Dist, T[V]. Dist + $C_{VW}$]
                T[W]. path = V;
            }
    }
}
```

Network Flow Problems

Suppose we are given a directed graph *G* = (*V*, *E*) with edge capacities c*v*,*w*. These capacities could represent the amount of water that could flow through a pipe or the amount of traffic that could flow on a street between two intersections. We have two vertices: *s*, which we call the *source*, and *t*, which is the *sink*. Through any edge, (*v*, *w*), at most *cv*,*w* units of "*flow*" may pass. At any vertex, *v*, that is not either *s* or *t*, the total flow coming in must equal the total flow going out. The maximum flow problem is to determine the maximum amount of flow that can pass from *s* to *t*. As an example, for the graph in Figure on the left the maximum flow is 5, as indicated by the graph on the right.

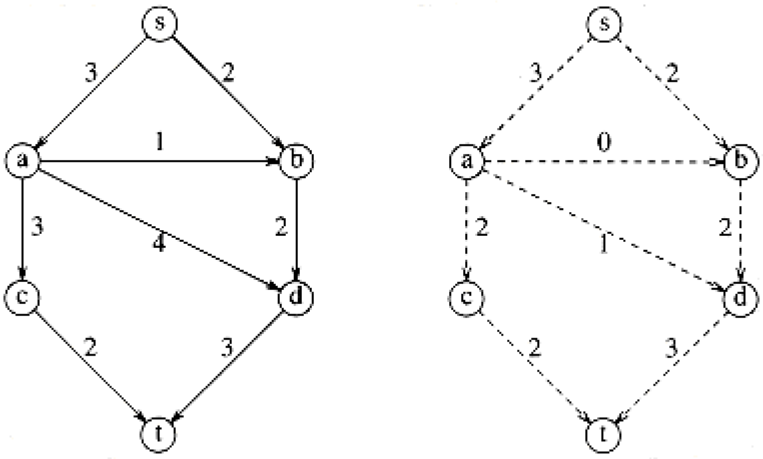


Figure 9.39 A graph (left) and its maximum flow

As required by the problem statement, no edge carries more flow than its capacity. Vertex *a* has three units of flow coming in, which it distributes to *c* and *d*. Vertex *d* takes three units of flow from *a* and *b* and combines this, sending the result to *t*. A vertex can combine and distribute flow in any manner that it likes, as long as edge capacities are not violated and as long as flow conservation is maintained (what goes in must come out).

<u>Minimum Spanning Tree</u>

Spanning Tree Definition

Consider a *connected*, *undirected* graph G=(V,E). A *spanning tree* of *G* is a subgraph of *G*, say T =(V’,E’), with the following properties:

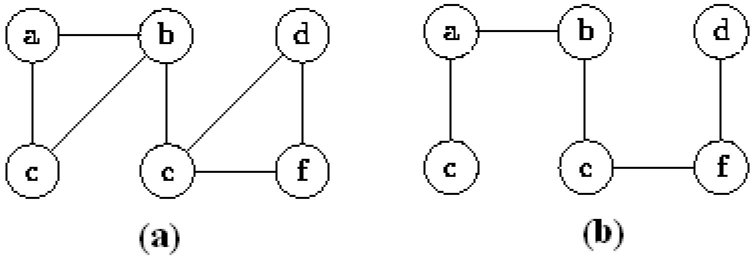
1. V’ = V (that is the spanning tree contains all the vertices of the graph)
2. *T* is connected.
3. *T* is acyclic.

A minimum spanning tree of a weighted connected graph G is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

Example

Undirected Graph

Spanning Tree



Prim's Algorithm

Prim's algorithm is used compute a minimum spanning tree. It uses a greedy technique. This algorithm begins with a set U initialized to {1}. It this grows a spanning tree, one edge at a time. At each step, it finds a shortest edge (u,v) such that the cost of (u, v) is the smallest among all edges, where u is in Minimum Spanning Tree and V is not in Minimum Spanning Tree.

SKETCH OF PRIM'S ALGORITHM

```
void Prim (Graph G)

{

        MSTTREE T;

        Vertex u, v;

        Set of vertices V;

        Set of tree vertices U;

        T = NULL;

        /* Initialization of Tree begins with the vertex `1' */

        U = {1}

        while (U # V)

        {
                Let (u,v) be a lowest cost such that u is in U and v is in V - U;

                T = T U {(u, v)};

                U = U U {V};

        }

}
```

ROUTINE FOR PRIMS ALGORITHM

```
void Prims (Table T)

{

        vertex V, W;

        /* Table initialization */

        for (i = 0; i < Numvertex ; i++)

        {

                T[i]. known = False;

                T[i]. Dist = Infinity;

                T[i]. path = 0;

        }

        for (; ;)

        {

                Let V be the start vertex with the smallest distance

                T[V]. dist = 0;

                T[V]. known = True;

                for each W adjacent to V

                        If (! T[W] . Known)
                {
                        T[W].Dist = Min(T[W]. Dist, $C_{VW}$);
                        T[W].path = V;
                }
        }
}
```

Kruskal's algorithm

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

```
function Kruskal($G$ = <$N$, $A$>: graph; length: $A \rightarrow R^+$): set of edges
Define an elementary cluster $C(v) \leftarrow \{v\}$.
Initialize a priority queue $Q$ to contain all edges in $G$, using the weights as keys.
Define a forest $T \leftarrow Ø$       //$T$ will ultimately contain the edges of the MST
 // n is total number of vertices
while $T$ has fewer than $n$-1 edges do
  // edge u,v is the minimum weighted route from u to v
  $(u,v) \leftarrow Q$.removeMin()
  // prevent cycles in T. add u,v only if T does not already contain a path between u and v.
  // the vertices has been added to the tree.
  Let $C(v)$ be the cluster containing $v$, and let $C(u)$ be the cluster containing $u$.
  if $C(v) \neq C(u)$ then
```

```
        Add edge $(v,u)$ to $T$.
        Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.
    return tree $T$
```

Introduction to NP – completeness

Most algorithms we have studied so far have polynomial-time running times. According to Cormen, Leiserson, and Rivest, polynomial-time algorithms can be considered tractable for the following reasons.

(1) Although a problem which has a running time of say $O(n^{20})$ or $O(n^{100})$ can be called intractable, there are very few practical problems with such orders of polynomial complexity.

(2) For reasonable models of computation, a problem that can be solved in polynomial time in one model can also be solved in polynomial time on another.

(3) The class of polynomial-time solvable problems has nice closure properties (since polynomials are closed under addition, multiplication, etc.)

The class of NP-complete (Non-deterministic polynomial time complete) problems is a very important and interesting class of problems in Computer Science. The interest surrounding this class of problems can be attributed to the following reasons.

1. No polynomial-time algorithm has yet been discovered for any NP-complete problem; at the same time no NP-complete problem has been shown to have a super polynomial-time (for example exponential time) lower bound.

2. If a polynomial-time algorithm is discovered for even one NP-complete problem, then all NP-complete problems will be solvable in polynomial-time.

It is believed (but so far no proof is available) that NP-complete problems do not have polynomial-time algorithms and therefore are intractable. The basis for this belief is the second fact above, namely that if any single NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial-time algorithm. Given the wide range of NP-complete problems that have been discovered to date, it will be sensational if all of them could be solved in polynomial time.

It is important to know the rudiments of NP-completeness for anyone to design "sound" algorithms for problems. If one can establish a problem as NP-complete, there is strong reason to believe that it is intractable. We would then do better by trying to design a good approximation algorithm rather than searching endlessly seeking an exact solution. An example of this is the TSP (Traveling Salesman Problem), which has been shown to be intractable. A practical strategy to solve TSP therefore would be to design a good approximation algorithm. This is what we did in Chapter 8, where we used a variation of Kruskal's minimal spanning tree algorithm to approximately solve the TSP. Another important reason to have good familiarity with NP-completeness is many natural interesting and innocuous-looking problems that on the surface seem no harder than sorting or searching, are in fact NP-complete.

The definition of NP-completeness is based on reducibility of problems. Suppose we wish to solve a problem X and we already have an algorithm for solving another problem Y. Suppose we have a function T that takes an input x for X and produces T(x), an input for Y such that the correct answer for X on x is yes if and only if the correct answer for Y on T(x) is yes. Then by composing T and the algorithm for y, we have an algorithm for X.

- If the function T itself can be computed in polynomially bounded time, we say X is polynomially reducible to Y and we write X $<=_p$ Y.
- If X is polynomially reducible to Y, then the implication is that Y is at least as hard to solve as X. i.e. X is no harder to solve than Y.
- It is easy to see that
  X <= Y and Y E **P** implies X E **P**.

A decision problem Y is said to be NP-hard if X **<=p Y for all X E NP.** An NP-hard problem Y is said to be NP-complete i**f Y E NP.** NPC is the standard notation for the class of all NP-complete problems.

- Informally, an NP-hard problem is a problem that is at least as hard as any problem in **NP**. If, further, the problem also belongs to **NP**, it would become NP-complete.
- It can be easily proved that if any NP-complete problem is in **P**, then **NP** = **P**. Similarly, if any problem in **NP** is not polynomial-time solvable, then no NP-complete problem will be polynomial-time solvable. Thus NP-completeness is at the crux of deciding whether or not **NP** = **P**.
- Using the above definition of NP-completeness to show that a given decision problem, say Y, is NP-complete will call for proving polynomial reducibility of each problem in **NP** to the problem Y. This is impractical since the class **NP** already has a large number of member problems and will continuously grow as researchers discover new members of **NP**.
- A much more practical way of proving NP-completeness of a decision problem Y is to discover a problem X E **NPC** such that Y <=p X. Since X is NP-complete and <=p is a transitive relationship, the above would mean that Z <=p Y for all Z E **NP**. Furthermore if Y E **NP**, then Y is NP-complete.

UNIT V

SORTING AND SEARCHING

Sorting – Insertion sort, Shell sort, Heap sort, Merge sort, Quick sort, Indirect sorting, Bucket sort, Introduction to Algorithm Design Techniques –Greedy algorithm (Minimum Spanning Tree), Divide and Conquer (Merge Sort), Dynamic Programming (All pairs Shortest Path Problem).

Insertion sort

Insertion sort belongs to the $O(n^2)$ sorting algorithms. Unlike many sorting algorithms with quadratic complexity, it is actually applied in practice for sorting small arrays of data.

```
void insertionSort(int arr[], int length) {

    int i, j, tmp;

    for (i = 1; i < length; i++) {

        j = i;

        while (j > 0 && arr[j - 1] > arr[j]) {

            tmp = arr[j];

            arr[j] = arr[j - 1];

            arr[j - 1] = tmp;

            j--;

        }

    }

}
```

Shell sort

Shellsort is one of the oldest sorting algorithms, named after its inventor D.L. Shell (1959). It is fast, easy to understand and easy to implement. However, its complexity analysis is a little more sophisticated.

The idea of Shellsort is the following:

a. arrange the data sequence in a two-dimensional array

b. sort the columns of the array

The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column. In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted. However, the number of sorting operations necessary in each step is limited, due to the presortedness of the sequence obtained in the preceding steps.

Example: Let 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 be the data sequence to be sorted. First, it is arranged in an array with 7 columns (left), then the columns are sorted (right):

$$\begin{array}{ccccccc} 3 & 7 & 9 & 0 & 5 & 1 & 6 \\ 8 & 4 & 2 & 0 & 6 & 1 & 5 \\ 7 & 3 & 4 & 9 & 8 & 2 & \end{array} \quad \rightarrow \quad \begin{array}{ccccccc} 3 & 3 & 2 & 0 & 5 & 1 & 5 \\ 7 & 4 & 4 & 0 & 6 & 1 & 6 \\ 8 & 7 & 9 & 9 & 8 & 2 & \end{array}$$

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is also still there. In the next step, the sequence is arranged in 3 columns, which are again sorted:

$$\begin{array}{ccc} 3 & 3 & 2 \\ 0 & 5 & 1 \\ 5 & 7 & 4 \\ 4 & 0 & 6 \\ 1 & 6 & 8 \\ 7 & 9 & 9 \\ 8 & 2 & \end{array} \quad \rightarrow \quad \begin{array}{ccc} 0 & 0 & 1 \\ 1 & 2 & 2 \\ 3 & 3 & 4 \\ 4 & 5 & 6 \\ 5 & 6 & 8 \\ 7 & 7 & 9 \\ 8 & 9 & \end{array}$$

Now the sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 6, an 8 and a 9 that have to move a little bit to their correct position.

```
void shellsort (int[] a, int n)

{

   int i, j, k, h, v;

   int[] cols = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,

                 1968, 861, 336, 112, 48, 21, 7, 3, 1}

   for (k=0; k<16; k++)

   {

      h=cols[k];
```

```
        for (i=h; i<n; i++)
        {
            v=a[i];
            j=i;
            while (j>=h && a[j-h]>v)
            {
                a[j]=a[j-h];
                j=j-h;
            }
            a[j]=v;
        }
    }
}
```

The correctness of the algorithm follows from the fact that in the last step (with *h* = 1) an ordinary Insertion Sort is performed on the whole array. But since data are presorted by the preceding steps (*h* = 3, 7, 21, ...) only few Insertion Sort steps are sufficient. How many exactly will be the subject of the following analysis. The above sequence of *h*'s (denoted as *h*-sequence in the following) is just one of several possible; actually, the performance of Shellsort depends on which *h*-sequence is used.

Heap sort

The heap sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is O(*n log n*) and like insertion sort, heap sort sorts in-place. The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
 void heapSort(int numbers[], int array_size)
```

```
{
  int i, temp;
  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);
  for (i = array_size-1; i >= 1; i--)
  {
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}
void siftDown(int numbers[], int root, int bottom)
{
  int done, maxChild, temp;
  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] > numbers[root * 2 + 1])
      maxChild = root * 2;
    else
```

```
      maxChild = root * 2 + 1;
    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}
```

Merge sort

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of *O*(*n* log(*n*)) Mergesort is optimal.

Similar to Quicksort, the Mergesort algorithm is based on a divide and conquer strategy. First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*) .

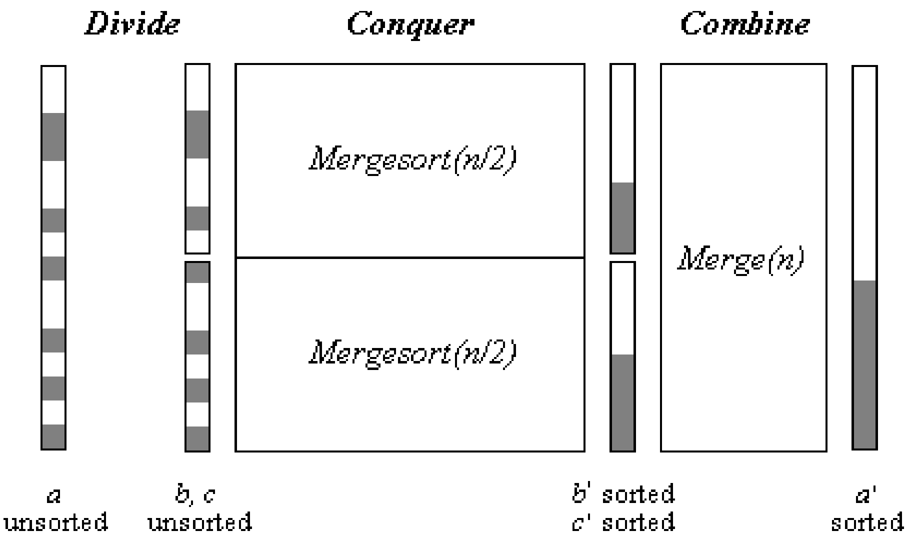


Figure 1: Mergesort(n)

The following procedure *mergesort* sorts a sequence *a* from index *lo* to index *hi*.

```
void mergesort(int lo, int hi)

{

   if (lo<hi)

   {

      int m=(lo+hi)/2;

      mergesort(lo, m);

      mergesort(m+1, hi);

      merge(lo, m, hi);

   }

}
```

First, index *m* in the middle between *lo* and *hi* is determined. Then the first part of the sequence (from *lo* to *m*) and the second part (from *m*+1 to *hi*) are sorted by recursive calls of *mergesort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when *lo* = *hi*, i.e. when a subsequence consists of only one element.

The main work of the Mergesort algorithm is performed by function *merge*. There are different possibilities to implement this function.

Function *merge* is usually implemented in the following way: The two halves are first copied into an auxiliary array *b*. Then the two halves are scanned by pointers *i* and *j* and the respective next-greatest element at each time is copied back to array *a* (Figure 2).

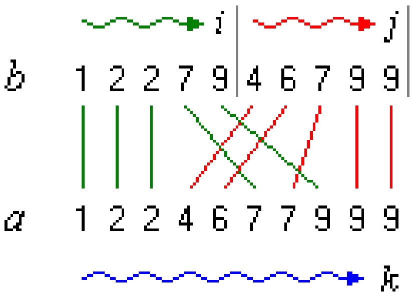


Figure 2: Merging two sorted halves

At the end a situation occurs where one index has reached the end of its half, while the other has not. Then, in principle, the rest of the elements of the corresponding half have to be copied back. Actually, this is not necessary for the second half, since (copies of) the remaining elements are already at their proper places.

```
void merge(int lo, int m, int hi)

{

   int i, j, k;

   i=0; j=lo;

   // copy first half of array a to auxiliary array b

   while (j<=m)

      b[i++]=a[j++];

   i=0; k=lo;
```

```
    // copy back next-greatest element at each time
    while (k<j && j<=hi)
        if (b[i]<=a[j])
            a[k++]=b[i++];
        else
            a[k++]=a[j++];
    // copy back remaining elements of first half (if any)
    while (k<j)
        a[k++]=b[i++];
}
```

Quick sort

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes.

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. **Sort both parts.** Apply quicksort algorithm recursively to the left and the right parts.

There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and **j** points to the last one. Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found. Index **j** is moved backward, until an element with value lesser or equal to the pivot is found. If **i ≤ j** then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when **i** becomes greater than **j**.

After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

> On the average quicksort has O(n log n) complexity, but strong proof of this fact is not trivial and not presented here. Still, you can find the proof in [1]. In worst case, quicksort runs O(n^2) time, but on the most "practical" data it works just fine and outperforms other O(n log n) sorting algorithms

```
void quickSort(int arr[], int left, int right) {

      int i = left, j = right;

      int tmp;

      int pivot = arr[(left + right) / 2];

       /* partition */

      while (i <= j) {

            while (arr[i] < pivot)

                  i++;

            while (arr[j] > pivot)

                  j--;

            if (i <= j) {

                  tmp = arr[i];

                  arr[i] = arr[j];

                  arr[j] = tmp;

                  i++;

                  j--;

            }

      };

       /* recursion */
```

```
    if (left < j)

        quickSort(arr, left, j);

    if (i < right)

        quickSort(arr, i, right);

}
```

Indirect Sorting

- If we are sorting an array whose elements are large, copying the items can be very expensive.
- We can get around this by doing *indirect sorting*.
    - o We have an additional array of pointers where each element of the pointer array points to an element in the original array.
    - o When sorting, we compare keys in the original array but we swap the element in the pointer array.
- This saves a lot of copying at the expense of indirect references to the elements of the original array.

Bucket sort

Bucket sort is possibly the simplest distribution sorting algorithm. The essential requirement is that the size of the universe from which the elements to be sorted are drawn is a small, fixed constant, say *m*.

For example, suppose that we are sorting elements drawn from **{0, 1, . . ., m-1}**, i.e., the set of integers in the interval **[0, *m*-1]**. Bucket sort uses ***m*** counters. The $\boldsymbol{i^{th}}$ counter keeps track of the number of occurrences of the $\boldsymbol{i^{th}}$ element of the universe. The figure below illustrates how this is done.

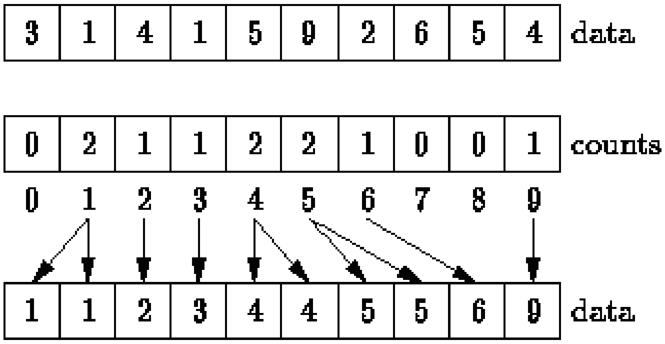


Figure: Bucket Sorting

In the figure above, the universal set is assumed to be **{0, 1, . . ., 9}**. Therefore, ten counters are required-one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. E.g., the sorted sequence contains no zeroes, two ones, one two, and so on.

```
void bucketSort(dataElem array[], int array_size)

        {

                int i, j;

                dataElem count[array_size];

                for(i =0; i < array_size; i++)

                        count[i] = 0;

                for(j =0; j < array_size; j++)

                        ++count[array[j]];

                for(i =0, j=0; i < array_size; i++)

                        for(; count[i]>0; --count[i])

                                array[j++] = i;

        }
```

Introduction to Algorithm Design Techniques

Greedy algorithm

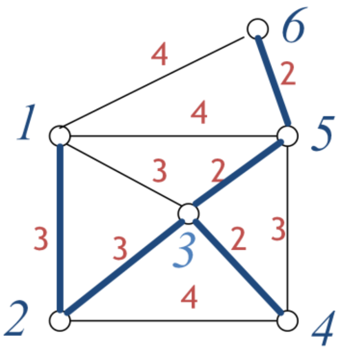
- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
    - You take the best you can get right now, without regard for future consequences
    - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be:
  At each step, take the largest possible bill or coin that does not overshoot
    - Example: To make $6.39, you can choose:
        - a $5 bill
        - a $1 bill, to make $6
        - a 25¢ coin, to make $6.25
        - A 10¢ coin, to make $6.35
        - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

Minimum spanning tree

- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
    - Start by picking any node and adding it to the tree
    - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
    - Stop when all nodes have been added to the tree



- The result is a least-cost (3+3+2+2+2=12) spanning tree

- If you think some other edge should be in the spanning tree:
    - Try adding that edge
    - Note that the edge is part of a cycle
    - To break the cycle, you must remove the edge with the greatest cost
        - This will be the edge you just added

Divide-and-Conquer Algorithms

A divide-and-conquer algorithm

- Derives the output directly, for small instances
- Divides large instances to smaller ones, and (recursively) applies the algorithm on the smaller instances.
- Combines the solutions for the subinstances, to produce a solution for the original instance.

Merge Sort

Sets of cardinality greater than one are decomposed into two equal subsets, the algorithm is recursively invoked on the subsets, and the returned ordered subsets are merged to provide a sorted variant of the original set.

The time complexity of the algorithm satisfies the recurrence equation

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

whose solution is *T*(*n*) = *O*(*n* log *n*).

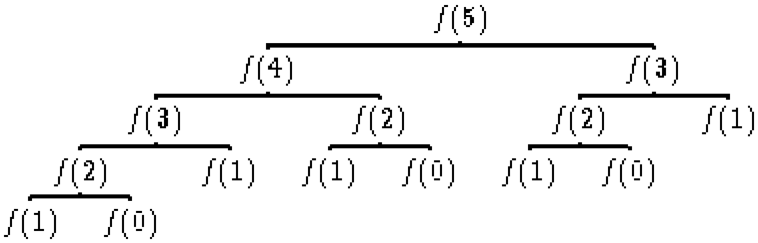
Dynamic Programming Algorithms

The approach assumes a recursive solution for the given problem, with a bottom-up evaluation of the solution. The subsolutions are recorded (in tables) for reuse.

21.1 Fibonnaci Numbers

$$f(i) = \begin{cases} f(i-1) + f(i-2) & \text{if } i > 1 \\ 1 & \text{otherwise} \end{cases}$$

A top-down approach of computing, say, *f*(5) is inefficient do to repeated subcomputations.

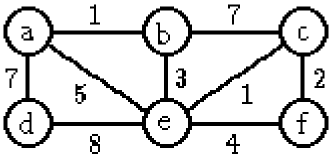


A bottom-up approach computes *f*(0), *f*(1), *f*(2), *f*(3), *f*(4), *f*(5) in the listed order.

All Pairs Shortest Paths (Floyd-Warshall)

A dynamic programming algorithm.

```
FOR k=1 TO n
  FOR i=1 TO n
    FOR j=1 TO n
      c(i,j,k) = min(  c(i,j,k-1),
                       c(i,k,k-1)+c(k,j,k-1)
                    )
```



k = Ø	a b c d e f a 1 ∞7 5 ∞b 7 ∞3 ∞c ∞1 2 d 8 ∞e 4 f
k = {*a*}	a b c d e f a 1 ∞7 5 ∞b 7,b-a-c ∞,b-a-d 3,b-a-e ∞,b-a-f c ∞,c-a-d 1,c-a-e 2,c-a-f d 8,d-a-e ∞,d-a-f e 4,e-a-f f
k = {*a*, *b*}	
k = {*a*, *b*, *c*}	
k = {*a*, *b*, *c*, *d*}	
k = {*a*, *b*, *c*, *d*, *e*}	
k = {*a*, *b*, *c*, *d*, *e*, *f*}	