EnggTree.com

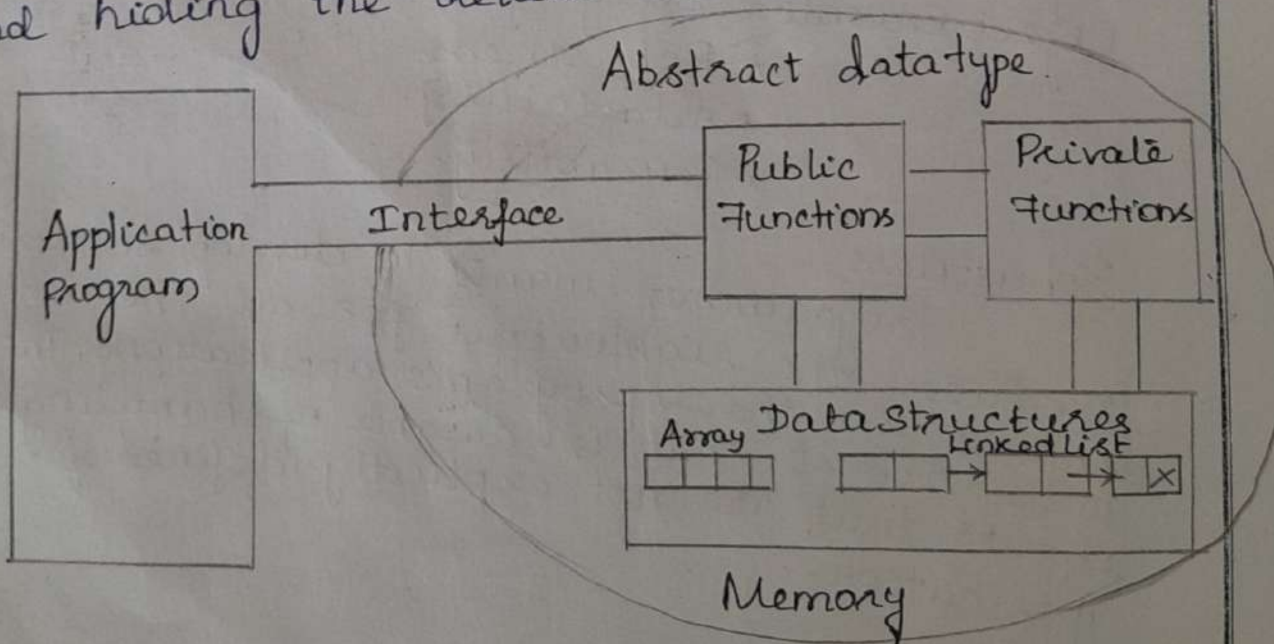
Unit: I
Abstract Data Types

Abstract Datatypes (ADTs) - ADT and classes - introduction to OOP - Classes in python - Inheritance - namespaces - Shallow and deep copying - Introduction to analysis of algorithms - asymptotic notations - Recursion - analyzing recursive algorithms.

Abstract Datatypes.

Abstract Datatype (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and set of operations.

The definition of ADT only mentions what operations are to be performed, but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.



Downloaded from EnggTree.com

EnggTree.com

ADT's and classes.

Classes are the python representation for "Abstract Datatypes". An ADT involves both data and operations on that data.

When we define a class, no memory is allocated, but when we instantiate (an object is created) memory is allocated. An object is an instance of a class, with its own copy of any non_static variables.

Introduction to OOP

Object Oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. The main actors in the Object Oriented paradigm are called objects. Each object is an instance of a class. The class definition typically specifies instance variables, also known as data members, that the object contains, as well as the methods also known as member functions, that the object can execute.

Object Oriented Design goals.

* Robustness
* Adaptability
* Reusability.

Robustness

Programmer wants to develop software that produces the right output for all the anticipated inputs in the program's applications. The software to be robust that is capable of handling unexpected inputs that are not explicitly defined for its application.

Downloaded from EnggTree.com

EnggTree.com

Adaptability.

Modern software applications, such as web browsers and Internet search engines, typically involve large programs that are used for many years. Therefore software needs to be able to evolve over time in response to changing conditions in its enviroment. This feature is called adaptability or evolvability.

Resuability.

The software is reusable, that is the same code should be usable as a component of different systems in various applications.

Object Oriented Design Principles.

* Modularity
* Abstraction
* Encapsulation

Modularity.

Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Modularity refers to an organizing principle in which different components of a software system are divided into separate functional units. In python we have already seen that a module is a collection of closely related functions and classes that are defined together in a single file of source code. (eg) Math, random, date ...

The structure imposed by modularity helps software reusability. This is particularly relevant in a study of datastructures, which can typically be designed with sufficient abstraction and generality to be reused in many applications.

Downloaded from EnggTree.com

EnggTree.com

Abstraction.

The process of poviding only the essentials and hiding the dethils is known as abstraction. Applying the abstraction paradigm to the design of datastructures gives rise to abstract datatype (ADT). An ADT is a mathematical model of a datastructure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations.

An ADT specifies what each operation does, but not how it does it. The collective set of behaviors supported by an ADT as its public interface. python supports ADT using a mechanism known as an abstract base class (ABC). An ABC cannot be instantiated, but it defines one or more common methods that all implementations of the abstraction must have. An ABC is realized by one or more concrete classes that inherit from the abstract base class while providing implementations for those method declared by the ABC. Python's abc module provide formal support for ABC although we omit such declaration for simplicity.

Encapsulation.

Encapsulation describes the concept of binding data and methods within a single unit. A class is an example of encapsulation as it binds all the datamembers and methods into a single unit.

Downloaded from EnggTree.com

EnggTree.com

classes in python.

Class creates a userdefined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

* Classes are created by keyword class.
* Attributes are the variables that belong to a class
* Attributes are always public and can be accessed using the dot(.) operator

class definition syntax.

```
Class classname:
    # constructor
    # Member fun1
    # Member fun2
    :
    # Member fun n
Object creation
Accessing member function.
```

class objects.

An object is an instance of a class.

An Object consist of

State: It is represented by the attributes of an Object.

Behavior: It is represented by the methods of an Object.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

Downloaded from EnggTree.com

EnggTree.com

Eg
```
class student:
    name="xxx"  #attribute1
    dept="CSE"  #attribute2
    def display(self): #method
        print("studentname:", self.name)
        print("Department:", self.dept)
Stud=student() #Object instantiation
print(stud.name)
Stud.display()
```

O/P
xxx
studentname: xxx
Department: CSE

Self keyword.

Class methods must have an extra parameter in the method definition. we donot give a value for this parameter. Self represents the instance of the class. By using the "Self" keyword we can access the attributes and methods of the class in python.

__init__ method.

Constructors are used to initializing the object's state. The __init__ method is used to create constructors in python. Constructors also contains a collection of statements that are executed at the time of object creation. It accepts the self keyword as a first argument which allows accessing the attributes or methods of the class.

Constructors can be of two types
* parameterized Constructor
* Non-parameterized constructor.

Parameterized constructor → we can pass any number of arguments at the time of creating the class object, depending upon the __init__() definition.

Downloaded from EnggTree.com

eg)
```
class employee:
    def __init__(self,ename,edept):
        self.ename = ename
        self.edept = edept
    def display(self):
        print("Empname", self.ename)
        print("Empdept", self.edept)

emp1= employee("Arun", "HR")
emp2= employee("Meena", "Production")
emp1.display()
emp2.display()
```

O/P

```
Empname Arun
Empdept HR
Empname Meena
Empdept Production.
```

Non Parameterized constructor.

The non parameterized constructor uses when we donot want to manipulate the value of the constructor that has only self as an argument.

```
class employee:
    def __init__(self):
        print("Non parameterized constructor")
    def display(self,dept):
        print("Emp dept", self.dept)

emp = employee()
emp.display("Testing")
```

O/P

```
Non parameterized constructor.
Emp dept Testing.
```

Downloaded from EnggTree.com

```
class CreditCard:
    def __init__(self, customer, bank, acnt, limit):
        self._customer = customer
        self._bank = bank
        self._account = acnt
        self._limit = limit
        self._balance = 0
    def get_customer(self):
        return self._customer
    def get_bank(self):
        return self._bank
    def get_account(self):
        return self._account
    def get_limit(self):
        return self._limit
    def get_balance(self):
        return self._balance
    def charge(self, price):
        if price + self._balance>self._limit: # if charge would exceed limit,
            return False # cannot accept charge
        else:
            self._balance += price
        return True
    def make_payment(self, amount):
        self._balance-= amount
if __name__ =='__main__':
    lst = []
    lst.append(CreditCard('abc','SBI','56531', 2500))
    lst.append(CreditCard( 'xyz' ,' IOB' ,'3485', 3500) )
    lst.append(CreditCard( 'aaa' , 'Axis' ,'5391', 5000) )
    for val in range(1, 17):
        lst[0].charge(val)
        lst[1].charge(2*val)
        lst[2].charge(3*val)
    for c in range(3):
        print( 'Customer = ', lst[c].get_customer( ))
        print( 'Bank = ', lst[c].get_bank( ))
        print(' Account = ', lst[c].get_account( ))
        print( 'Limit =' , lst[c].get_limit( ))
        print( 'Balance = ', lst[c].get_balance( ))
        while lst[c].get_balance( ) > 100:
            lst[c].make_payment(100)
        print( 'New balance =' , lst[c].get_balance( ))
```

Output:

```
Customer = abc
Bank = SBI
Account = 56531
Limit = 2500
Balance = 136
New balance = 36

Customer = xyz
Bank = IOB
Account = 3485
Limit = 3500
Balance = 272
New balance = 172
New balance = 72

Customer = aaa
Bank = Axis
Account = 5391
Limit = 5000
Balance = 408
New balance = 308
New balance = 208
New balance = 108
New balance = 8
```

Downloaded from EnggTree.com

EnggTree.com

Operator Overloading.

Operator overloading means giving extended meaning beyond their predefined operations. For example "+" operator is used to adding numbers as well as join strings and merging two lists.

Python provided some special functions to perform operator overloading, which is automatically invoked when it is associated with that operator. when the user uses the "+" operator, the function __add__ will automatically the "+" operator will be defined in the command where the "+" operator will be defined

+ operator (eg) a+b a=5, b=10	Method name __add__	* right hand operand as a parameter * returns the result of the expression.	a+b is converted to a method call on object 'a' a.__add__(b)

(eg)
```
class addition:
    def __init__(Self, x, y):
        Self.x = x
        self.y = y
    def __add__(Self, a):
        return self.x + a.x, self.y + a.y
val1 = addition(10, 20)
val2 = addition (50, 60)
val3 = val1 + val2
print (val3)
Print ("using special methods")
print(val1.__add__(val2))
```

O/P

```
(60, 80)
Using special methods
(60, 80)
```

Downloaded from EnggTree.com

EnggTree.com

Overloaded Operations, implemented with Python's special methods.

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	
abs(a)	a.__abs__()	
a < b	a.__lt__(b)	
a <= b	a.__le__(b)	
a > b	a.__gt__(b)	
a >= b	a.__ge__(b)	
a == b	a.__eq__(b)	
a != b	a.__ne__(b)	
v in a	a.__contains__(v)	
a[k]	a.__getitem__(k)	
a[k] = v	a.__setitem__(k,v)	
del a[k]	a.__delitem__(k)	
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)	
len(a)	a.__len__()	
hash(a)	a.__hash__()	
iter(a)	a.__iter__()	
next(a)	a.__next__()	
bool(a)	a.__bool__()	
float(a)	a.__float__()	
int(a)	a.__int__()	
repr(a)	a.__repr__()	
reversed(a)	a.__reversed__()	
str(a)	a.__str__()	

Downloaded from EnggTree.com

EnggTree.com

Multidimensional Vector Class

Vector class representing the coordinates of a Vector in a multidimensional space. In a threedimensional space we represent a vector with coordinates {5,-2,3}. List cannot be used with vector. Since + operator will not add the vector elements of the list (eg) [5,-2,3]+[1,4,2]=[5,-2,3,1,4,2]

```
class Vector:
  def __init__(self, d):
    self. coords = [0]*d
  def __len__ (self):
    return len(self. coords)
  def __getitem__ (self, j):
    return self. coords[j]
  def __setitem__ (self, j, val):
    self. coords[j] = val
  def __add__ (self, other):
    if len(self) != len(other):
      raise ValueError('dimensions must agree' )
    result = Vector(len(self))
    for j in range(len(self)):
      result[j] = self[j] + other[j]
    return result
  def __eq__ (self, other):
    return self. coords == other. coords
  def __ne__ (self, other):
    return not self == other
  def __str__ (self):
    return '<' + str(self. coords)[1:-1] + '>'
v = Vector(5) # construct five-dimensional <0, 0, 0, 0, 0>
v[1] = 23 # <0, 23, 0, 0, 0> (based on use of setitem )
v[-1] = 45 # <0, 23, 0, 0, 45> (also via setitem )
print(v[4]) # print 45 (via getitem )
u=v+v # <0, 46, 0, 0, 90> (via add )
print(u) # print <0, 46, 0, 0, 90>
total = 0
for entry in v: # implicit iteration via len and getitem
  total += entry
print(total)
u = v + [5, 3, 10, -2, 1]
print(u)
```

output

```
45
<0, 46, 0, 0, 90>
68
<5, 26, 10, -2, 46>
```

vector u={5,-2,3}+{1,4,2}={6,2,5}. so vectors can't be implemented with list. Vector class that provides a better abstraction for the notion of a geometric vector

Downloaded from EnggTree.com

EnggTree.com

Inheritance.

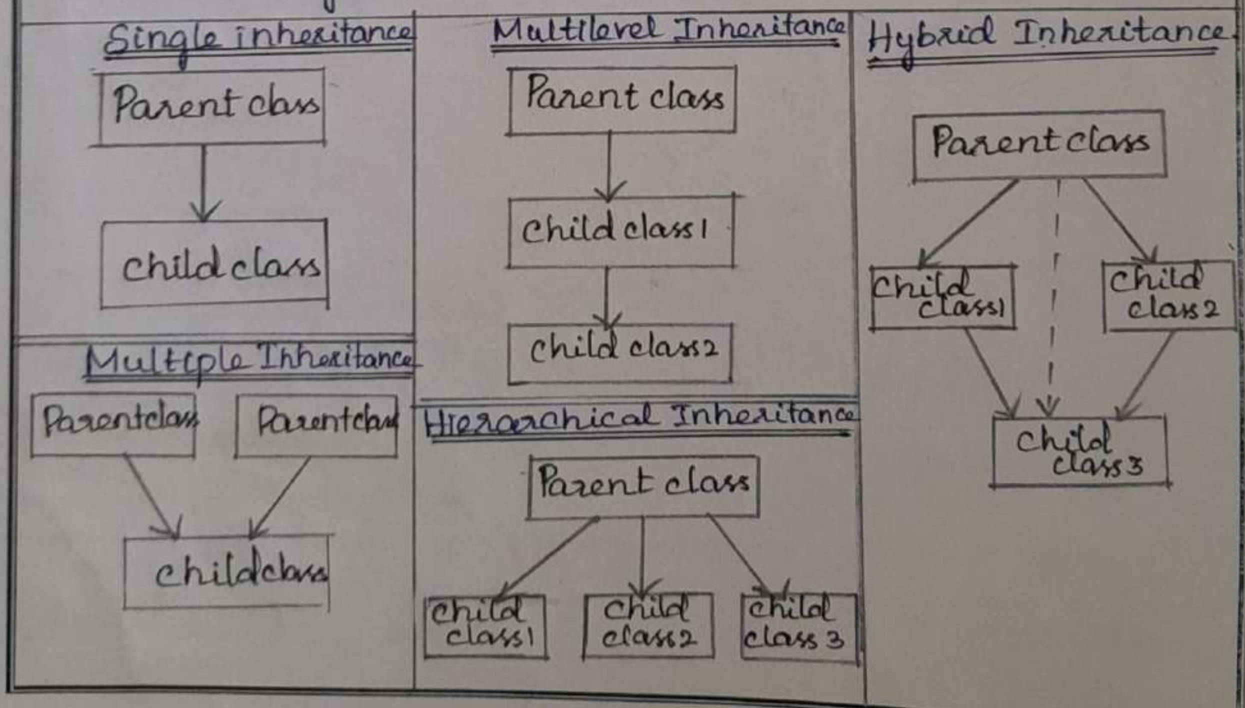
In OOP inheritance is an important aspect. The main purpose of inheritance is the reusability of code. The process of inheriting the properties of the parent class into a child class is called inheritance.

In Object Oriented terminology the existing class is called as the base class, parent class, or super class, while the newly defined class is known as the subclass or child class.

Types of Inheritance.

In python based upon the number of child and parent classes involved, there are five types of inheritance.

* Single Inheritance
* Multiple Inheritance
* Multilevel Inheritance
* Hierarchical Inheritance
* Hybrid Inheritance.



Downloaded from EnggTree.com

Single Inheritance

In single Inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class

Multiple Inheritance.

In Multiple Inheritance, one child class can inherit from multiple parent classes. one child class & multiple parent class

Multilevel Inheritance.

In multilevel inheritance a class inherits from a child class or derived class.

Hierarchical Inheritance.

In hierarchical inheritance more than one child is derived from a single parent class.

Hybrid inheritance.

It is the combination of different inheritance

Super() Function.

When a class inherits all properties and behavior from the parent class is called inheritance

In child class, we can refer to parent class by using the super() function, The super function returns a temporary objects of the parent class that allows us to call a parent class method inside a child class method.

Downloaded from EnggTree.com

EnggTree.com

Extending the credit card class (Single Inheritance)

A subclass has been created for the creditcard class. The newclass will differ from the original Predator creditcard in two ways.

* If an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged.
* To access a monthly interest charge on the outstanding balance, based upon a Annual percentage rate (APR) specified as a constructor parameter.

The mechanism for calling the inherited constructor relies on the syntax super()

super().__init__(customer, bank, acnt, limit)

Code:

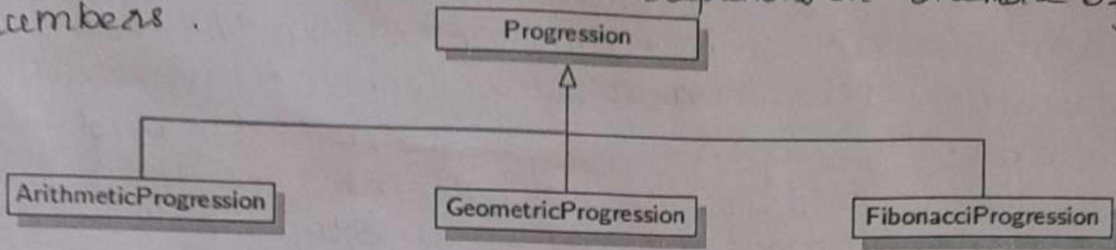
```
class CreditCard:
  def __init__(self, customer, bank, acnt, limit):
    self. customer = customer
    self. bank = bank
    self. account = acnt
    self. limit = limit
    self. balance = 10000
  def get_details(self):
    print(self.customer)
    print(self.account)
    print(self.bank)
    print(self.limit)
  def transact(self,price):
    if price>self.balance or self.balance<self.limit:
        print("Balance not sufficient to make payment")
    else:
       self.balance=self.balance-price
    print("Payment done !!")
    print(self.balance)
  def charge(self,price):
    if price+self.balance>self.limit:
      return False
    else:
      self.balance=self.balance+price
    return True
class PredatoryCreditCard(CreditCard):
  def __init__(self, customer, bank, acnt, limit, apr):
    super().__init__(customer, bank, acnt, limit)
    self. apr = apr
  def charge(self, price):
    success = super( ).charge(price)
    if not success:
      self. balance += 5
      return success
  def process_month(self):
      if self. balance > 0:
        monthlyfactor = pow(1 + self. apr, 1/12)
        self. balance *= monthlyfactor
        print(self.balance)
cc = PredatoryCreditCard( "Kumar", "IOB Bank" , 5391,2000,122)
cc.get_details()
cc.transact(100)
cc.charge(100)
cc.process_month()
```

o/p

Kumar
5391
IOB Bank
2000
Payment done
9900

Downloaded from EnggTree.com

Hierarchy of Numeric Progressions A numeric progression is a sequence of numbers, where each number depends on onemore of the previous numbers.




```
class Progression:
   def __init__(self,start=0):
      self.current =start
   def advance(self):
      self.current+= 1
   def __next__(self):
      if self.current is None: # our convention to end a progression
         raise StopIteration( )
      else:
         answer =self.current # record current value to return
      self.advance( ) # advance to prepare for next time
      return answer # return the answer
   def __iter__(self):
      return self
   def print_progression(self,n):
      print(' '.join(str(next(self)) for j in range(n)))
class ArithmeticProgression(Progression): # inherit from Progression
   def __init__(self,increment=1,start=0):
      super( ).__init__(start) # initialize base class
      self.increment =increment
   def advance(self): # override inherited version
      self.current += self.increment
class GeometricProgression(Progression): # inherit from Progression
   def __init__ (self, base=2, start=1):
      super( ).__init__(start)
      self.base = base
   def advance(self): # override inherited version
      self.current*= self.base
class FibonacciProgression(Progression):
   def __init__ (self,first=0,second=1):
      super( ).__init__(first) # start progression at first
      self.prev = second-first # fictitious value preceding the first
   def advance(self):
      self.prev,self.current = self.current,self.prev+self.current
if __name__ == '__main__':
   print( 'Default progression:' )
   Progression().print_progression(10)
   print( 'Arithmetic progression with increment 5:' )
   ArithmeticProgression(5).print_progression(10)
   print( 'Arithmetic progression with increment 5 and start 2: ')
   ArithmeticProgression(5, 2).print_progression(10)
   print( 'Geometric progression with default base: ')
   GeometricProgression( ).print_progression(10)
   print( 'Geometric progression with base 3:' )
   GeometricProgression(3).print_progression(10)
   print( 'Fibonacci progression with default start values:' )
   FibonacciProgression( ).print_progression(10)
   print( 'Fibonacci progression with start values 4 and 6: ')
   FibonacciProgression(4, 6).print_progression(10)
```

O/P

```
Default progression
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5
0 5 10 15 20 25 30 35 40 45
Arithmetic progression with increment 5 and start 2
2 7 12 17 22 27 32 37 42 47
Geometric progression with base 2
1 2 4 8 16 32 64 128 256 512
Fibonacci progression with default start values
0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6
4 6 10 16 26 42 68 110 178 288
Geometric progression with base 3
1 3 9 27 81 243 729 2187 6561 19683
```

Downloaded from EnggTree.com

EnggTree.com

Namespaces and Object-Orientation.

A namespace is an abstraction that manages all of the identifiers that are defined in a particular scope, mapping each name to its associated value.

Instance Namespaces.

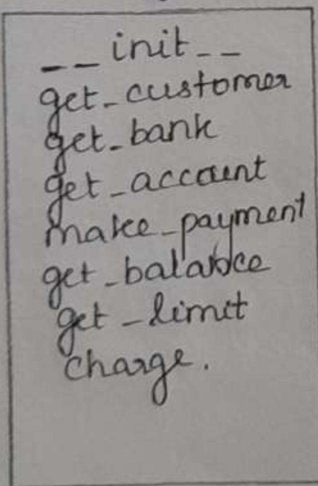
The instance namespace, which manages attributes specific to an individual object.

Eg: Credit card class maintains a distinct balance, a distinct account number, a credit limit and so on...

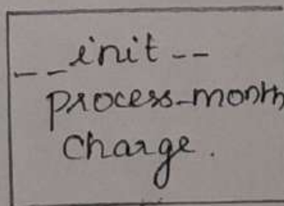
Class Namespace.

Class namespace is used to manage members that are to be shared by all instances of a class, or used without reference to any particular instance.

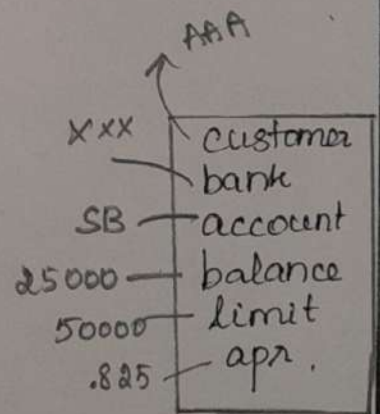
class namespace containing methods of the creditcard class, another class namespace with methods of the predatory creditcard class and finally a single instance namespace for a sample instance of the predatory creditcard class.



The class namespace for credit class.



The class namespace for predatorycreditcard.



The instance namespace for a predatory creditcard object.

Downloaded from EnggTree.com

EnggTree.com

Shallow and Deep Copying

A shallow copy is a copy of an object that stores the reference of the original elements. It creates the new collection object and then occupying it with reference to the child objects found in the original.

It makes copies of the nested objects reference and doesn't create a copy of the reference and doesn't create a copy of the nested objects. So if we make any changes to the copy of the object will reflect in the original object. We use the copy() function to implement it.

eg)

```
import copy
lst1 = [10,20,30]
lst2 = copy.copy(lst1)
print("Original elements before shallow copying")
print(lst1)
lst2[1] = 100
print("After shallow copying")
print(lst2)
print(lst1)
```

O/P

```
Original Elements before shallow copying
[10, 20, 30]
After shallow copying
[10,100,30]
[10, 20, 30]
```

Deep copy in python.

A deep copy is a process where we create a new object and add copy elements recursively. We will use the deecopy() method which present in copy module. The independent copy is created of original object and its entire object.

```
import copy
x = [[10,20,30],['a','b','c'],['x','y','z']]
z = copy.deepcopy(x)
print(x,z)
z[0][1] = 'a'
print(z)
```

O/P

```
[[10,20,30],
 ['a','b','c'],
 ['x','y','z']]
[[10,20,30],['a','b','c'],
 ['x','y','z']]
[[10,'a',30],['a','b','c']
 ['x','y','z']]
```

Downloaded from EnggTree.com

EnggTree.com

Introduction to Analysis of Algorithms.

Data structure is a systematic way of organizing and accessing data, and an algorithm is a step-by step procedure for performing some task in a finite amount of time.

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the time spent during each execution. This operation is called Experimental analysis.

Limitations of Experimental Analysis

* Experimental running times of two algorithm are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
* Experimental can be done only on a limited set of test inputs.
* An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Moving Beyond Experimental Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithm that

* Allows us to evaluate the relative efficiency of any two algorithms
* Is performed by studying a high-level description of the Algorithm without need for implementation
* Takes into account all possible inputs.

Downloaded from EnggTree.com

EnggTree.com

Counting Primitive operations

To analyse the running time of an algorithm without performing experiments, we perform an analysis directly on a high level description of the algorithm.

Primitive operations.

1. Assigning an identifier to an object
2. Determining the object associated with an identifier.
3. Performing an arithmetic operation.
4. Comparing two numbers.
5. Accessing a single element of a python list by index
6. calling a function
7. Returning from a function.

Primitive operations may be translated to a small number of instructions. To calculate the execute time of each primitive operation to count how many primitive operations are executed, and use this number t as a measure of the running time of the algorithm.

This operation count will correlate to an actual running time in a specific computer. Thus the number t of primitive operations of an algorithm performs will be proportional to the actual running time of that algorithm.

Measuring operations as a function of input size.

To capture the order of growth of an algorithm's running time, we will associate with each algorithm, a function f(n) that characterizes the number of primitive operations that are performed as a function of the input size n.

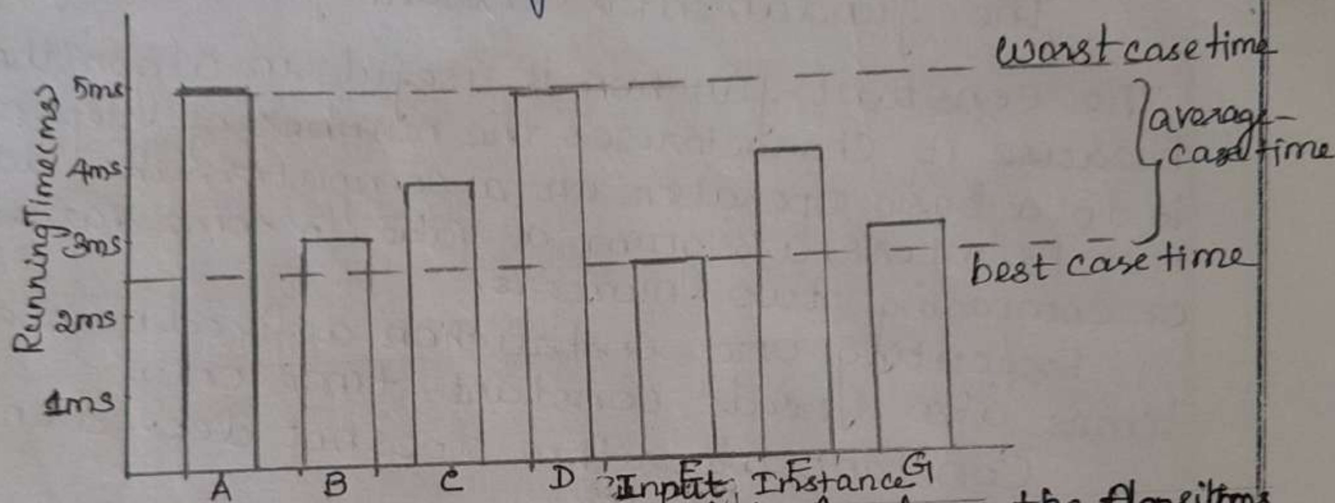
Focusing on the worst-case input.

An algorithm may run faster on some inputs than it does on others of the same size.

Downloaded from EnggTree.com

EnggTree.com

The running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. Average case analysis is quite challenging. Worst case analysis is much easier than average-case analysis.



Functions used to Analyse the Algorithms

Analysis of algorithm

Analyzing the efficiency of a program involves characterizing the running time and space usage of Algorithms and data structure operations.

Running Time

Algorithms transform input objects into output objects. The running time of an algorithm or a datastructure method typically grows with the input size, although it may also vary for different inputs of the same size.

We characterize the algorithm's running time as a function f(n) of the input size n.

Downloaded from EnggTree.com

EnggTree.com

common function used in analysis.

1. The constant function. $f(n) = c$

For any argument n, the constant function f(n) assigns the value c. It doesn't matter what the input size n is, f(n) will always be equal to the constant value c

The fundamental constant function is $f(n) = 1$

The constant function is useful in algorithm analysis because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

Executing one instruction a fixed number of times also needs constant time only.

Constant algorithm does not depend on the input size.

2. The Logarithm Function.

The general form of a logarithm function is $f(n) = \log_b n$ for some constant $b > 1$

This function is defined as follows

$$x = \log_b n, \text{ if and only if } b^x = n$$

The value b is known as the base of the logarithm.

Computing the algorithm function for any integer n is not always easy, but we can easily compute the smallest integer greater than or equal to $\log_b n$, for this number is equal to the number of times we can divide n by b until we get a number less than or equal to 1

(eg) $\log_3 27$ is 3 since $27/3/3/3 = 1$

$\log_2 12$ is 4 since $12/2/2/2/2 = 0.75 \approx 1$

Logarithm function gets slightly slower as n grows. whenever n doubles, the running time increases by a constant.

Downloaded from EnggTree.com

EnggTree.com

3. **The Linear Function**

Given an input value n, the linear function f assigns the value n itself.

$$f(n) = n$$

This function arises in an algorithm analysis any time we do a single basic operation for each of n elements. For example, comparing a number x to each element of an array of size n will require n comparisions.

eg) Print the elements of an array of size n

4. **The N-log-N function**

$$f(n) = n \log n$$

This function grows a little faster than the linear function and a lot slower than the quadratic function (n^2). If we can improve the running time of solving some problem from quadratic to n log n then we have an algorithm that runs must faster in general.

eg) Merge sort.

5. **The Quadratic Function.**

$$f(n) = n^2$$

The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops. where the inner loop performs a linear number of operations and the outer loop is performed a linear no. of. times. The algorithm performs $n * n = n^2$ operations.

The quadratic function can also be used in the context of nexted loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations and so on.

No. of. Operations $1+2+3+ \ldots + (n-1)+n$.

Downloaded from EnggTree.com

EnggTree.com

For any integer $n >= 1$ we have $1+2+3+ \cdots +(n-1)+n = n*(n+1)/2$.

Quadratic algorithms are practical for relatively small problems (whenever n doubles, the running time increases fourfold.

eg) some manipulations of the n by n array.

6. The cubic function and other polynomials

The cubic function $f(n) = n^3$. This function appears less frequently in the context of the algorithm analysis than the constant, linear and quadratic functions. It is used only for small problems. whenever n doubles, the running time increases.

eg: n by n matrix multiplication.

A polynomial function is a function of the form $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$

$a_0, a_1, a_2 \ldots a_n$ are constants

$a_n != 0$.

Integer n, which indicates the highest power in the polynomial, is called the degree of the polynomial.

7. The Exponential Function.

$$f(n) = b^n$$

In this function b is a positive constant, called the base, and the argument n is the exponent. In algorithm analysis, the most common base for the exponential function is $b=2$.

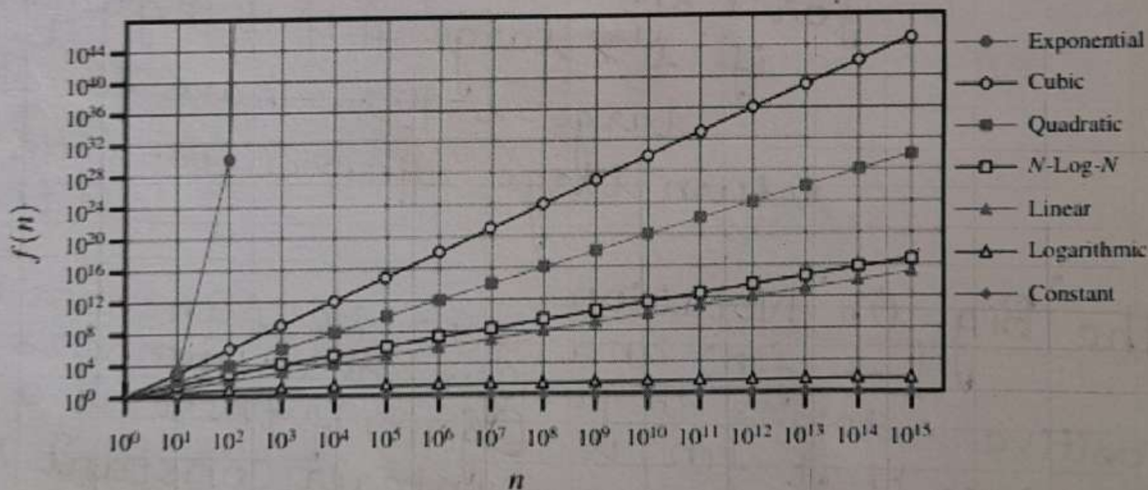
If we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2^n. Exponential alg is usually not appropriate for practical use.

Downloaded from EnggTree.com

Comparing Growth Rates.

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function and we would like our algorithms to run in linear or n-logn time. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs.

Constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Downloaded from EnggTree.com

EnggTree.com

Asymptotic Analysis

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n taking a bigpicture approach. The running time of an algorithm grows proportionally to n

We characterize the running times of algorithms by using functions that map the size of the input n, to values that correspond to the main factor that determines the growth rate in terms of n.

(eg) To find the largest element in the list

```
def findmax(lst): # Returns the max element
    large = lst[0]  # The initial value to beat
    for i in lst :  # for each value
        if i > large  # if it is greater than the best so far
            large = i  # we have found a new best
    return large  # when loop ends large is the max.
```

The "Big-Oh" Notation.

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c>0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c\,g(n)$, for $n \geq n_0$.

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as $f(n)$ is big-Oh of $g(n)$.

(eg) The function $8n+5$ is $O(n)$

By the Big-Oh definition, we need to find a real constant $c>0$ and an integer constant

Downloaded from EnggTree.com

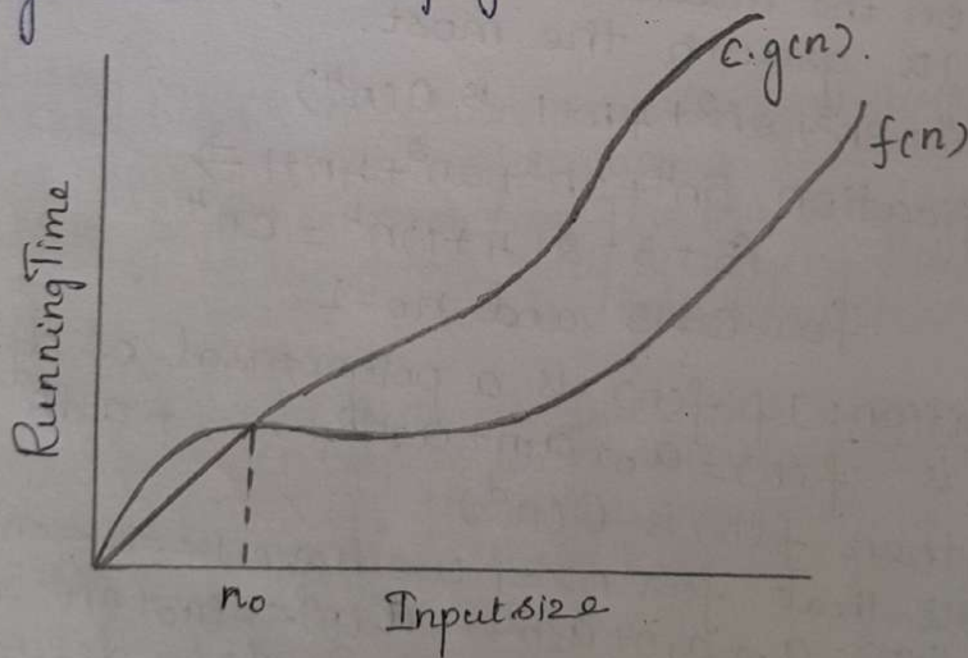
$n_0 \geq 1$ such that $8n+5 \leq cn$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c=9$ and $n_0=5$. Indeed, this is one of infinitely many choices available because there is a trade off between c and n_0

The big-Oh notation allows us to say that a function f(n) is less than or equal to another function g(n) up to a constant factor and in the asymptotic sense as n grows towards infinity.

If f(n) is of the form of An+B, where A and B are constants. It is called a linear function and it is O(n)

The big-Oh notation gives an upper bound on the growth rate of a function.

The statement "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n).



Downloaded from EnggTree.com

EnggTree.com

Characterizing Running Time – Big Oh Notation

The big-oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the size of the problem.

For example, if we are interested in finding a specific element in an array of integers, we should let n denote the number of elements of the array. Using the big-Oh notation, we can write the running time of a sequential search algorithm.

Proposition: The sequential search algorithm, for searching a specific element in an array of n integers, runs in O(n) time.

Some properties of the Big-Oh Notation.

The Big-oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth the most.

eg) $5n^4+3n^3+2n^2+4n+1$ is $O(n^4)$

Justification $5n^4+3n^3+2n^2+4n+1 \Rightarrow$

$(5+3+2+4+1)n^4 = cn^4$

for $c=15$ and $n_0=1$

Proposition: If f(n) is a polynomial of degree d that is $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$ and $a_d > 0$ then f(n) is $O(n^d)$

Note that for $n >= 1$ we have $1 <= n <= n^2 <= \ldots <= n^d$ hence $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots a_d n^d <= (a_0 + a_1 + \ldots a_d) n^d$

$\therefore$ we can show that f(n) is $O(n^d)$ by defining $c = a_0 + a_1 + \ldots a_d$ and $n_0 = 1$.

The highest degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial.

Downloaded from EnggTree.com

EnggTree.com

Characterizing Functions in simplest Terms.

we should use the big-oh notation to characterize a function as closely as possible. eg: $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, It is more accurate to say that $f(n)$ is $O(n^3)$

Rules of using big-Oh

* If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$. we can drop the lower order terms and constant factors.
* Use the smallest/closest possible class of functions, for example "2n is O(n)" instead of "2n is $O(n^2)$"
* use the simplest expression of the class, for eg "3n+5 is O(n)" instead of 3n+5 is O(3n)".

Any algorithm running in $O(n \log n)$ time should be considered efficient. Even $O(n^2)$ may be fast when n is small. But $O(2^n)$ should almost never be considered efficient.

Big-Omega.

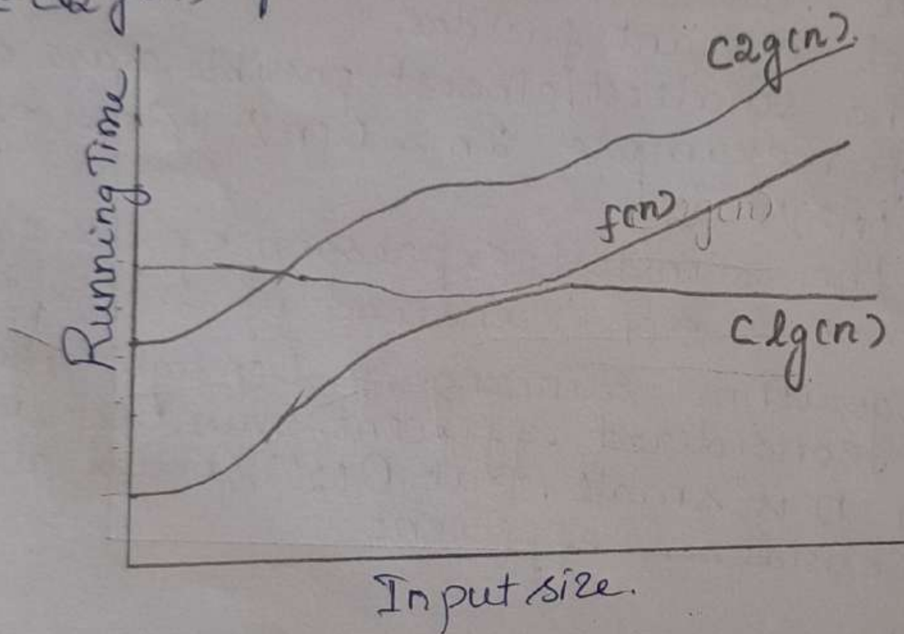
The big-Oh notation provides an asymptotic way of saying that a function is less than or equal to another function. The big-Omega notation provides an asymptotic way of saying that a function grows at a rate that is greater than or equal to that of another.

Let $f(n)$ and $g(n)$ be functions mapping non negative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 >= 1$ such that $f(n) >= c g(n)$ for $n >= n_0$

Downloaded from EnggTree.com

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We say that f(n) is Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n)) that is, it contains real constants $c_1>0$ and $c_2>0$, and integer constant $n_0>=1$ such that $c_1 g(n) <= f(n) <= c_2 g(n)$ for $n >= n_0$.



Downloaded from EnggTree.com

Recursion

Recursion is a technique by which a function makes one or more calls to itself during execution or by which a datastructure relies upon smaller instances of the very same type of structure in representation.

Recursion is an important technique in the study of datastructures and algorithms.

Factorial Function

It is a classic mathematical function that has a natural recursive definition.

English ruler.

Has an recursive pattern that is a simple example of a fractal structure.

Binary search

It is one of the most important computer algorithms. It allows us to efficiently locate a desired value in a data set.

File system.

It has a recursive structure in which directories can be nested arbitrarily deeply within other directories.

Illustrative Examples.

Factorial Function.

The factorial of a positive integer n, denoted n! is defined as the product of the integers from 1 to n. If $n=0$ then n! is defined as 1 by convention. more formally, for any integer $n \geq 0$

$$n! = \begin{cases} 1 & \text{if } n=0 \\ n*(n-1)*(n-2)*\ldots 3*2*1 & \text{if } n \geq 1 \end{cases}$$

Downloaded from EnggTree.com

EnggTree.com

Factorial function is known by equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of permutations of n items.

Eg: a, b, c these three characters can be arranged in 3! = 3x2x1 (ie) abc, acb, bac, bca, cab & cba.

Natural recursive definition for the factorial function 5! = 5x4x3x2*1 (ie) 5*(4x3x2*1) = 5x4!

The recursive definition can be formalized as

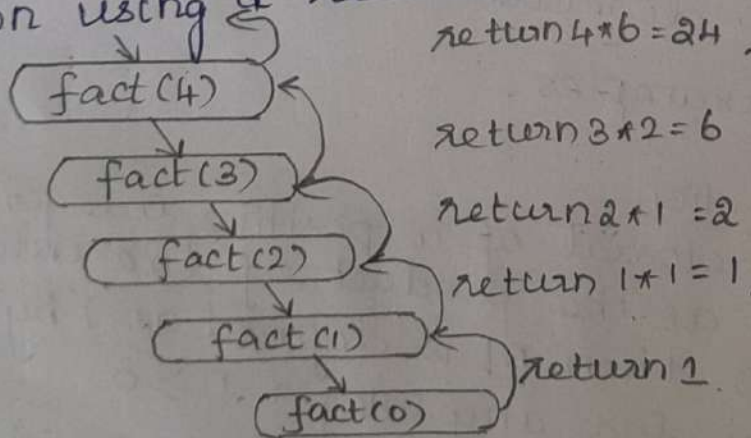
$$n! = \begin{cases} 1 & \text{if } n=0 \\ n*(n-1)! & \text{if } n \geq 1 \end{cases}$$

In this case n=0 is the base case. It also contains one or more recursive cases.

Recursive implementation of the factorial function

```
def fact(n):
    if n==0:
        return 1
    else
        return n*fact(n-1)
```

we illustrate the execution of a recursive function using a recursion trace:



Downloaded from EnggTree.com

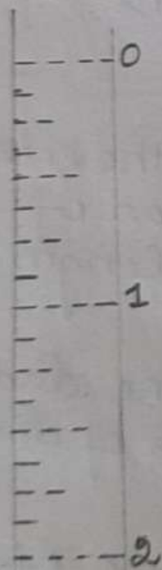
EnggTree.com

In python, each time a function (recursive or otherwise) is called, a structure known as an activation record or frame is created to store information about the progress of that invocation of the function.

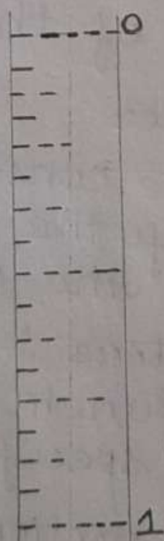
2. Drawing an English Ruler

Complex example of the use of recursion, consider how to draw the markings of a typical English ruler. For each inch, we place a tick with a numeric label. we denote the length of the tick designating a whole inch as the major tick length. Between the marks for whole inches, the ruler contains a series of minor ticks, placed at intervals of 1/2 inch, 1/4 inch and so on. As the size of the interval decreases by half, the tick length decreases by one.

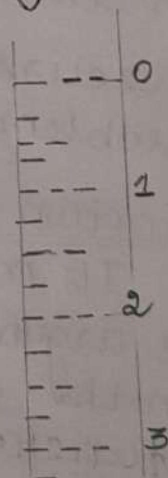
Three sample outputs of an English ruler drawing.



a) A 2 inch ruler with major tick length 4



(b) A 1-inch ruler with major tick length 5



(c) A 3 inch ruler with major tick length 3

Downloaded from EnggTree.com

EnggTree.com

Recursive Approach to Ruler Drawing:

The English ruler pattern is a simple example of a fractal, that is a shape that has a self-recursive structure at various levels of magnification.

Consider the rule with major tick length 5 shown in the above figure (b). Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at 1/2 inch) has length 4.

Observe that the two patterns of ticks above and below this central tick are identical and each has a central tick of length 3.

In general an interval with a central tick length L ≥ 1 is composed of

* An interval with a central tick length L-1
* A single tick of length L
* An interval with a central tick length L-1

The recursive implementation of the draw ruler problem consists of three functions.

Main function draw-ruler

It manages the construction of the entire ruler. Its arguments specify the total number of inches in the ruler and the major tick length.

utility function draw-line

The utility function draw-line, draws a single tick with a specified number of dashes

Recursive draw-interval function.

This function draws the sequence of minor ticks within some interval based upon the length of the intervals central tick.

Base case when L=0 that draws nothing

For L ≥ 1 the first & last steps are performed recursively calling draw_interval(L-1)

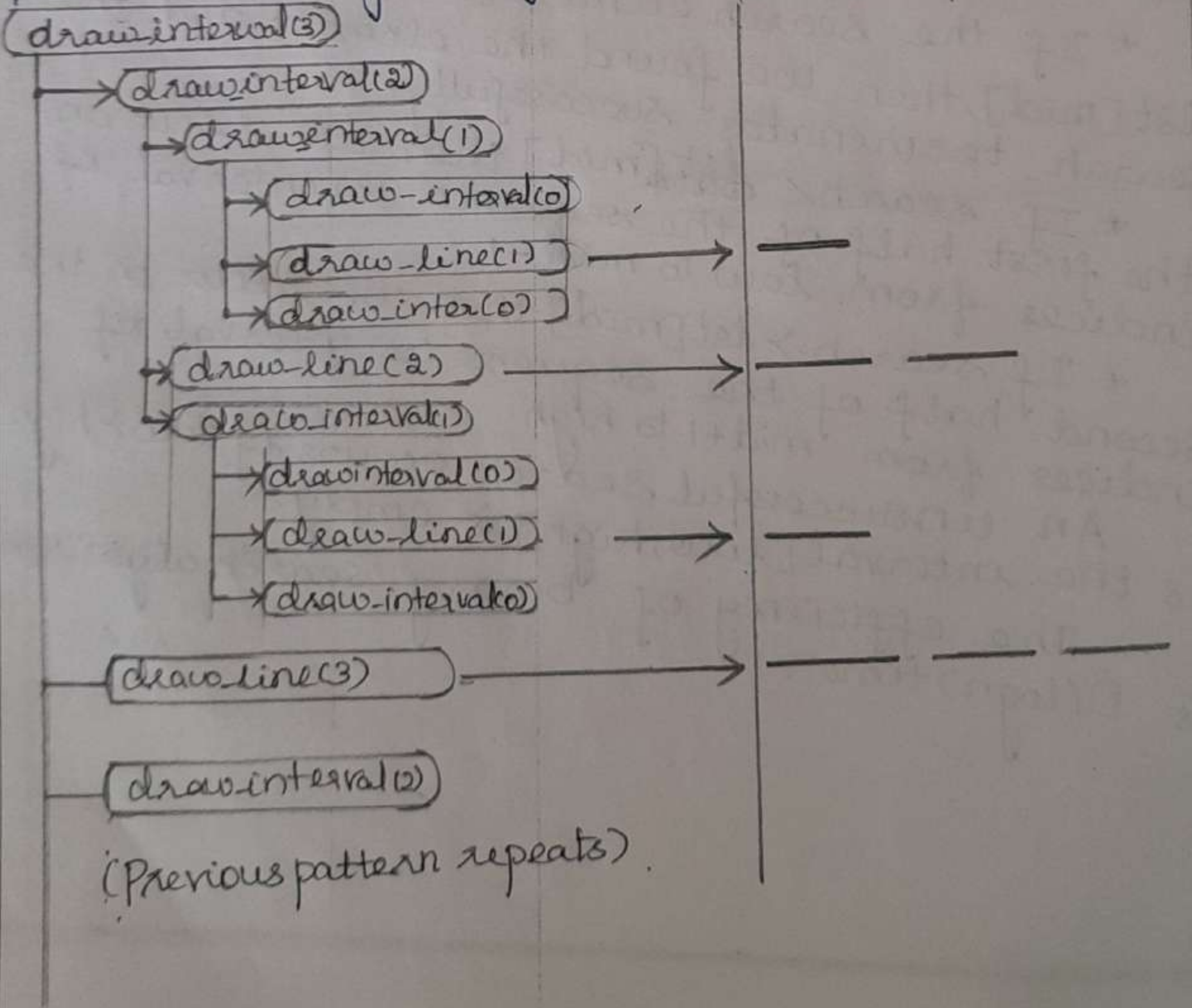
The middle step is performed by calling the function draw-line(L)

Downloaded from EnggTree.com

EnggTree.com

```
def draw_line(tick_length, tick_label=''):
    line = '-' * tick_length
    if tick_label:
        line += ' ' + tick_label
    print(line)
def draw_interval(center_length):
    if center_length > 0:
        draw_interval(center_length - 1)
        draw_line(center_length)
        draw_interval(center_length - 1)
def draw_ruler(num_inches, major_length):
    draw_line(major_length, '0')
    for j in range(1, 1 + num_inches):
        draw_interval(major_length - 1)
        draw_line(major_length, str(j))
```

Ruler Drawing using a Recursion Trace.



Downloaded from EnggTree.com

EnggTree.com

Binary Search

Binary search is used to locate a target value within a sorted sequence of n elements, efficiently.

0	1	2	3	4	5	6	7
3	7	10	13	25	29	32	38

When the sequence is unsorted, the standard approach used in the binary search is to sort the element first then the algorithm maintains two parameters low and high. Initially low=0 and high=n-1. we then compare the target value to the medium element, that is lst[mid] with index

mid = low+high//2

we consider three cases.

* If the search element/target value equals lst[mid], then we found the element and the search terminates successfully.
* If search< lst[mid] then we recur on the first half of the sequence (ie) interval of indices from low to mid-1
* If search > lst[mid] then we recur on the second half of the sequence (ie) interval of indices from mid+1 to high.

An unsuccessful search occurs if low > high as the interval [low, high] is empty.

The efficiency of binary search algorithm is O(logn) time.

Downloaded from EnggTree.com

EnggTree.com

Algorithm.

```
def binarysearch(lst, x, low, high):
    if low > high:
        return False
    else:
        mid = (low+high)//2
        if x == lst[mid]:
            return True
        elif x < lst[mid]
            return binarysearch(lst, x, low, mid-1)
        else
            Return binarysearch(lst, x, mid+1, high).
```

search element: 29

0	1	2	3	4	5	6	7
3	7	10	13	25	29	32	38
↑ low			↑ mid				↑ high

mid = 0+7//2
= 3

0	1	2	3	4	5	6	7
3	7	10	13	25	29	32	38
				↑ low	↑		↑ high

mid = 4+7//2
= 5

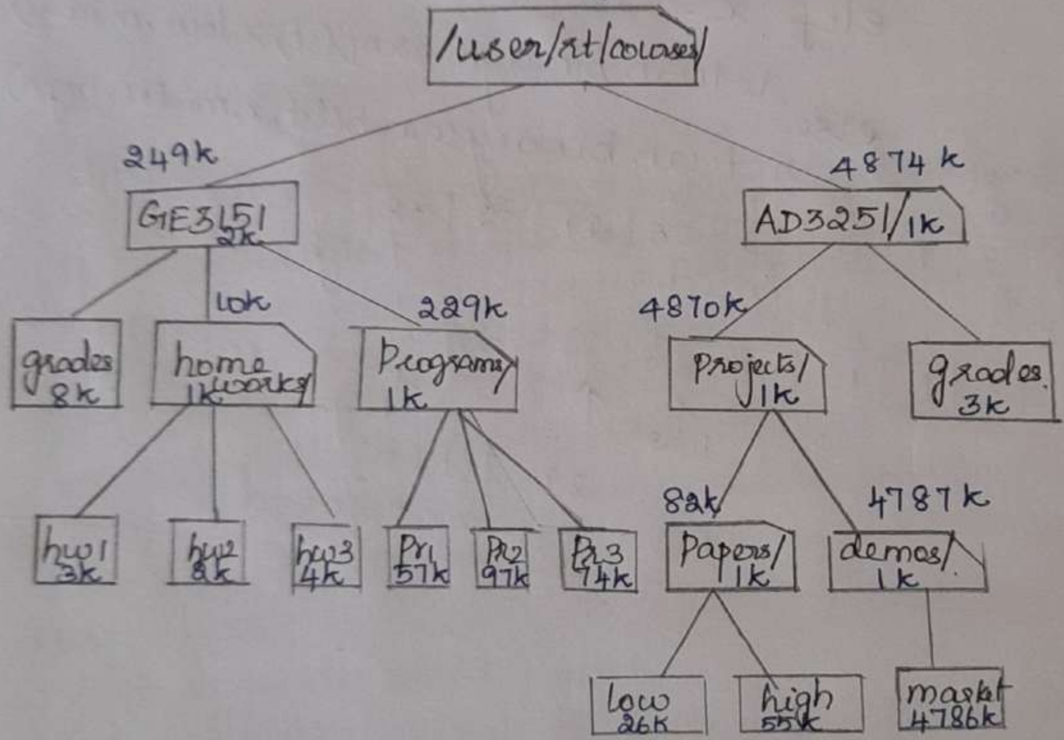
29 = lst[5]
element found.

Downloaded from EnggTree.com

EnggTree.com

File systems.

Modern operating systems define file systems directories in a recursive way. Namely a file system consists of a top-level directory and the contents of this directory consists of files and other directories, which in turn contain files and other directories, and so on. The OS allows directories to be nested arbitrarily deep



The numbered value shows the cumulative disk space used by that directory and all its recursive contents.

The cumulative disk space for an entry can be computed with a simple recursive algorithm. It is equal to the immediate disk space used by the entry plus the sum of the cumulative disk space usage of any entries that are stored directly within the entry.

Downloaded from EnggTree.com

EnggTree.com

Algorithm DiskUsage(path):

Input: A string designating a path to a file-system entry

Output: The cumulative disk space used by that entry and any nested entries.

```
total = size(path)
if path represents a directory then
    for each child entry stored within directory path do
        total = total + DiskUsage(child)
return total
```

Python's OS Module

To provide a python implementation of a recursive algorithm for computing disk usage, we use python's OS module. This is an extensive library, but we will only need the following four functions.

* OS.path.getsize(path)
  Return the immediate disk usage for the file or directory that is identified by the string path. (eg) /user/rt/courses.
* OS.path.isdir(path)
  Return True if entry designated by string path is a directory. False otherwise.
* OS.listdir(path)
  Return a list of strings that are the names of all entries within a directory designated by string path.
  (eg) If the parameter is /user/rt/courses, this returns the list ['GE3151', 'AD3251']
* OS.path.join(path, filename)
  Compose the path string and filename string using an appropriate operating system separator between the two (eg) unix or linux / char \ character for windows.

Downloaded from EnggTree.com

EnggTree.com

Python Implementation.

```
import os
def disk_usage(path):
    total = os.path.getsize(path)
    if os.path.isdir(path):
        for filename in os.listdir(path):
            childpath = os.path.join(path, filename)
            total += disk_usage(childpath)

    print(total, path)
    return total
```

Recursion Trace.

During the execution of the algorithm, exactly one recursive call is made for each entry in the portion of the file system that is considered.

```
8       /user/rt/courses/cs016/grades
3       /user/rt/courses/cs016/homeworks/hw1
2       /user/rt/courses/cs016/homeworks/hw2
4       /user/rt/courses/cs016/homeworks/hw3
10      /user/rt/courses/cs016/homeworks
57      /user/rt/courses/cs016/programs/pr1
97      /user/rt/courses/cs016/programs/pr2
74      /user/rt/courses/cs016/programs/pr3
229     /user/rt/courses/cs016/programs
249     /user/rt/courses/cs016
26      /user/rt/courses/cs252/projects/papers/buylow
55      /user/rt/courses/cs252/projects/papers/sellhigh
82      /user/rt/courses/cs252/projects/papers
4786    /user/rt/courses/cs252/projects/demos/market
4787    /user/rt/courses/cs252/projects/demos
4870    /user/rt/courses/cs252/projects
3       /user/rt/courses/cs252/grades
4874    /user/rt/courses/cs252
5124    /user/rt/courses/
```

Downloaded from EnggTree.com

EnggTree.com

Analyzing Recursive Algorithms.

Mathematical techniques for analyzing the
efficiency of an algorithm, based upon an estimate
of the number of primitive operations that are
executed by the algorithm. we use notations such as
big-oh to summarized the relationship between the
number of operations and the input size for a
problem.

Computing Factorials.

It is relatively easy to analyze the efficiency
of our function for computing factorials. To
compute fact(n), we see that there are n+1 activations,
as the parameter decreases from n in the first call
to n-1 in the second call and so on, until reaching
the base case with parameter 0.

Therefore we conclude that the overall number
of operations for computing fact(n) is O(n), as
there are n+1 activations, each of which accounts
for O(1) operations.

Drawing an English Ruler.

In analyzing the English Ruler application, we
consider the fundamental question of how many
total lines of output are generated by an initial
call to draw-interval(c), where c denotes the center
length. This is a reasonable benchmark for the overall
efficiency of the algorithm as each line of output
is based upon a call to the draw_line utility, and
each recursive call to draw-interval with nonzero
parameter makes exactly one direct call to draw_line

Proposition: For $c \geq 0$, a call to draw_interval(c) results
in precisely $2^c - 1$ lines of output.

The base case of $c=0$, which generates no output,
and $2^0 - 1 = 1 - 1 = 0$. This is the base of the claim.

Downloaded from EnggTree.com

Next we can see that the number of lines printed by draw_interval(c) is one more than twice the number generated by a call to draw_interval(c-1) as once center line is printed between two such recursive calls. By induction, the number of lines is thus $1+2*(2^{c-1}-1)=1+2^c-2=2^c-1$

Performing a Binary Search.

In Binary search, the running time is proportional to the number of recursive calls performed.

Proposition: The binary search algorithm runs in $O(\log n)$ time for a sorted sequence with n elements.

Justification.

In Binary search each recursive call the number of candidate entries still to be searched is given by the value

$$high-low+1$$

Moreover, the number of remaining candidates is reduced by atleast one half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(mid-1)-low+1=\left\lceil\frac{low+high}{2}\right\rceil-low\le\frac{high-low+1}{2}$$

(or)

$$high-(mid+1)+1=high-\left\lfloor\frac{low+high}{2}\right\rfloor\le\frac{high-low+1}{2}$$

Initially, the number of candidates is n, after the first call in a binary search it is atmost n/2 after the second call it is atmost n/4 and so on. In general after the jth call in the binary search, the number of candidate entries remaining is at most $n/2^j$. In worst case the recursive calls stop when there are no more candidate entries.

$$\frac{n}{2^r}<1 \text{ In other words } r>\log n.$$

$$\text{Thus } r=\lfloor\log n\rfloor+1$$

which implies the binary search runs in $O(\log n)$ time.

Downloaded from EnggTree.com

Computing disk space usage.

To compute the overall disk space usage in a specified portion of a file system. To characterize the problem size for our analysis, let n denote the number of file system entries in the portion of the file system that is considered.

To characterize the cumulative time spent for an initial call to the disk-usage function, we must analyze the total number of recursive invocations that are made, as well as the number of operations that are executed within those invocations.

We define the nesting level of each entry such that the entry on which we begin has nesting level 0, entries stored directly within it have nesting level 1, entries stored within those entries have nesting level 2 and so on.

Based on the reasoning we conclude that there are O(n) recursive calls, each of which runs in O(n) time, leading to an overall running time is O(n²)

Downloaded from EnggTree.com

EnggTree.com

AD3251 DATA STRUCTURES DESIGN L T P C 3 0 0 3

COURSE OBJECTIVES:
- To understand the concepts of ADTs
- To design linear data structures – lists, stacks, and queues
- To understand sorting, searching and hashing algorithms
- To apply Tree and Graph structures

UNIT I ABSTRACT DATA TYPES 9

Abstract Data Types (ADTs) – ADTs and classes – introduction to OOP – classes in Python – inheritance – namespaces – shallow and deep copying Introduction to analysis of algorithms – asymptotic notations – recursion – analyzing recursive algorithms

UNIT II LINEAR STRUCTURES 9

List ADT – array-based implementations – linked list implementations – singly linked lists – circularly linked lists – doubly linked lists – applications of lists – Stack ADT – Queue ADT – double ended queues

UNIT III SORTING AND SEARCHING **9**

Bubble sort – selection sort – insertion sort – merge sort – quick sort – linear search – binary search – hashing – hash functions – collision handling – load factors, rehashing, and efficiency

UNIT IV TREE STRUCTURES **9**

Tree ADT – Binary Tree ADT – tree traversals – binary search trees – AVL trees – heaps – multiway search trees

UNIT V GRAPH STRUCTURES **9**

Graph ADT – representations of graph – graph traversals – DAG – topological ordering – shortest paths – minimum spanning trees

TOTAL: 45 HOURS

COURSE OUTCOMES:
At the end of the course, the student should be able to:
· explain abstract data types
· design, implement, and analyse linear data structures, such as lists, queues, and stacks, according to the needs of different applications
· design, implement, and analyse efficient tree structures to meet requirements such as searching, indexing, and sorting
· model problems as graph problems and implement efficient graph algorithms to solve them

TEXT BOOKS:
1. Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, “Data Structures and Algorithms in Python” (An Indian Adaptation), Wiley, 2021.
2. Lee, Kent D., Hubbard, Steve, “Data Structures and Algorithms with Python” Springer Edition 2015.
3. Narasimha Karumanchi, “Data Structures and Algorithmic Thinking with Python” Careermonk, 2015.

REFERENCES:

1. Rance D. Necaise, “Data Structures and Algorithms Using Python”, John Wiley & Sons, 2011.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, “Introduction to Algorithms", Third Edition, PHI Learning, 2010.
3. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, Fourth Edition, Pearson Education, 2014
4. Aho, Hopcroft, and Ullman, “Data Structures and Algorithms”, Pearson Education India, 2002.

Downloaded from EnggTree.com

UNIT: II
Linear Structures

A Stack is a Linear Data structure. Stack is a collection of objects that are inserted and removed according to the Last-in, first-out (LIFO) principle. We can perform the two operations in the stack * PUSH * POP

The Stack Abstract Data Type.

Stacks are the simplest concept in data structures. Formally, a stack is an abstract data type (ADT) such that an instance S supports the following two methods.

S.push(e) : Add element e to the top of Stack S

S.pop() : Remove and return the top element from the stack S, an error occurs if the stack is empty.

Additional Access methods.

S.top() : Return a reference to the top element of Stack S, without removing it; an error occurs if the stack is empty.

S.is_empty() : Return True if stack S does not contain any elements.

len(S) : Return the number of elements in stack S. In python we implement this with the special method __len__.

(eg)

Operation	Return value	Stack cont
S.push(5)	–	[5]
S.push(3)	–	[5,3]
len(S)	2	[5,3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]

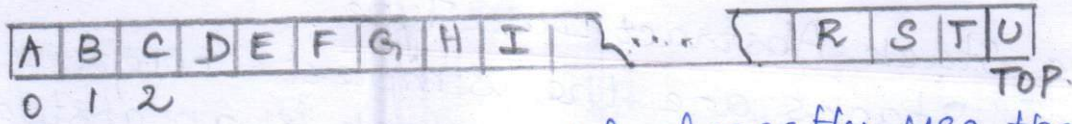
Operation	Return Value	Stack Contents
S.is_empty()	True	[ ]
S.pop()	Error	[ ]
S.push(5)	-	[5]
S.push(10)	-	[5,10]
S.top()	10	[5,10]

Downloaded from EnggTree.com

EnggTree.com

Simple Array- Based Stack Implementation.

Python list can be used as the stack. It uses the append() method to insert element to the list where stack uses the push() method. The list also provides the pop() method to remove the last element, But the list becomes slow as it grows.



A programmer could directly use the list class in place of a formal stack class, lists also include behaviors that would break the abstraction that the stack ADT represents.

we demonstrate how to use a list for internal storage while providing a public interface consistent with a stack.

Adapter Pattern.

The adapter design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface.

Stack Method	Realization with Python list
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[-1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Downloaded from EnggTree.com

Implementing a stack using a python list.

```
Class ArrayStack:
    def __init__(self):
        self.data = []

    def __len(self):
        return len(self.data)

    def is_empty(self):
        return len(self.data)==0

    def push(self,e):
        self.data.append(e)

    def top(self):
        if self.is_empty():
            raise Empty('stack is empty')
        return self.data[-1]

    def pop(self):
        if self.is_empty():
            raise Empty('stack is empty')
        return self.data.pop()
```

Analyzing the performace of Array-Based Stack Implementation:

'n' is the current number of elements in the stack, when an operation causes the list to resize its internal array. The space usage for a stack is O(n).

Operation	Running Time
S.push(e)	O(1)*
S.pop()	O(1)*
S.top()	O(1)
S.is_empty	O(1)
slen(S)	O(1)

Downloaded from EnggTree.com

EnggTree.com

Reversing Data using a stack.

As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence. For example if the values 1,2&3 are pushed onto a stack in that order, they will be popped from the stack in the order 3,2 and then 1.

This idea can be applied in various problems.

(eg) To print lines of a file in reverse order in order to display a data set in decreasing order rather than increasing order.

sol

This can be accomplished by reading each line and pushing it onto a stack, and then writing the lines in the order they are popped.

```
def reverse_file(filename):
    S = ArrayStack()
    f = open(filename)
    for line in f:
        S.push(line.rstrip('\n'))  # re-insert newlines when writing.
    f.close().
    f1 = open("filename",'w')
    while not S.is_empty():
        f1.write(S.pop()+'\n')
    f1.close().
```

Function call.

```
filename = "D:\data1.txt"
reverse_file(filename)
with open(filename) as file:
    for f in file.readlines():
        print(f, end=" ")
```

Input

data1.txt
Stack operations
Reversing Data

Output

data1.txt
Reversing Data
Stack operations.

Downloaded from EnggTree.com

EnggTree.com

Matching Parentheses and HTML Tags.

Matching parentheses and HTML tags are two related applications of stacks, both the problems test the pair of matching delimiters.

Matching Parentheses.

This application we take an example of arithmetic expressions that may contain various pairs of grouping symbols such as

* Parentheses "(" and ")"
* Braces "{" and "}"
* Brackets "[" and "]"

Each opening symbol must match its corresponding closing symbol.

Eg:
The expression [(5+x)-(y+z)]. Each opening symbol must match its corresponding closing symbol.

Correct

( ) (( )) {([( )])}

((( )(( )){([( )])}))

Incorrect

) (( )) {([( )])}

({[ ])}

(

Algorithm - Matching Delimiters.

```
def is_matched(expr):
    lefty = '({['
    righty = ')}]'
    S = ArrayStack()
    for c in expr:
        if c in lefty:
            S.push(c)
        elif c in righty:
            if S.is_empty():
                return False
            if righty.index(c) != lefty.index(S.pop()):
                return False
    return S.is_empty()
```

Function call

```
Print(is_matched({()}[]))
```

Downloaded from EnggTree.com

EnggTree.com

In an HTML document, portions of text are delimited by HTML tags. A simple opening HTML tag has the form <name> and the corresponding closing tag has the form </name>.

Commonly used HTML tags.

- body : document body
- h1 : section header
- center : center justify
- p : paragraph
- ol : numbered (ordered list)
- li : list item.

Algorithm:

```
def is_matched_html(raw):
    S = ArrayStack()
    j = raw.find('<')
    while j != -1
        k = raw.find('>', j+1)
        if k == -1
            return False
        tag = raw[j+1 : k]
        if not tag.startswith('/'):
            S.push(tag)
        else:
            if S.is_empty():
                return False
            if tag [1: ] != S.pop():
                Return False
        j = raw.find('<', k+1)
    return(S.is_empty())
```

Function call.

```
Print(is_matched_html
(open("d:\htmleg.html").read()))
```

Ideally an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags. We make a left to Right pass through the raw string, using index j to track our progress and the find method of the str class to locate the '<' and '>' characters, that define the tags. Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack.

Downloaded from EnggTree.com

EnggTree.com

We assume the input is a sequence of characters, such as '[(5+x) − (y+z)]'. We perform a left-to-right scan of the original sequence, using a stack S to facilitate the matching of grouping symbols. Each time we encounter an opening symbol, we push that symbol onto S, and each time we encounter a closing symbol, we pop a symbol from the stack S, and check that these two symbols form a valid pair. If we reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, there must be an opening delimiter on the stack without a matching symbol.

Matching Tags in a Markup Language.

Matching delimiters is in the validation of markup languages such as HTML or XML. HTML is the standard format for hyperlinked documents on the internet and XML is an extensible markup language used for a variety of structured data sets.

D:\htmleg.html.

```
<body>
<center>
<h1> Python </h1>
</center>
<P> Python is formally an
interpreted programming
language. python is easy
to learn </P>
<ol>
<li> Modes of python</li>
<li> Datatypes of python</li>
</ol>
</body>
```

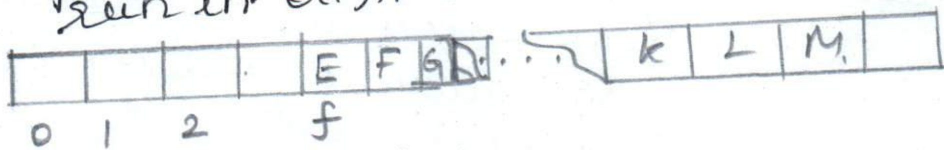
Downloaded from EnggTree.com

 EnggTree.com

Operation	Return Value	first ← last
Q.enqueue(5)	-	[5]
Q.enqueue(8)	-	[5, 8]
len(Q)	2	[5, 8]
Q.dequeue()	5	[8]
Q.is_empty	False	[8]
Q.first()	8	[8]
Q.enqueue(10)	-	[8, 10]
Q.enqueue(5)	-	[8, 10, 5]

Array-Based Queue Implementation.

The Queue ADT we could enqueue element e by calling append(e) to add it to the end of the list, use the syntax pop(0) to intentionally remove the first element from the list when dequeuing. If we call pop(0) it causes the worst case behavior of Θ(n) time. so that we can replace the dequeued entry in the array with a reference to None and maintain an explicit variable f to store the index of the element, that is currently at the front of the queue. such an algorithm for dequeue would run in O(1) time.



Using an Array circularly.

We allow the front of the queue to drift rightward, and we allow the contents of the queue to "wrap around" the end of an underlying array. we assume that our underlying array has fixed length N that is greater than the actual number of elements in the queue. New elements are enqueued toward the end of the current queue. progressing from the front to index N-1 and continuing at index 0 and 1.

Downloaded from EnggTree.com

EnggTree.com

Queues.

A queue is a collection of objects that are inserted and removed according to the first-in-first out (FIFO) principle. That is, elements can be inserted at any time, but only the elements that has been in the queue the longest can be next removed.

(eg) People waiting in line to purchase tickets
Phone calls being routed to customer service center.
FIFO queues are also used by many computing devices * Networked printer. * web server responding to requests.

The Queue Abstract Data Type.

The queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the queue, and element insertion is restricted to the back of the sequence.

The queue ADT supports the two methods.

Q.enqueue(e) : Add element e to the back of queue Q

Q.dequeue() : Remove and return the first element from queue Q; an error occurs if the queue is empty.

Supporting methods.

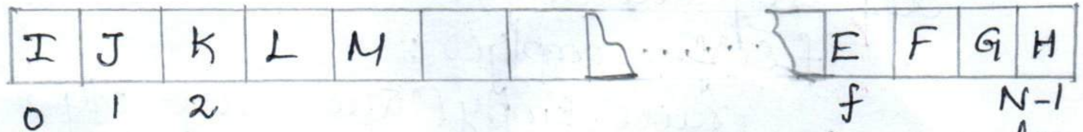
Q.first() : Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.

Q.is_empty() : Return True if queue Q does not contain any elements.

len(Q) : Return the number of elements in queue Q, in python we implement this with the special method __len__

Downloaded from EnggTree.com

EnggTree.com



When we dequeue an element the front index is calculated by f=(f+1) % N, which is computed by taking the remainder after an integral division.
(Eg) If we have a list of length 10, and front index 7, we can advance the front by formally computing (7+1) % 10 which is simply 8, as divided by 10

A python Queue Implementation.

The queue class maintains three instance variables.

data : is a reference to a list instance with a fixed capacity

size : is an integer representing the current number of elements stored in the queue.

front : is an integer that represents the index within data of the first element of the queue.

Array-based implementation of a queue.

```
Class ArrayQueue:
    capacity = 10
    def __init__(Self):
        Self.data = [None] * ArrayQueue.capacity
        Self.size = 0
        Self.front = 0
    def __len__(self)
        return Self.size
    def is_empty(Self):
        return Self.size == 0
    def first (Self):
        if Self.is_empty():
            raise Empty("Queue is empty")
        return self.data[Self.front]
```

Downloaded from EnggTree.com

 EnggTree.com

```
def dequeue(self):
    if self.is_empty():
        raise Empty("Queue is empty")
    answer = self.data[self.front]
    Self.data[self.front] = None
    Self.front = (Self.front + 1) % len(Self.data)
    Self.size -= 1
    return answer
def enqueue(self, e):
    if Self.Size == len(self.data):
        Self.resize(2 * len(self.data))
    avail = (self.front + self.size) % len(Self.data)
    self.data[avail] = e
    self.size += 1
def resize(self, cap):
    Old = self.data
    Self.data = [None] * cap
    walk = Self.front
    for k in range(self.size):
        self.data[k] = old[walk]
        walk = (1 + walk) % len(old)
    Self.front = 0
```

Adding and Removing Elements.

The goal of the enqueue method is to add a new element to the back of the queue. we need to determine the proper index at which to place the new element.

we compute the location of the next opening based on the formula.

avail = (self.front + self.size) % len(self.data)

For example, consider a queue with capacity 10, current size 3 and first element at index 5. The three elements of such a queue are stored at indices 5, 6, & 7. The new element should be placed at index (front + size) = 8. In a case with wrap-around, the use of the modular arithmetic achieves the desired circular semantics.

Downloaded from EnggTree.com

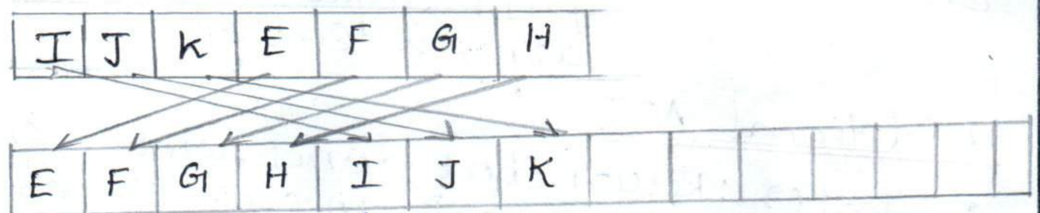
EnggTree.com

When the dequeue method is called, the current value of self.front designates the index of the value that is to be removed and returned.

The second significant responsibility of the dequeue method is to update the value of front to reflect the removal of the element, and the presumed promotion of the second element to become the new first.

Resizing the Queue.

When enqueue is called at a time, when the size of the queue equals the size of the underlying list, we rely on a standard technique of doubling the storage capacity of the underlying list.



Shrinking the underlying Array.

A desirable property of a queue implementation is to have its space usage be Θ(n) where n is the current number of elements in the queue.

The enqueue expands the array but the dequeue implementation never shrinks the underlying array.

Analyzing the Array-Based Queue Implementation.

Operation	Running Time
Q.enqueue(e)	O(1)*
Q.dequeue()	O(1)*
Q.first()	O(1)
Q.is_empty()	O(1)
len(Q)	O(1)

Downloaded from EnggTree.com

EnggTree.com

Double-Ended Queues.

A queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a double ended queue or deque.

The Deque Abstract Data Type.

D.add_first(e) : Add element e to the front of deque D

D.add_last(e) : Add element e to the back of deque D

D.delete_first() : Remove and return the first element from deque D, an error occurs if the deque is empty

D.delete_last() : Remove and return the last element from deque D; an error occurs if the deque is empty.

Additional ADT

D.first() : Return (but do not remove) the first element of deque D; an error occurs if the deque is empty

D.last() : Return (but do not remove) the last element of deque D; an error occurs if the deque is empty

D.is_empty() : Return True if deque D does not contain any elements.

len(D) : Return the number of elements in deque D; in python, we implement this with the method __len__.

Operation	Return Val	Deque
D.add_last(50)	–	[50]
D.add_first(13)	–	[13, 50]
D.add_first(7)	–	[7, 13, 50]
D.first()	7	[7, 13, 50]
D.delete_last()	50	[7, 13]
len(D)	2	[7, 13]
D.last()	13	[7, 13]
D.is_empty()	False	[7, 13]

Downloaded from EnggTree.com

EnggTree.com

Implementing a Deque with a Circular Array:

Implement the deque ADT is much the same way as the ArrayQueue class. To calculate the index of the back of the deque, or the first available slot beyond the back of the deque, we use modular arithmetic for the computation.

back = (self.front + Self.Qsize - 1) % len(self.data)

Implementation of the ArrayDeque.add_last method is same as ArrayQueue.enqueue, ArrayDeque.delete_first method is same as ArrayQueue.dequeue.

To implement add_first in deque we need to wrap around the beginning of the array.

This can be calculated as

self.front = (Self.front - 1) % len (Self.data)

The efficiency of an ArrayDeque is O(1) running time, but with that bound being amortized for operations that may change the size of the underlying list.

Deques in the python Collections Module

The Collections.deque interface was chosen to be consistent with established naming conventions of python's list class. The append and pop method used to add and remove the element at the end of the list

Therefore appendleft and popleft designate an operation at the beginning of the list. The library deque constructor also supports an optional maxlen parameter to force a fixed-length deque. If a call to append at either end is invoked when the dequeue is full, it does not throw an error, instead of that one element can be dropped from the opposite side.

The current python distribution implements Collection.deque with a hybrid approach that uses circular arrays into an organized manner. The deque class is formally documented to guarantee O(1) time operations at either end, but O(n) time worst-case operations, when using index notation near the middle of the deque.

Downloaded from EnggTree.com

EnggTree.com

The constructor instantiates the two sentinel nodes and links them directly to each other. The size member provide public support for __len__ and is_empty. so these behaviors can be directly inherited by the subclasses.

The other two methods _insert_between and _delete_node provide generic support for insertions and deletions.

The implementation of the _delete_node method is used to delete a node, the neighbors of the node to be deleted are linked directly to each other

Implementing a Deque with a DLL.

An implementation of a LinkedDeque class that inherits from the _DoublyLinkedBase class of we donot provide an explicit __init__ method for the LinkedDeque class, as the inherited version of that method suffices to initialize a new instance.

we also rely on the inherited methods __len__ and is_empty in meeting the deque ADT.

with the use of sentinels the header does not store the first element of the deque, it is the node just after the header that stores the first element. Similarly the node just before the trailer stores the last element of the deque.

we use the inherited _insert_between method to insert at either end of the deque. To insert an element at the front of the deque, we place it immediately between the header and the node just after the header. An insertion at the end of deque is placed immediately before the trailer node.

Downloaded from EnggTree.com

```
Class deque:
    def __init__(self):
        Self.data=[ ]
    def is_empty(self):
        return len(Self.data)==0
    def __len__(self):
        return len(self.data)
    def addrear(self, data):
        Self.data.append(data)
    def addfront (Self, data):
        Self.data.insert(0,data)
    def removefront(self):
        return self.data.pop(0)
    def removerear(self):
        return self.data.pop()
```

```
d=deque()
print(d.is_empty())
d.addrear(18)
d.addrear(12)
d.addfront(5).
d.addfront(29)
print(d.is_empty())
print(d.len())
print(d.removerear)
print(d.data)
```

Comparision of dequeADT and the Collections.deque class.

Deque ADT	Collections.deque	
len(d)	len(d)	number of elements
d.add_first()	d.appendleft()	add to beginning
d.add_last()	d.append()	add to end
d.delete_first()	d.popleft()	remove from beginning
d.delete_last()	d.pop()	remove from end
d.first()	d[0]	access first element
d.last()	d[-1]	access last element
	d[j]	access arbitrary entry by index
	d[j]=val	Modify arbitrary entry by index
	d.clear()	Clear all contents
	d.rotate(k)	circularly shift rightward k steps
	d.remove(e)	remove first matching element
	d.count(e)	Count number of matches for e

Downloaded from EnggTree.com

Unit II

Linear structures.

List ADT.

List is a ordered sequence of data items. Each list element must have some data type. The operations of List ADT does not depend on the element data type. A list is said to be empty when it contains no elements.

List should be able to grow and shrink in size as we insert and remove elements. We can insert and remove elements from anywhere in the list. Since insertions take place at the current position, and we can insert the element to the front or the back of the list as well as anywhere between the list.

Array-Based list Implementation.

Here is an implementation for the array-based list named ArrayList. ArrayList inherits the List ADT. ArrayList implements the List interface.

First notice that class ArrayList is required to give implementations for all of the methods listed as part of the List interface. The listArray holds the list elements. listArray, maxSize and curr are declared to be private, they may only be accessed by methods of class ArrayList.

Inserting an element at the head of an array-based list requires shifting all existing elements in the array by one position toward the tail.

Here is an array based list with five elements

list size: 5

0	1	2	3	4	5	6	7	8	9
10	20	12	15	17					

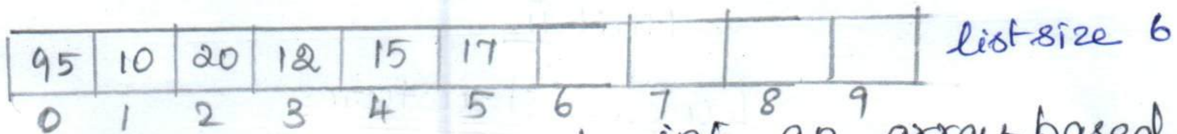
We will insert an element 95 to position 0

0	1	2	3	4	5	6
	10	20	12	15	17	

shifting all existing elements one position to the right to make room.

list size 5

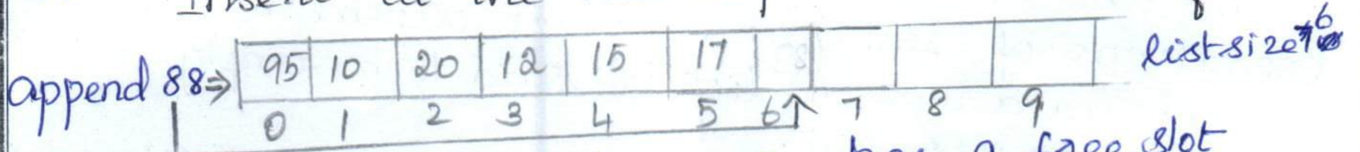
Downloaded from EnggTree.com



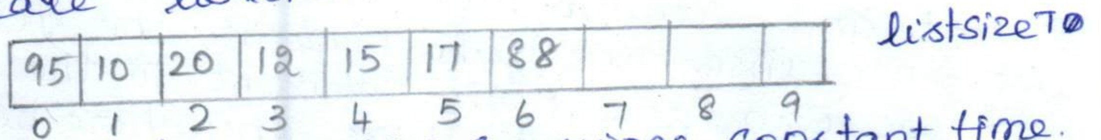
Thus the cost to insert into an array based list in the worst case O(n) where n items in the list.

Append and Remove.

Insert at the tail of the list is easy.



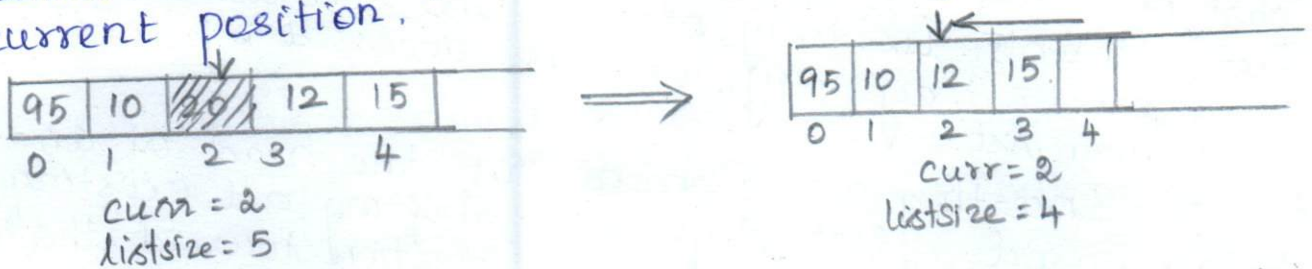
First check that the array has a free slot Now simply insert the value into the empty position and update listsize.



The append operation requires constant time.

Removing an element from the head of the list is similar to insert in that all remaining elements must shift toward the head by one position to fill in the gap. If we want to remove the element at position i, then n-i-1 elements must shift toward the head.

eg) Here is a list containing 5 elements. we will remove 20 in 2nd position of the array, which is the current position.



It decreases the listsize by 1, Return the deleted element. since we might have to shift all the remaining elements, deletion from an array-based list is O(n) in the worst case if there are n elements in the list.

In the averagecase insertion or removal each requires moving half of the elements. which is O(n).

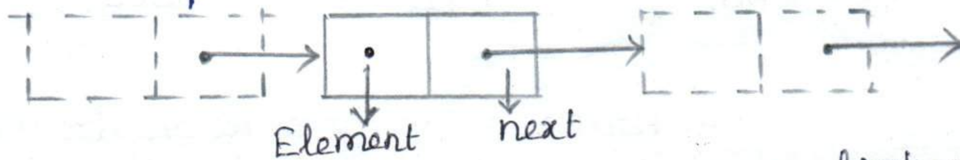
Downloaded from EnggTree.com

Linked list Implementation.

Linked list provides an alternative to an array-based sequence. A linked list in contrast, relies on a more distributed representation in which a light weight Object known as a node, is allocated for each element. Each node maintains a reference to its element and one or more references to neighboring in order to collectively represent the linear order of the sequence.

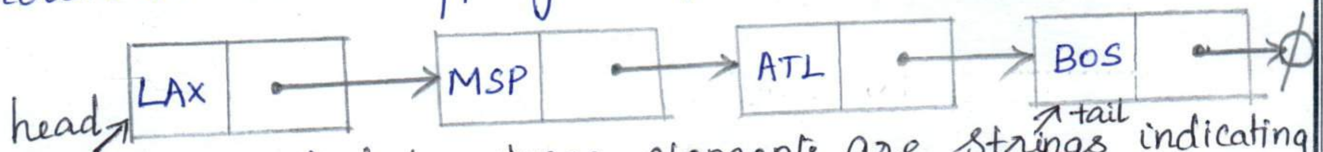
Singly Linked List.

A singly linked list is a collection of nodes that collectively form a linear sequence. Each node stores a reference to an Object that is an element of the sequence.



The first and last node of a linked list are known as the head and tail of the list.

By starting at the head, and moving from one node to another by following each node's next reference, we can reach the tail of the list. we can identify the tail as the node having None as its next reference. This process is commonly know as traversing the linked list. Because the next reference of a node can be viewed as a link or pointer to another node, the process of traversing a list is known as link hopping or pointer hopping.



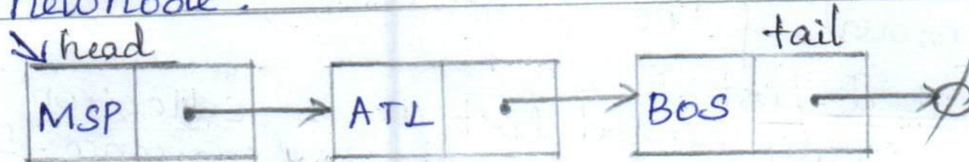
Singly Linked list whose elements are strings indicating airport codes.

The member named head that identifies the first node of the list, tail that identifies the last node of the list. The None object is denoted as ∅

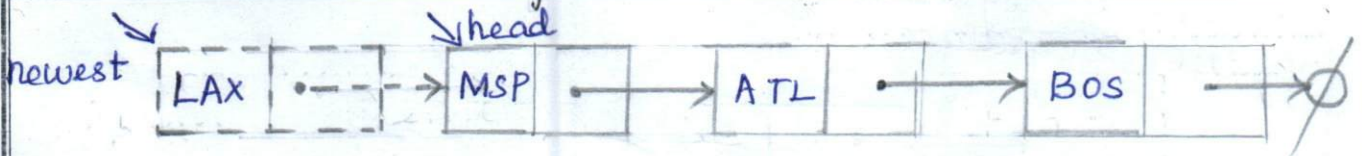
Downloaded from EnggTree.com

Inserting an Element at the Head of a SLL

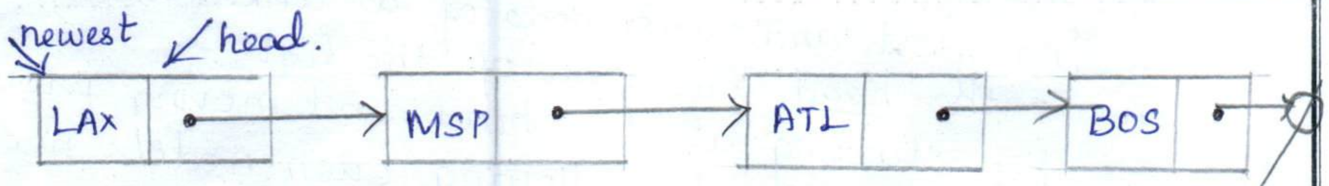
Linked list does not have a predetermined fixed size, it uses space proportionally to its current number of elements. When using a SLL, we can easily insert an element at the head of the list. The main idea is that we create a newnode, set its element to the new element, set its next link to refer to the current head, and then set the list's head to point to the newnode.



Before Insertion.



After creation of a newnode.



After reassignment of the head reference.

Algorithm

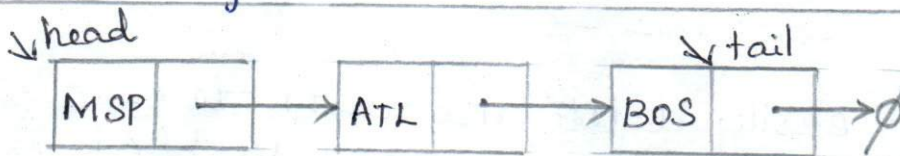
```
add_first(L, e):
    newest = Node(e)
    newest.next = L.head
    L.head = newest
    L.size = L.size + 1
```

Downloaded from EnggTree.com

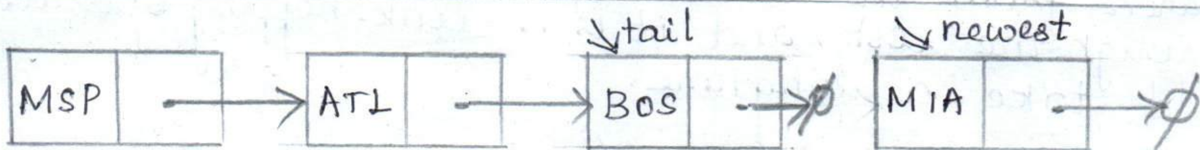
EnggTree.com

Inserting an Element at the Tail of the SLL.

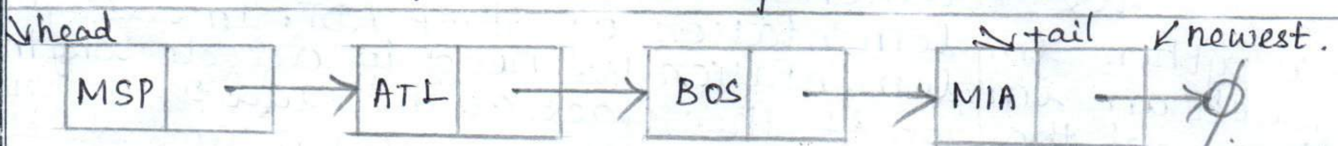
In this case, we create a newnode, assign its next reference to Node, set the next reference of the tail to point to this newnode, and then update the tail reference itself to this newnode.



Before the Insertion



After creation of a newnode



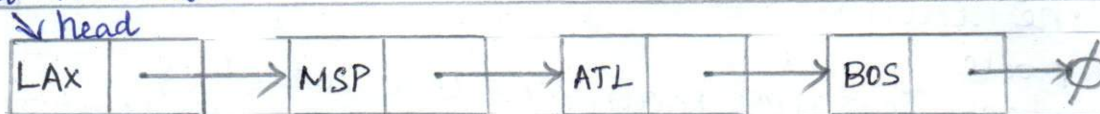
After reassignment of the tail reference.

Algorithm:

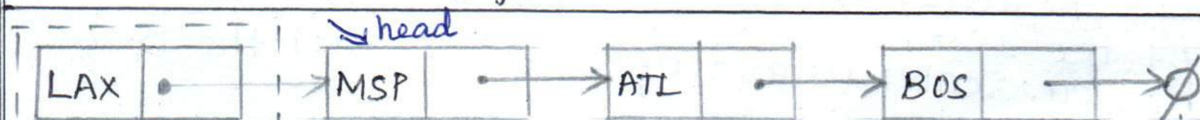
```
add_last(L, e):
    newest = Node(e)
    newest.next = None
    L.tail.next = newest
      L.tail = newest
      L.size = L.size+1
```

Removing an Element from a SLL

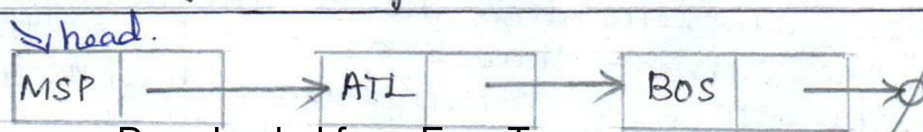
Removing an element from the head of a SLL is essentially the reverse operation of inserting a newelement at the head.



Before the removal



After linking out the old head



Downloaded from EnggTree.com

Final Configuration

 EnggTree.com

Algorithm

```
remove_first(L):
    if L.head is None then
        List is empty.
    L.head = L.head.next
    L.size = L.size - 1
```

We cannot easily delete the last node of a singly linked list. The only way to access the last node is to search from the head of the list all the way through the list, and it is a link-hopping operations could take a longtime.

Implementing a stack with a SLL.

We demonstrate use of a SLL by providing a python implementation of stack ADT. In designing such an implementation, we need to decide whether to model the top of the stack at the head or at the tail of the list. We can efficiently insert and delete elements in constant time only at the head. Since all stack operations affects the top.

The definition of Node class.

```
class_Node:
    __slots__ = '_element', '_next'  #streamline memory usage
    def __init__(self, element, next):
        self._element = element
        self._next = next.
```

A node has only two instance variables: _element and _next. The constructor of the _Node class is used to specify the initial values for both fields of a newly created node.

Each stack instance maintains two variables

* The _head member is a reference to the node at the head of the list.
* The current number of elements with the _size instance variable.

When implementing the top method, the goal is to return the element that is at the top of the stack. When the stack is nonempty, self._head is a reference to the first node of the linked list. The top element can be identified as self._head._element.

Downloaded from EnggTree.com

EnggTree.com

Implementing a Queue with a SLL

```
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""

  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = '_element', '_next'
    def __init__(self, element, next):
      self._element = element
      self._next = next

  def __init__(self):
    """Create an empty queue."""
    self._head = None
    self._tail = None
    self._size = 0                          # number of queue elements

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0

  def first(self):
    """Return (but do not remove) the element at the front of the queue."""
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._head._element              # front aligned with head of list

  def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._head._element
    self._head = self._head._next
    self._size -= 1
    if self.is_empty():                     # special case as queue is empty
      self._tail = None                     # removed head had been the tail
    return answer

  def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)            # node will be new tail node
    if self.is_empty():
      self._head = newest                   # special case: previously empty
    else:
      self._tail._next = newest
    self._tail = newest                     # update reference to tail node
    self._size += 1
```

We can use a SLL to implement the queue ADT, while supporting worst-case O(1) time for all operations. Because we need to perform operations on both ends of the queue, we will explicitly maintain both a _head reference and a _tail reference as instance variables for each queue.

Downloaded from EnggTree.com

EnggTree.com

Implementing a Stack with a SLL

```
class LinkedStack:
  """LIFO Stack implementation using a singly linked list for storage."""

  #------------------------- nested _Node class -------------------------
  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = '_element', '_next'         # streamline memory usage

    def __init__(self, element, next):      # initialize node's fields
      self._element = element               # reference to user's element
      self._next = next                     # reference to next node

  #------------------------- stack methods -------------------------
  def __init__(self):
    """Create an empty stack."""
    self._head = None                       # reference to the head node
    self._size = 0                          # number of stack elements

  def __len__(self):
    """Return the number of elements in the stack."""
    return self._size

  def is_empty(self):
    """Return True if the stack is empty."""
    return self._size == 0

  def push(self, e):
    """Add element e to the top of the stack."""
    self._head = self._Node(e, self._head)  # create and link a new node
    self._size += 1

  def top(self):
    """Return (but do not remove) the element at the top of the stack.

    Raise Empty exception if the stack is empty.
    """
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._head._element              # top of stack is at head of list
def pop(self):
  """Remove and return the element from the top of the stack (i.e., LIFO).

  Raise Empty exception if the stack is empty.
  """
  if self.is_empty():
    raise Empty('Stack is empty')
  answer = self._head._element
  self._head = self._head._next             # bypass the former top node
  self._size −= 1
  return answer
```

Downloaded from EnggTree.com

Circular Linked list.

A circularly linked list provides a more general model than a standard linked list for data sets that are cyclic, that is, which do not have any particular notion of a beginning and end.

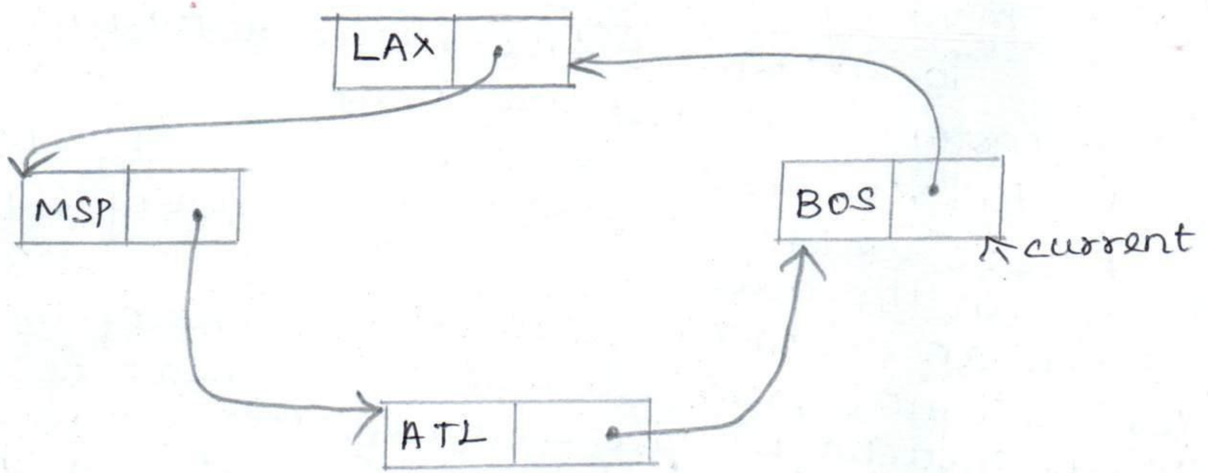
We can have the tail of the list use its next reference to point back to the head of the list call such as structure a CLL.

Even though a CLL has no beginning or end, we must maintain a reference to a particular node in order to make use of the list.

We use the identifier current to describe such a designated node.

By setting current = current.next, we can effectively advance through the nodes of the list.

Eg: SLL with circular structure.



Eg: CLL with current denoting a reference to a select node.

Downloaded from EnggTree.com

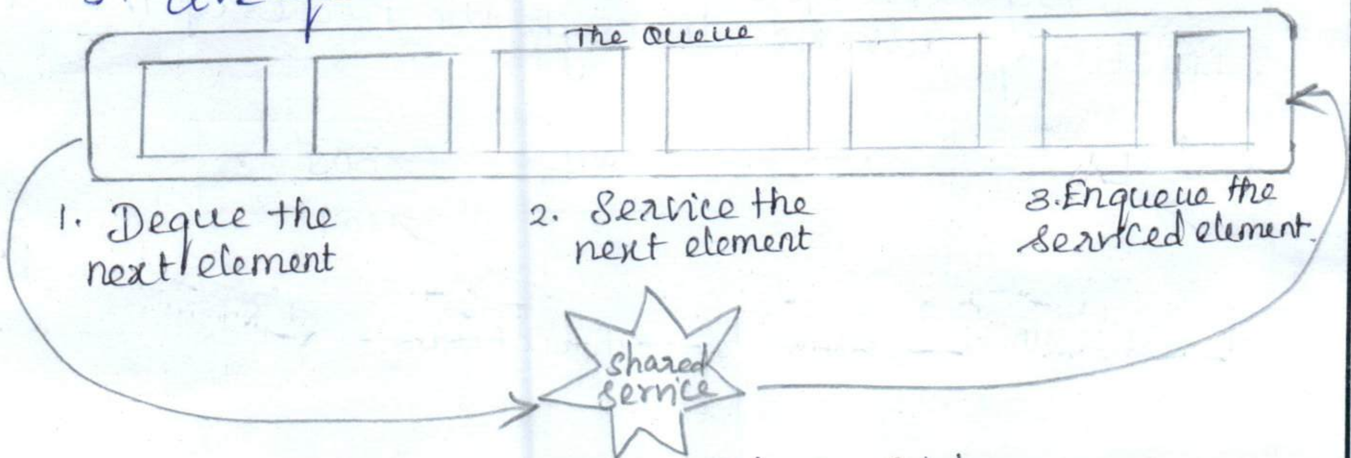
EnggTree.com

Round Robin Schedulers.

To motivate the use of a CLL, we consider a round-robin scheduler, which iterates through a collection of elements in a circular fashion and services each element by performing a given action on it. Round Robin scheduling is often used to allocate slices of CPU time to various applications running concurrently on a computer.

A round-robin scheduler could be implemented with the general queue ADT, by repeatedly performing the following steps on queue Q

1. e = Q.dequeue()
2. Service element e
3. Q.enqueue(e).



Implementing a Queue with a CLL.

Implementing the queue ADT using CLL

To implement the queue ADT using CLL two instance variables are used

- _tail, which is a reference to the tail node
- _size which is the current number of elements in the queue.

when an operation involves the front of the queue we recognize self._tail._next as the head of the queue. when enqueue is called, a newnode is placed just after the tail, but before the current head and then the newnode becomes the tail.

The CircularQueue class supports a rotate method that more efficiently enacts the combination of removing the front element and reinserting it at the back of the queue. circular representation set Self._tail=Self._tail._next to make the old head become the newtail

Downloaded from EnggTree.com

Queue-Implementation of CLL

```
class CircularQueue:
  """Queue implementation using circularly linked list for storage."""
  class _Node:
    __slots__ = '_element','_next'
    def __init__(self,element,next):
      self._element = element
      self._next    = next

  def __init__(self):
    """Create an empty queue."""
    self._tail = None                   # will represent tail of queue
    self._size = 0                      # number of queue elements

  def __len__(self):
    """Return the number of elements in the queue."""
    return self._size

  def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0
  def first(self):
    """Return (but do not remove) the element at the front of the queue.

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    head = self._tail._next
    return head._element

  def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    oldhead = self._tail._next
    if self._size == 1:                 # removing only element
      self._tail = None                 # queue becomes empty
    else:
      self._tail._next = oldhead._next  # bypass the old head
    self._size -= 1
    return oldhead._element

  def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)        # node will be new tail node
    if self.is_empty():
      newest._next = newest             # initialize circularly
    else:
      newest._next = self._tail._next   # new node points to head
      self._tail._next = newest         # old tail points to new node
    self._tail = newest                 # new node becomes the tail
    self._size += 1

  def rotate(self):
    """Rotate front element to the back of the queue."""
    if self._size > 0:
      self._tail = self._tail._next     # old head becomes new tail
```

Downloaded from EnggTree.com

EnggTree.com

Doubly Linked Lists.

A linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it, such a structure is known as Doubly linked list.

These lists allow a greater variety of O(1) time update operations, including insertions and deletions at arbitrary positions within the list.

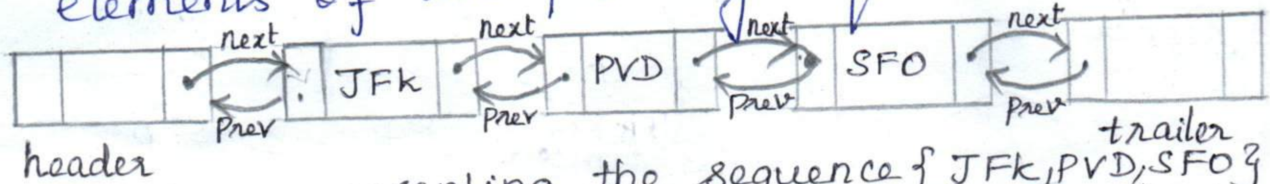
The term "next" for the reference to the node that follows another, and we introduce the term "prev" for the reference to the node that precedes it.

Header and Trailer Sentinels.

Header node → Beginning of the list

Trailer node → End of the list.

dummy node (Sentinels or guards) → they donot store elements of the primary sequence.



A DLL representing the sequence {JFk, PVD, SFO} using sentinels header and trailer to demarcate the ends of the list.

When using sentinel nodes, an empty list is initialized so that the next field of the header points to the trailer, and the prev field of the trailer points to the header, the remaining fields of the sentinels are irrelevant.

Advantage of using sentinel

The header and trailer nodes never change. only the node between them change.

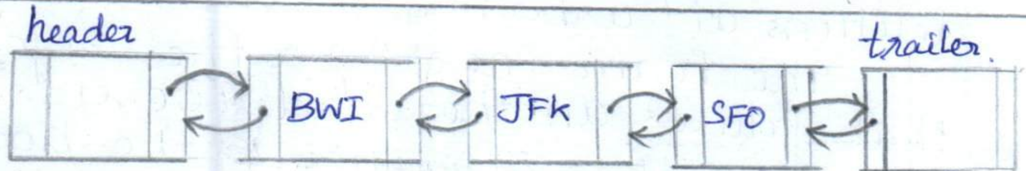
We can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes.

Downloaded from EnggTree.com

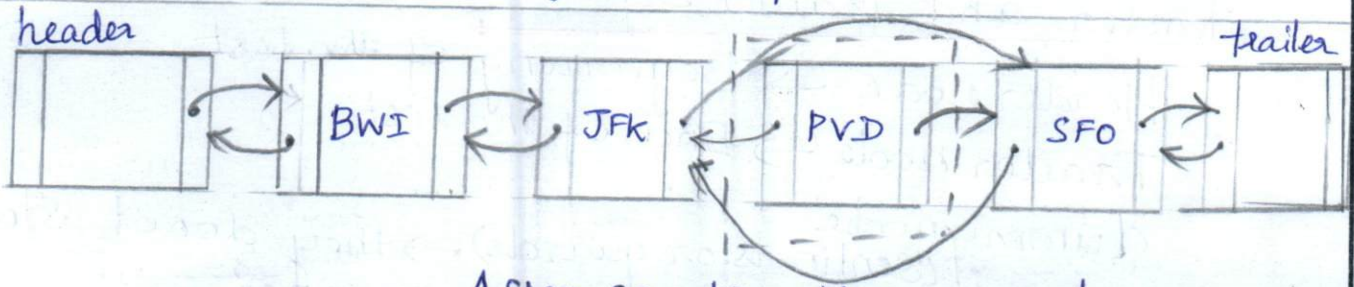
EnggTree.com

Inserting and Deleting with a Doubly Linked list.

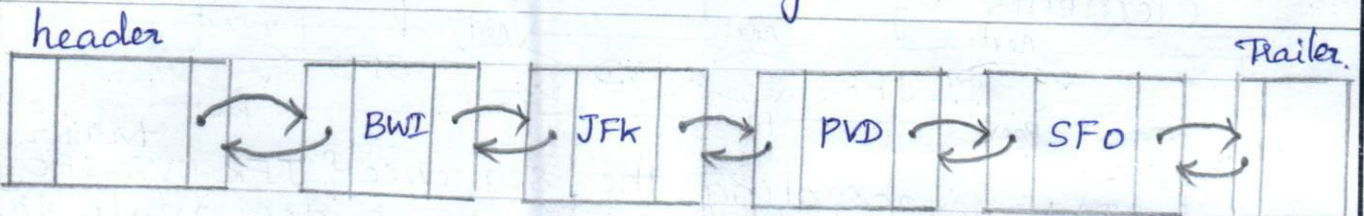
Every insertion into the Doubly linked list will take place between a pair of existing nodes. for example when a new element is inserted at the front of the sequence, we will simply add the new node between the header and the node that is currently after the header.



Before the operation.

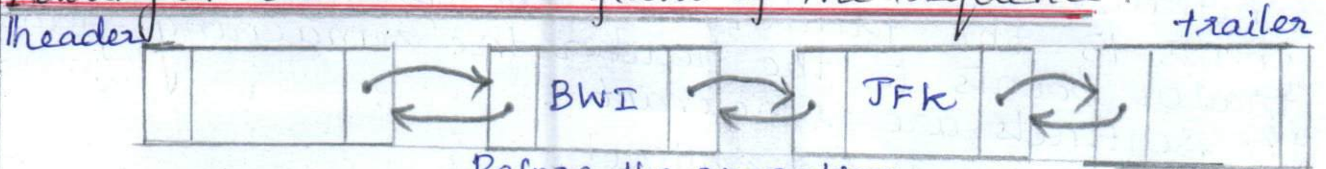


After creating the new node.

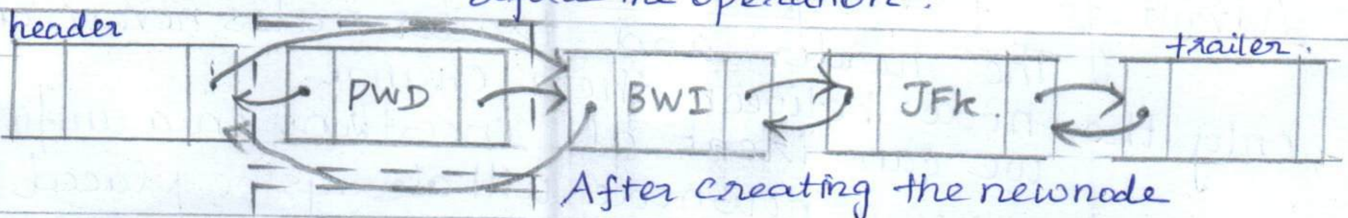


After linking the neighbors to the new node.

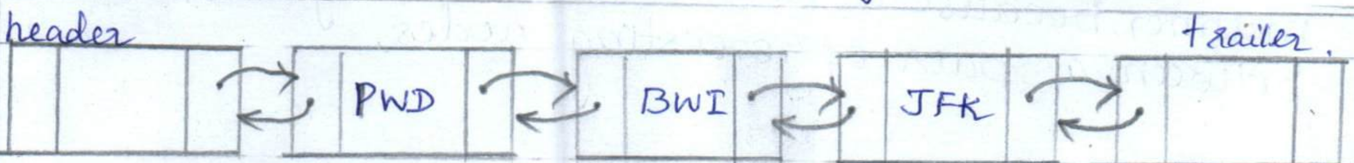
Adding an element to the front of the sequence.



Before the operation.



After creating the new node



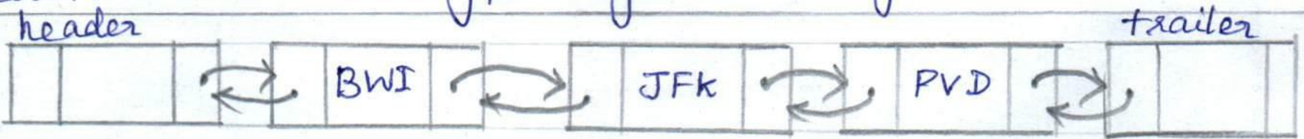
After linking the neighbors to the new node.

Downloaded from EnggTree.com

EnggTree.com

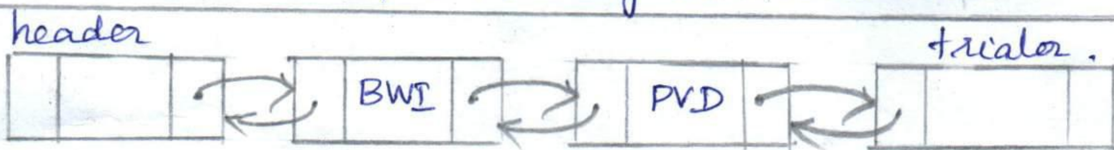
The deletion of a node.

The deletion of a node proceeds in the opposite fashion of an insertion. The two neighbors of the node to be deleted are linked directly to each other, thereby passing the original node.



Before the Removal

After linking out the old node



After the removal

Basic Implementation of a Doubly Linked list.

DoublyLinkedBase class relies on the use of a nonpublic _Node class that is similar to that for a singly linked list.

```
class _Node:
    __slots__ = '_element', '_prev', '_next'
    def __init__(self, element, prev, next):
        self._element = element
        self._prev = prev
        self._next = next
```

Downloaded from EnggTree.com

EnggTree.com

Implementation of DLL

```
class _DoublyLinkedBase:
  """A base class providing a doubly linked list representation."""
  class _Node:
      __slots__ = '_element', '_prev', '_next'
      def __init__(self, element, prev, next):
          self._element = element
          self._prev = prev
          self._next = next

  def __init__(self):
    """Create an empty list."""
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, None, None)
    self._header._next = self._trailer          # trailer is after header
    self._trailer._prev = self._header          # header is before trailer
    self._size = 0                              # number of elements

  def __len__(self):
    """Return the number of elements in the list."""
    return self._size

  def is_empty(self):
    """Return True if list is empty."""
    return self._size == 0

  def _insert_between(self, e, predecessor, successor):
    """Add element e between two existing nodes and return new node."""
    newest = self._Node(e, predecessor, successor)  # linked to neighbors
    predecessor._next = newest
    successor._prev = newest
    self._size += 1
    return newest

  def _delete_node(self, node):
    """Delete nonsentinel node from the list and return its element."""
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1
    element = node._element                     # record deleted element
    node._prev = node._next = node._element = None  # deprecate node
    return element                              # return deleted element
```

Downloaded from EnggTree.com

```python
class LinkedDeque(_DoublyLinkedBase):          # note the use of inheritance
  """Double-ended queue implementation based on a doubly linked list."""

  def first(self):
    """Return (but do not remove) the element at the front of the deque."""
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._header._next._element         # real item just after header

  def last(self):
    """Return (but do not remove) the element at the back of the deque."""
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._trailer._prev._element        # real item just before trailer

  def insert_first(self, e):
    """Add an element to the front of the deque."""
    self._insert_between(e, self._header, self._header._next)   # after header

  def insert_last(self, e):
    """Add an element to the back of the deque."""
    self._insert_between(e, self._trailer._prev, self._trailer)  # before trailer

  def delete_first(self):
    """Remove and return the element from the front of the deque.

    Raise Empty exception if the deque is empty.
    """
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._delete_node(self._header._next)    # use inherited method

  def delete_last(self):
    """Remove and return the element from the back of the deque.

    Raise Empty exception if the deque is empty.
    """
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._delete_node(self._trailer._prev)   # use inherited method
```

Downloaded from EnggTree.com

EnggTree.com

UNIT III SORTING AND SEARCHING

Bubble sort – selection sort – insertion sort – merge sort – quick sort – linear search – binary search – hashing – hash functions – collision handling – load factors, rehashing, and efficiency

Bubble sort.

A simple solution to the sorting problem is the bubble sort algorithm which rearranges the values by iterating over the list multiple times, causing larger values to bubble to the top or end of the list.

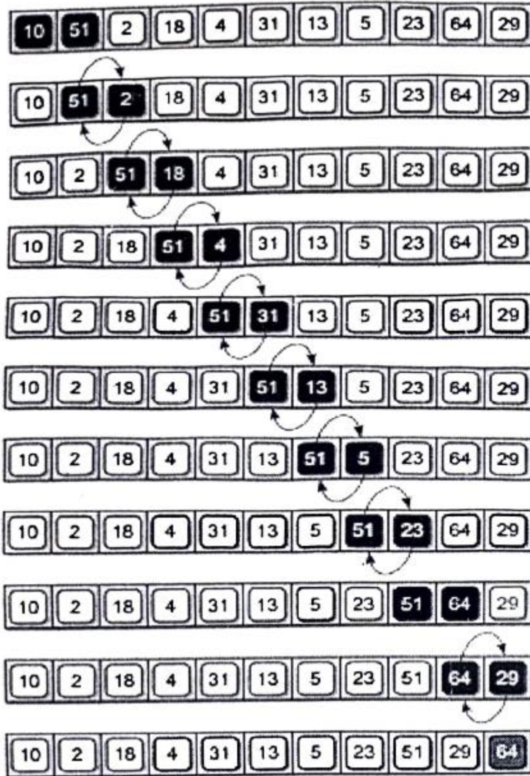
The algorithm requires multiple passes over the cards, with each pass starting at the first card and ending one card earlier than on the previous iteration. During each pass, the cards in the first and second positions are compared. If the first is larger than the second, the two cards are swapped. Next, the cards in positions two and three are compared. If the 2nd position is bigger than third one, they are swapped, otherwise we leave them as they were. This process continues for each successive pair of cards until the card with the largest face value is positioned at the end.

In the second pass the card with the second largest face value is positioned in the next to last position. In the third pass also it do the same task and the 3rd largest face value is ~~faced~~ positioned

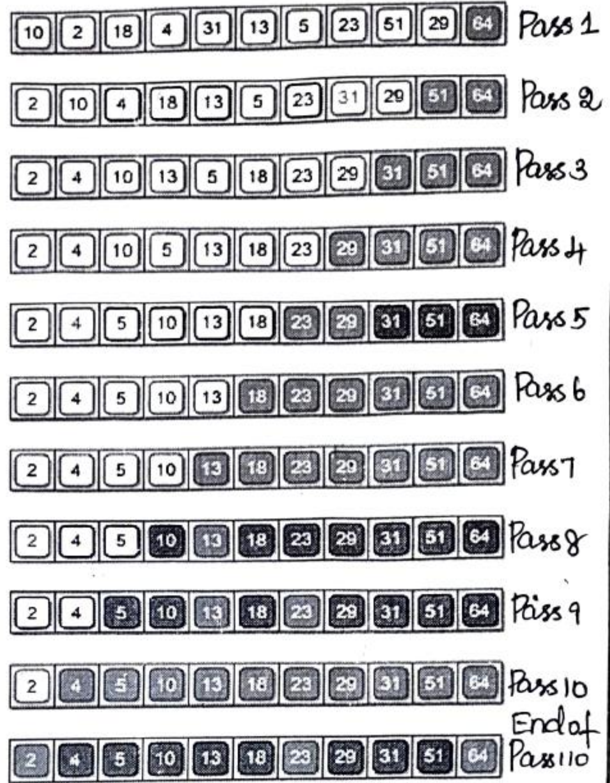
```
def bubblesort(lst):
    n=len(lst)
    for i in range(n-1):
        for j in range(n-i-1):
            if (lst[j]>lst[j+1]):
                temp=lst[j]
                lst[j]=lst[j+1]
                lst[j+1]=temp.
```

The efficiency of the bubble sort algorithm only depends on the number of keys in the array and is independent of the specific values and the initial arrangement of those values. To determine the efficiency, we must determine the total number of iterations by the innerloop for a sequence containing n values. The outerloop is executed n-1 times since the algorithm makes n-1 passes over the sequence. the number of iterations for the innerloop is not fixed, but depends on the current iteration of the outer loop.

Downloaded from EnggTree.com
CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES



First pass which places the biggest element 64 at the correct position.



It shows the remaining passes to sort the element.

Bubble sort is considered one of the most inefficient algorithms due to the total number of swap required. Given an array of keys in reverse order, a swap is performed for every iteration of the inner loop, which can be costly in practice.

Suppose if the sequence is already sorted, in this case there would be no need to sort the sequence, but our implementation still performs n^2 iterations because it has no way of knowing the sequence is already sorted.

Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

Bubble sort

11	3	7	18	12	15	13
0	1	2	3	4	5	6

n = 7

Iteration - I of the outer loop:

Iterations of inner loop:

11	3	7	18	12	15	13

i) i = 0, j = 0

lst[j] > lst[j+1] ⇒ lst[0] > lst[1] ⇒ 11 > 3 ⇒ True

temp = lst[j] = lst[0] = 11

lst[j] = lst[0] = lst[j+1] = lst[1] = 3

lst[j+1] = lst[1] = temp = 11

3	11	7	18	12	15	13

ii) i = 0, j = 1

11 > 7 ⇒ True

temp = 11

lst[1] = 7

lst[2] = 11

3	7	11	18	12	15	13

iii) i = 0, j = 2

11 > 18 ⇒ False

3	7	11	18	12	15	13

iv) i = 0, j = 3

18 > 12 ⇒ True

temp = 18

lst[3] = 12

lst[4] = 18

3	7	11	12	18	15	13

v) i = 0, j = 4

18 > 15 ⇒ True

temp = 18

lst[4] = 15

lst[5] = 18

3	7	11	12	15	18	13

vi) i = 0, j = 5

18 > 13 ⇒ True

temp = 18

lst[5] = 13

lst[6] = 18

3	7	11	12	15	13	18
0	1	2	3	4	5	6

Iteration - II of the outer loop:

Iterations of inner loop:

3	7	11	12	15	13	18
0	1	2	3	4	5	6

i) i = 1, j = 0

3 > 7 ⇒ False

3	7	11	12	15	13	18

ii) i = 1, j = 1

7 > 11 ⇒ False

3	7	11	12	15	13	18

Downloaded from EnggTree.com

EnggTree.com

3	7	11	12	15	13	18

iii) i = 1, j = 2
11 > 12 ⇒ False

3	7	11	12	15	13	18

iv) i = 1, j = 3
12 > 15 ⇒ False

3	7	11	12	15	13	18

v) i = 1, j = 4
15 > 13 ⇒ True
temp = 15
lst[4] = 13
lst[5] = 15

3	7	11	12	13	15	18
0	1	2	3	4	5	6

Iteration - III of the outer loop:
Iterations of inner loop:

3	7	11	12	13	15	18
0	1	2	3	4	5	6

i) i = 2, j = 0
3 > 7 ⇒ False

3	7	11	12	13	15	18

ii) i = 2, j = 1
7 > 11 ⇒ False

3	7	11	12	13	15	18

iii) i = 2, j = 2
11 > 12 ⇒ False

3	7	11	12	13	15	18

iv) i = 2, j = 3
12 > 13 ⇒ False

3	7	11	12	13	15	18
0	1	2	3	4	5	6

Iteration - IV of the outer loop:
Iterations of inner loop:

3	7	11	12	13	15	18
0	1	2	3	4	5	6

i) i = 3, j = 0
3 > 7 ⇒ False

3	7	11	12	13	15	18

ii) i = 3, j = 1
7 > 11 ⇒ False

3	7	11	12	13	15	18

iii) i = 3, j = 2
11 > 12 ⇒ False

3	7	11	12	13	15	18
0	1	2	3	4	5	6

Iteration - V of the outer loop:
Iterations of inner loop:

3	7	11	12	13	15	18
0	1	2	3	4	5	6

i) i = 4, j = 0
3 > 7 ⇒ False

3	7	11	12	13	15	18

ii) i = 4, j = 1
7 > 11 ⇒ False

3	7	11	12	13	15	18
0	1	2	3	4	5	6

Iteration - VI of the outer loop:
Iterations of inner loop:

i)

3	7	11	12	13	15	18
0	1	2	3	4	5	6

i = 5, j = 0
3 > 7 ⇒ False

3	7	11	12	13	15	18
0	1	2	3	4	5	6

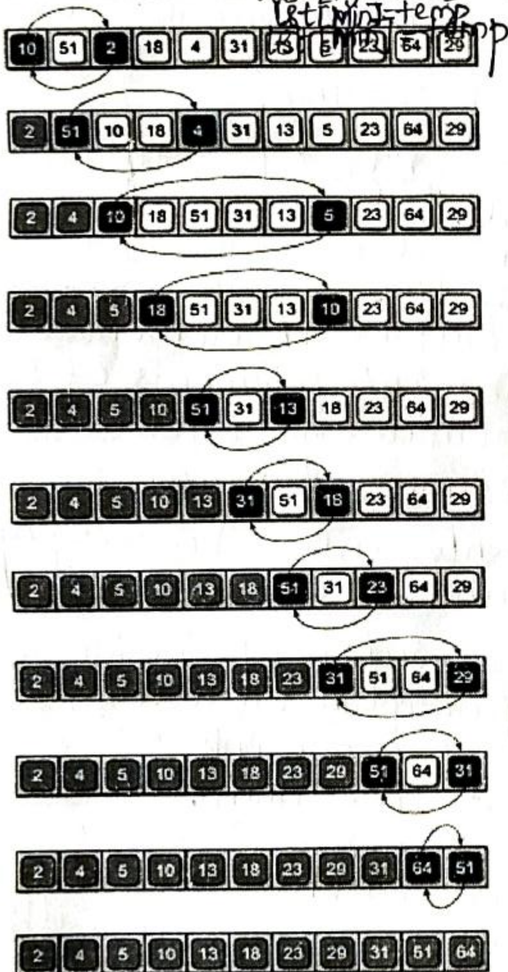
Downloaded from EnggTree.com

EnggTree.com

Selection sort.

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list, in each iteration and places the element at the beginning of the unsorted list.

```
def selection(lst):
    n = len(lst)
    for i in range(n-1):
        Min = i
        for j in range(i+1, n):
            if lst[j] < lst[Min]
                Min = j
        if Min != i:
            temp = lst[i]
            lst[i] = lst[Min]
            lst[Min] = temp
```



The process starts by finding the smallest value in the sequence and swaps it with the value in the first position of the sequence. The second smallest value is then found and swapped with the value in the 2nd pos. This process continues positioning each successive value by selecting them from those not yet sorted and swapping with the values in the respective position.

The selection sort which makes n-1 passes over the array to reposition n-1 values is also $O(n^2)$

The difference between the selection sort and bubble sort is that the selection sort reduces the number of swaps required to sort the list is $O(n)$.

Downloaded from EnggTree.com

EnggTree.com

SELECTION SORT

lst	8	48	3	17	6	38
	0	1	2	3	4	5

n=6

Pass - I: 8 48 3 17 6 38

i) i=0, min=0, j=1

lst[j] < lst[min] ⇒ 48<8 ⇒ F

8 48 3 17 6 38

ii) i=0, min=0, j=2

3<8 ⇒ T

min = 2

min != 0 ⇒ 2!=0 ⇒ T

temp = lst[i] ⇒ temp = lst[0] ⇒ temp=8

lst[i] = lst[min] ⇒ lst[0] = lst[2] ⇒ lst[0]=3

lst[min] = temp ⇒ lst[2] = 8

3 48 8 17 6 38

iii) i=0, min=0, j=3

17<3 ⇒ F

3 48 8 17 6 38

iv) i=0, min=0, j=4

6<3 ⇒ F

3 48 8 17 6 38

v) i=0, min=0, j=4

38<3 ⇒ F

3 48 8 17 6 38

Pass - II:

i) i=1, min=1, j=2

8<48 ⇒ T, 2!=1 ⇒ T
min=2,

temp=8; lst[2]=48, lst[1]=8

3 8 48 17 6 38

ii) i=1, min=1, j=3

17<8 ⇒ F

3 8 48 17 6 38

iii) i=1, min=1, j=4

6<8 ⇒ T

min=4, 4!=1 ⇒ T

temp=6, lst[4]=8, lst[2]=6

3 6 48 17 8 38

iv) i=1, min=1, j=5

38<6 ⇒ F

3 6 48 17 8 38

Pass - III:

i) i=2, min=2, j=3

17<48 ⇒ T.

min=3, 3!=2 ⇒ T

temp=17, lst[3]=48, lst[2]=17

3 6 17 48 8 38

ii) i=2, min=2, j=4

8<17 ⇒ T

min=4, 4!=2 ⇒ T

temp=8, lst[4]=17, lst[2]=8

3 6 8 48 17 38

iii) i=2, min=2, j=5

38<8 ⇒ F

3 6 8 48 17 38

Pass - IV:

i) i=3, min=3, j=4

17<48 ⇒ T

min=4, 4!=3 ⇒ T

temp=48, lst[3]=17, lst[4]=48

3 6 8 17 48 38

ii) i=3, min=3, j=5

38<17 ⇒ F

3 6 8 17 48 38

Pass - V:

i) i=4, min=4, j=5

38<48 ✓

min=5, 5!=4 ⇒ T

temp=48, lst[4]=38, lst[5]=48

3 6 8 17 38 48

Sorted list:

3	6	8	17	38	48
0	1	2	3	4	5

Downloaded from EnggTree.com

EnggTree.com

Mergesort.

Merge sort use recusion in an algorithmic design pattern called divide and conquer.

Divide and conquer.

The divide and conquer pattern consists of the following three steps.

Divide: If the inputsize is smaller than a certain threshold (one or two), solve the problem directly using a straight forward method and return the solution so obtained. otherwise, divide the inputdata into two or more disjoint subsets.

Conquer: Recursively solve the subproblems associated with the subsets.

Combine: Take the solutions to the subproblems and merge them into a solution to the original problem.

Use Divide and conquer for sorting.

Divide : If S has zero or one element return S immediately it is already sorted.
Otherwise remove all the elements from S and put them int two sequences S1 & S2, each containing about half of the elements of S, that is S1 contains the first [n/2] elements of S and S2 contains the remaining [n/2] elements.

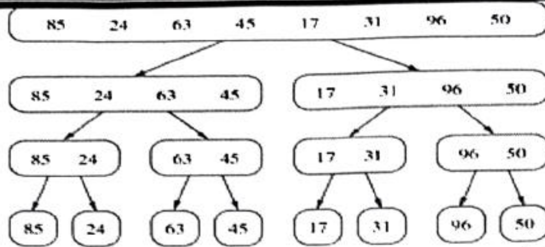
Conquer : Recursively sort sequences S1 and S2

Combine : Put back the elements into S by merging the sorted sequence S1 and S2 into a sorted sequence

We can visualize an execution of the Mergesort algorithm by means of a binary tree T, called the Merge sort tree. Each node of T represents a recursive invocation of the mergesort algorithm. We associate with each node V of T the sequence S that is processed by the invocation associated with V. The children of node V are associated with the recursive calls that process the subsequence S1 and S2 of S. The external nodes of T are associated with individual elements of S; corresponding to instances of the algorithm that makes no recursive calls.

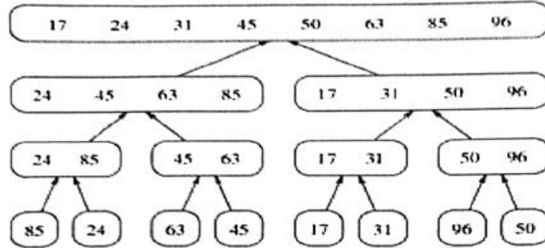
Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

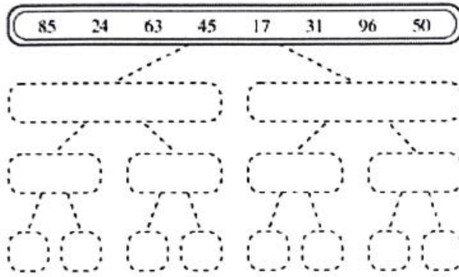


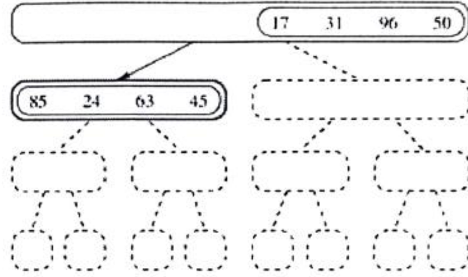
Input sequences processed at each node T

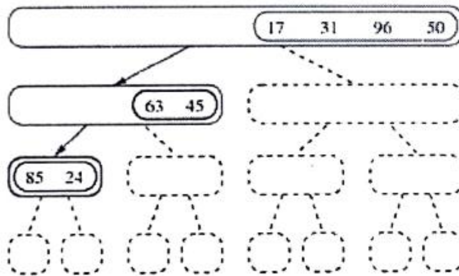
Merge-sort tree T for an execution of the mergesort algorithm on a sequence with 8 elements.

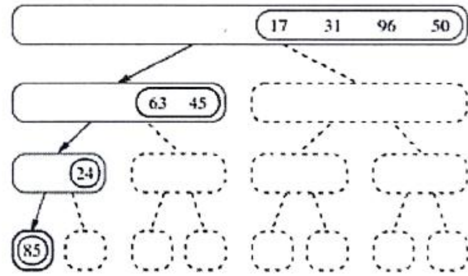


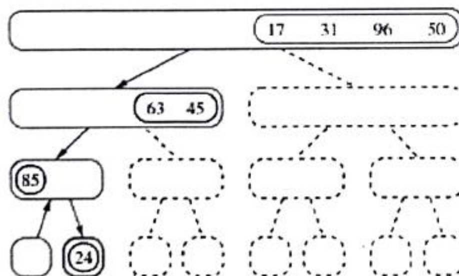
Output sequences generated at each node of T

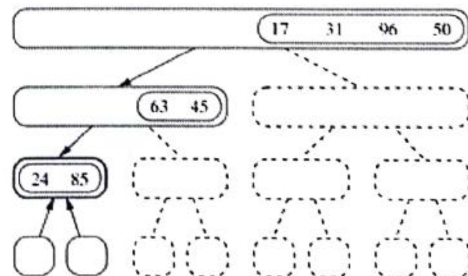

(a)


(b)


(c)

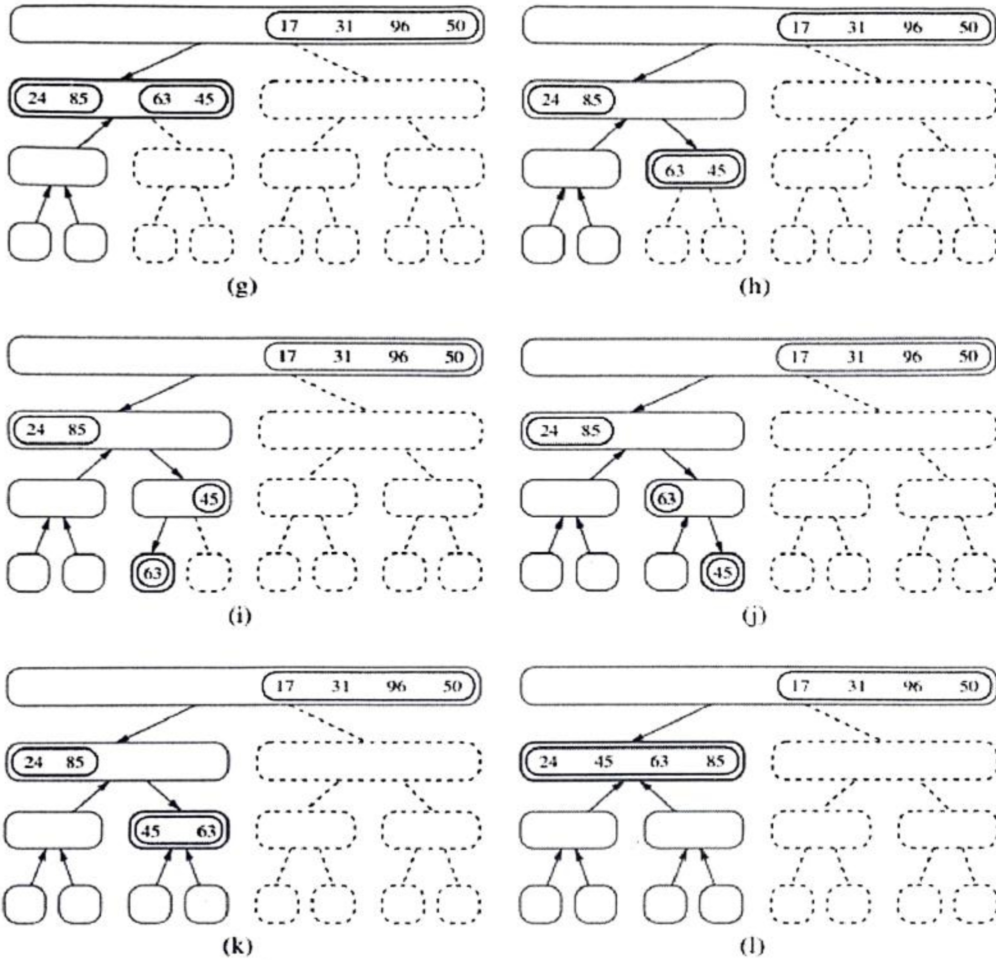

(d)


(e)

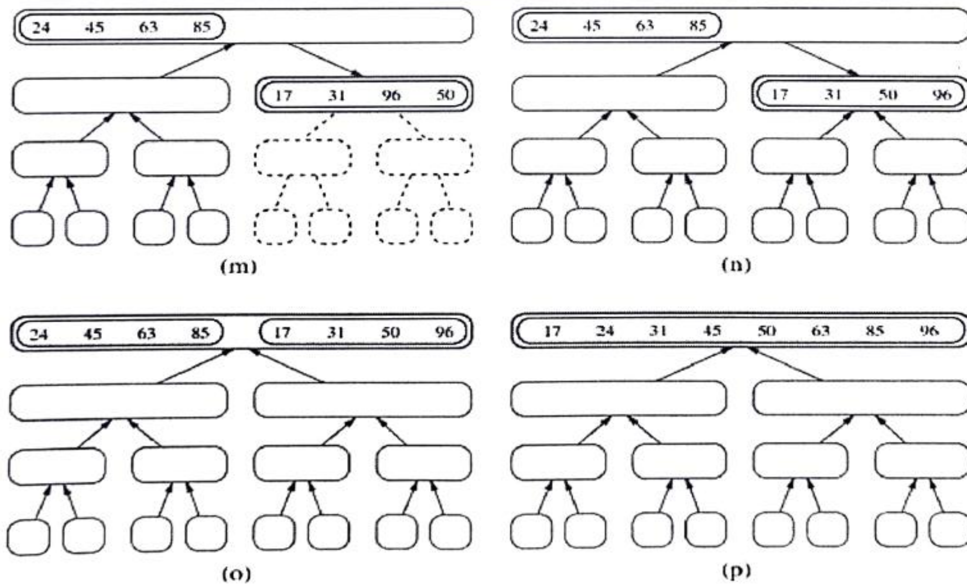

(f)

Visualization of an execution of merge-sort. Each node of the tree represents a recursive call of merge sort. The node drawn with dashed lines represent the current call. The empty nodes drawn with thin lines represent completed calls. The remaining nodes represent calls that are waiting for a child invocation to return.

Downloaded from EnggTree.com



Visualization of an Execution of Mergesort



Visualization of an execution of Mergesort.

Downloaded from EnggTree.com

EnggTree.com

Arraybased Implementation of Mergesort

The sequence of items is represented as an python list. The merge function is responsible for the subtask of merging two previously sorted sequences S1 and S2 with the output copied into S. We copy one element during each pass of the while loop, conditionally determining whether the next element should be taken from S1 or S2.

```
def merge(S1, S2, S):
    i = j = 0
    while i+j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
            S[i+j] = S1[i]
            i += 1
        else
            S[i+j] = S2[j]
            j += 1
```

We illustrate a step of the merge process. During the process, index i represents the number of elements of S1 that have been copied to S, while index j represents the number of elements of S2 that have been copied to S. Assuming S1 & S2 both have at least one uncopied element, we copy the smaller of the two elements being considered. If we reach the end of one of the sequences we must copy the next element from the other.

(a) Before the copy

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S_1	2	5	8	11 (i)	12	14	15							
S_2	3	9	10 (j)	18	19	22	25							
S	2	3	5	8	9	(i+j)								

(b) After the copy

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
S_1	2	5	8	11 (i)	12	14	15							
S_2	3	9	10	18 (j)	19	22	25							
S	2	3	5	8	9	10	(i+j)							

A step in the merge of two sorted arrays for which S2[j] < S1[i].

Downloaded from EnggTree.com

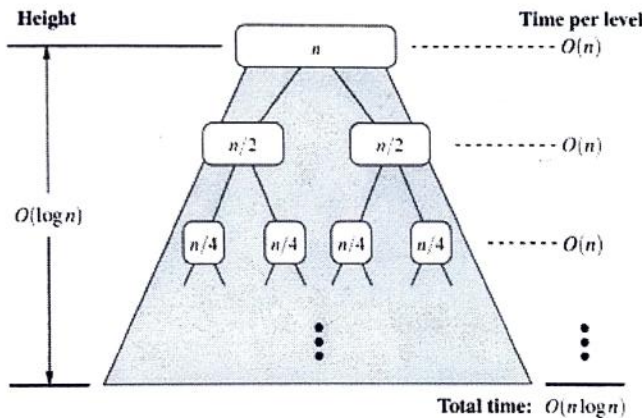
CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

```
def merge_sort(S):
    n = len(S)
    if n<1:
        return
    mid = n//2
    S1 = S[0:mid]   #copy of first half
    S2 = S[mid:n]   # copy of second half
    merge_sort(S1)  # sort copy of first half
    merge_sort(S2)  # sort copy of second half
    merge(S1, S2, S) # merge sorted halves back into S.
```

The running time of mergesort.

Let n_1 and n_2 be the number of elements of S1 and S2 respectively. It is clear that the operations performed inside each pass of the while loop take O(1) time. During each iteration of the loop, one element is copied from either S1 or S2 into S. Therefore the number of iterations of the loop is n_1+n_2. The running time of algorithm is $O(n_1+n_2)$.



Recurrence Equation.

Let the function t(n) denote the worst case running time of mergesort on an input sequence of size n. Since merge sort is recursive, we can characterize function t(n) by means of an equation where the function t(n) is recursively expressed in terms of itself.

$$t(n) = \begin{cases} b & \text{if } n \le 1 \\ 2t(n/2) + cn & \text{otherwise.} \end{cases}$$

Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

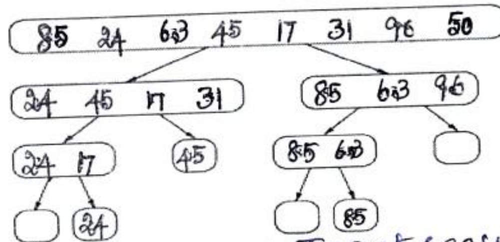
QuickSort.

Quicksort algorithm is also based on the divide and conquer paradigm.

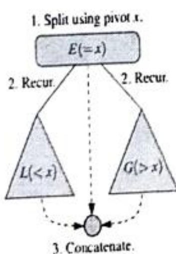
High-level Description of QuickSort.

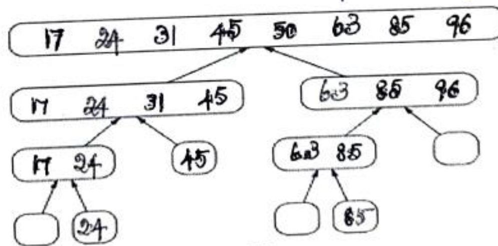
The quicksort algorithm sorts a sequence S using a simple recursive approach.

* **Divide** : If S has at least two elements, select a specific element x from S, which is called the pivot. As in common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences.
    * L, storing the elements in S less than x
    * E, Storing the elements in S equal to x
    * G, Storing the elements in S greater than x.
* **Conquer** : Recursively sort sequences L and G
* **Combine** : Put back the elements into S inorder by first inserting the elements of L then those of E, and finally those of G.



(a) Input sequences processed at each node of T





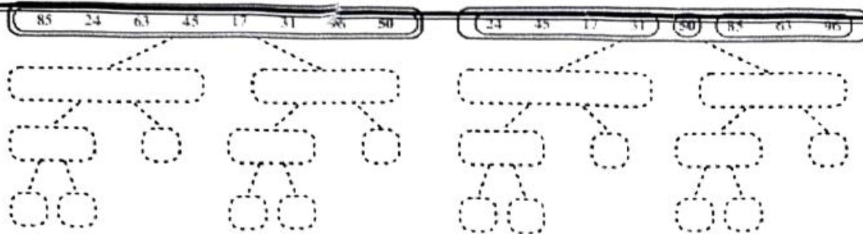
(b)

Output sequences generated at each node of T

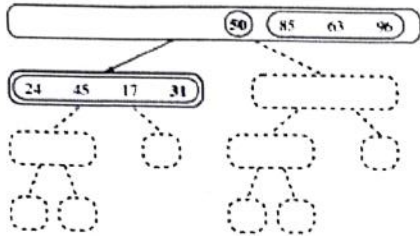
Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

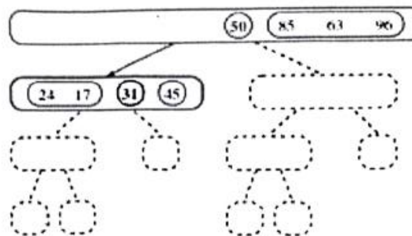
EnggTree.com



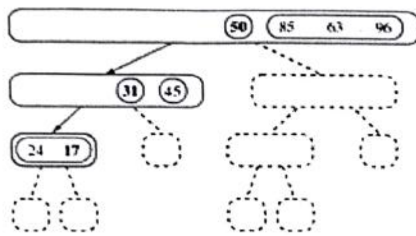
(a) (b)



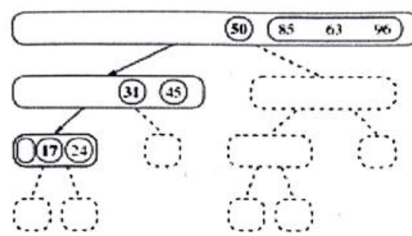
(c)



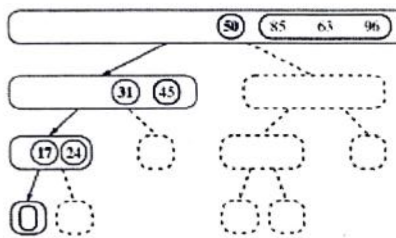
(d)



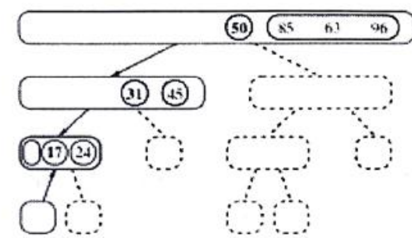
(e)



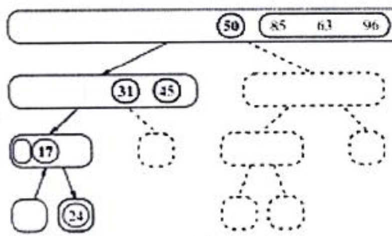
(f)



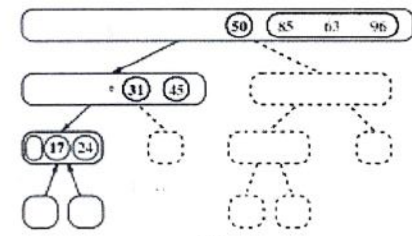
(g)



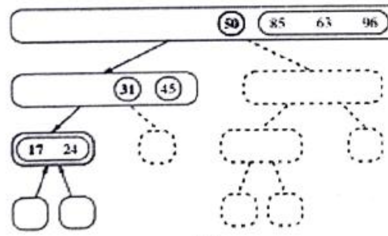
(h)



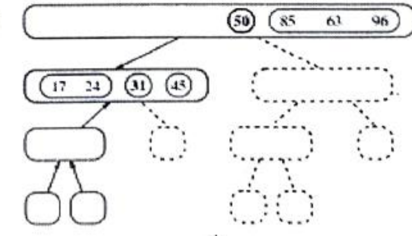
(i)



(j)



(k)

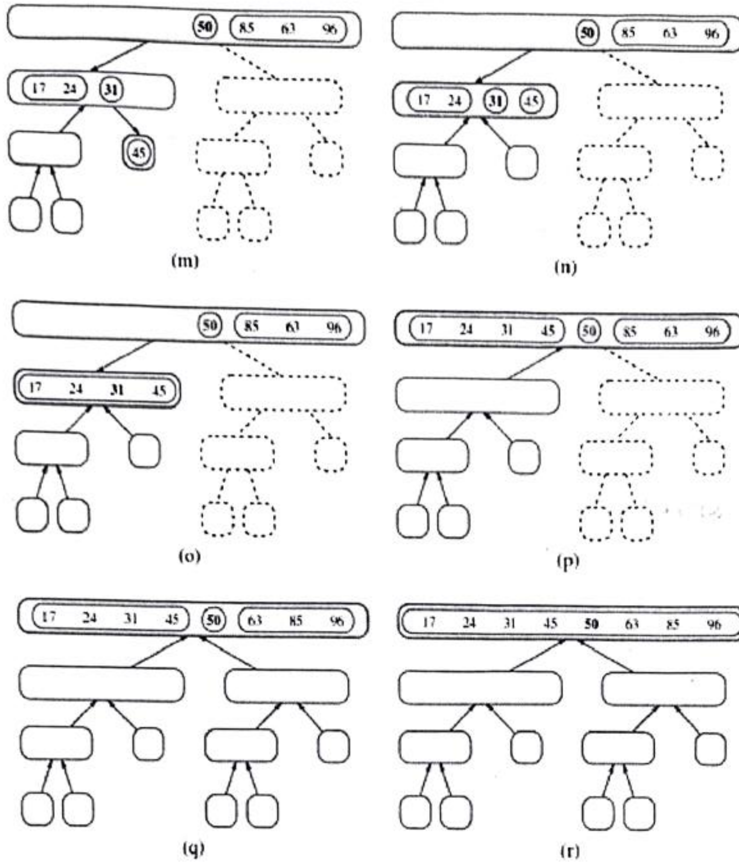


(l)

Visualization of an execution of quicksort.

Downloaded from EnggTree.com

EnggTree.com



Visualization of an execution of quicksort.
concatenation steps performed in o and r

Performing Quicksort on General sequences.

```
def quicksort(S):
    n = len(S)
    if n < 1:
        return
    p = S.first()
    L = LinkedQueue()
    E = LinkedQueue()
    G = LinkedQueue()
    while not S.isempty():
        if S.first() < p:
            L.enqueue(S.dequeue())
        elif p < S.first():
            G.enqueue(S.dequeue())
        else
            E.enqueue(S.dequeue())
    quicksort(L)
    quicksort(G)
    while not L.isempty():
        S.enqueue(L.dequeue())
    while not E.isempty():
        S.enqueue(E.dequeue())
    while not G.isempty():
        S.enqueue(G.dequeue())
```


Downloaded from EnggTree.com

Running Time of Quick sort.

The divide step and the final concatenation of quicksort can be implemented in linear time. Thus the time spend at a node v of T is proportional to the input size s(v) of v, defined as the size of the sequence handled by the invocation of quick sort associated with node v.

Since subsequence E has atleast one element (pivot), the sum of the input sizes of the childen of v is atmost s(v) − 1

Let s_i denote the sum of the input sizes of the nodes at depth i for a particular quick sort tree T. clearly $s_0 = n$, since the root r of T is associated with the entire sequence, also $s_1 \leq n-1$, since the pivot is not propagated to the children of r.

The overall running time of an execution of quick sort as $O(n.h)$ where h is the overall height of the quick sort tree T for that execution.

The overall performance of quick sort algorithm is $O(n \log n)$

Randomized quick sort.

One common method for analyzing quicksork is to assume that the pivot will always divide the sequence in a reasonably balanced manner.

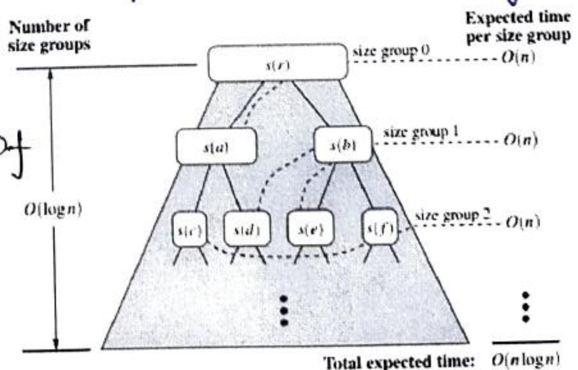
Having pivots close to the "middle" of the set of elements leads to an $O(n \log n)$ running time for quicksort.

Picking pivots at random.

Instead of picking the pivot as the first or last element of S, we pick an element of S at random as the pivot, keeping the rest of the algorithm unchanged.

The variation of quick sort is called randomized quicksort. The expected running time of randomized quicksort on a sequence S of size n is $O(n \log n)$.

A visual time analysis of the quick sort tree T.



Downloaded from EnggTree.com

Insertion sort.

The insertion sort is a straight forward and more efficient algorithm. Insertion sort iterate over the input elements by growing the sorted array at each iteration compare the current element with the largest value available in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position in the array. This is achieved by shifting all the elements towards the right, which are larger than the current element, in the sorted array to one position ahead.

Algorithm.

```
def insertionsort(lst):
    n=len(lst)
    for i in range(1,n):
        min = lst[i]
        j = i-1
        while j>=0 and min<lst[j]:
            lst[j+1] = lst[j]
            j -= 1
        lst[j+1] = min
```

Time complexity of insertion sort algorithm is $O(n^2)$

eg) 8, 5, 18, 2, 7

n = 5, **i = 1**
min = 5
j = 0.
5<8 (T)
5, 8, 18, 2, 7.

i = 2
min = 18
j = 1
1>=0 and 18<8 (F).
lst [2] = 18
5, 8, 18, 2, 7

i = 3
min = 2
j = 2
2>=0 and 2<18 (T)
lst [3] = 18
5, 8, _, 18, 7
j = 1
1>=0 and 2<8 (T)
lst [2] = 8
5, _, 8, 18, 7
j = 0
0>=0 and 2<5 (T)
lst [1] = 5
_, 5, 8, 18, 7
lst [0] = 2
2, 5, 8, 18, 7.

i = 4 min = 7.
j = 3
3>=0 and 7<18 (T).
lst [4] = 18
2, 5, 8, _, 18
j = 2.
2>=0 and 7<8 (T)
lst [3] = 8.
2, 5, _, 8, 18
j = 1
1>=0 and 7<5 (F).
lst [2] = 7
2, 5, 7, 8, 18

Downloaded from EnggTree.com

Linear Search.

Linear search is a very simple search Algorithm. In this type of search, a sequential search is made overall items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continus till the end of the elements in the list.

Algorithm.

```
def linearsearch(lst, x):
    n = len(x)
    for i in range(n):
        if lst[i] == x:
            return i
    return -1
```

Eg : input list : 50, 38, 42, 12, 10, 17, 19, 4

n = 8

Search element x = 12

i = 0	lst[0] = 50	check 50 == 12 (False).
i = 1	lst[1] = 38	check 38 == 12 (False)
i = 2	lst[2] = 42	check 42 == 12 (False)
i = 3	lst[3] = 12	check 12 == 12 (True)

Linear search algorithm is suitable for smaller list, because it check every element to get the desired number suppose there are 1000 element list and desired element is available at the last position, this will consume much time by comparing with each element of the list.

Time complexity of linear search Algorithm

Best case = O(1)

Average case = O(n)

worst case = O(n).

Downloaded from EnggTree.com

EnggTree.com

Binary search.

Binary search is a fast search algorithm with runtime complexity of O(log n). This search algorithm works on the principle of divide and conquer. condition for this algorithm is that element must be in sorted form.

Binary search searches a particular item by comparing the middle most item of the element. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the subarray to the right of the middle item, otherwise, the item is searched for in the sub-array to the left of the middle item. This process continues on the subarray as well until the size of the subarray reduces to zero.

Eg:

Element	9	15	18	28	30	38	42	48	52	65
Index	0	1	2	3	4	5	6	7	8	9

~~mid = (low + high)//2~~.

Algorithm.

```
def binarysearch(lst, x, low, high):
    if low > high:
        return False
    else:
        mid = (low+high)//2
        if x == lst[mid]:
            return True
        elif x < lst[mid]:
            return binarysearch(lst, x, low, mid-1)
        else:
            return binarysearch(lst, x, mid+1, high)
```


Downloaded from EnggTree.com

EnggTree.com

eg)

9	15	18	28	30	38	42	48	52	65
0	1	2	3	4	5	6	7	8	9

mid = 0+9//2 = 4 mid

Search element = 38

Now we compare the value stored at location 4, with the search element 38 (ie) lst[4] = 30 ⇒ 38 = =30 False

The search value is greater than mid element. So we change the low to mid+1 and find the new mid value again.

Now low = mid+1 ⇒ 4+1 , high = 9

mid = 5+9//2 = 7

9	15	18	28	30	38	42	48	52	65

lst[7] = 48 , search element = 38.

The value stored at location 7 is not a match, rather it is less than what we are searching. So the value must be the lower part from this location.

9	15	18	28	30	38	42	48	52	65

we calculate the mid again.

low = 5; high = mid - 1 = 6 mid = 5+6//2 = 5.

we compare the value stored at location 5 with our target value. we find the search element

9	15	18	28	30	38	42	48	52	65

we conclude that the target value 31 is stored at location5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

The complexity of binary search algorithm is

O(log n)

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES
Downloaded from EnggTree.com

EnggTree.com

Hashing.

Hashing is a technique or process of mapping keys, Values into the hash table by using a hash function. The efficiency of mapping depends on the efficiency of the hash function.

Hash Tables.

Hash table is a datastructure which stores data in an associative manner, in which the address or the index value of the data element is generated from a hash function. This makes accessing the data faster as the index value behaves as a key for the data value.

In python, the dictionary datatypes represent the implementation of hash tables.

A Map M supports the abstraction of using keys as indices with a syntax such as M[k]. A map with n items uses keys that are known to be integers in a range from 0 to N-1 for some $N \geq n$. In this case we can represent the map using a lookup table of length N.

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

A lookup table with length 11 for a map containing items (1,D), (3,Z), (6,C) and (7,Q)

In this representation, we store the value associated with key k at index k of the table. Basic map operations of __getitem__, __setitem__ and __delitem__ can be implemented in O(1) worst case.

Challenges in extending this framework.

* First we may not wish to devote an array of length N if it is the case that N >> n.
* Second, we do not in general require that a map's keys be integers.

Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

The hash table is used to map hash function, general keys to corresponding indices in a table.

Keys will be well distributed in the range from 0 to N-1 by a hash function, but in practice there may be two or more distint keys that get mapped to the same index.

we will conceptualize our table as a bucketarray may manage a collection of items that are sent to a specific index by the hash function.

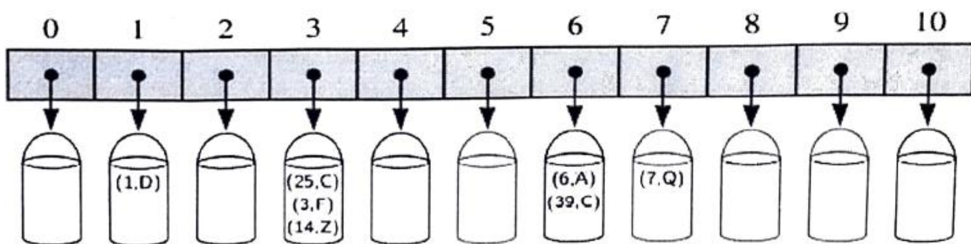


Figure 10.4: A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

Hash functions.

The hash function h, is to map each key k to an integer in the range [0,N-1], where N is the capacity of the bucket array for a hash table.

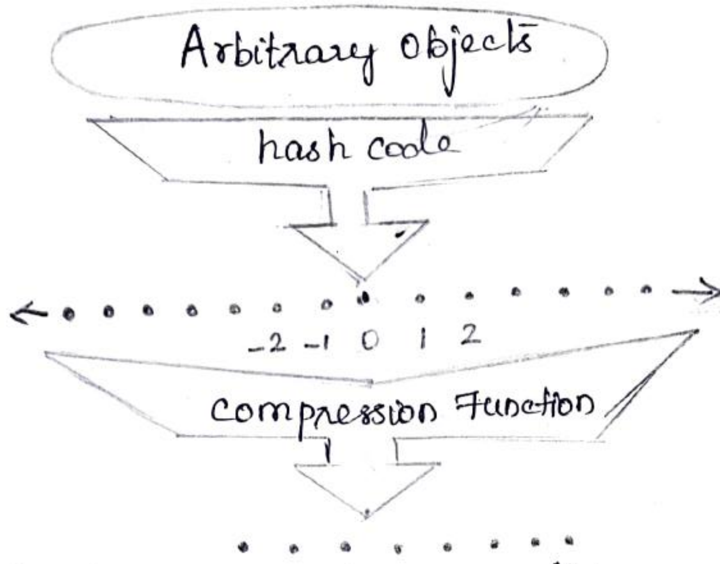
The main idea of this approach is to use the hash function value h(k) as an index into our bucket array A instead of the key k. That is we store the item (k,v) in the bucket A[h(k)]

If there are two or more keys with the same hash value, then two different items will be mapped to the same bucket in A, in this case collision has occured.

It is common to view the evaluation of a hash function h(k) as consisting of two portions - a hashcode that maps a key k to an integer and a compression function that maps the hash code to an integer within a range of indices [0,N-1] for a bucket array.

Downloaded from EnggTree.com

EnggTree.com



Two parts of a hash function: a hash code & a compression fun.

The advantage of separating the hash function into two such components is that the hash code portion of that computation is independent of a specific hash table size. This allows the development of a general hash code for each object that can be used for a hash table of any size, only the compression function depends upon the table size. This is convenient, because the underlying bucket array for a hash table may be dynamically resized, depending on the number of items currently stored in the map.

Hash codes

The first action that a hash function performs is to take an arbitrary key k in our map and compute an integer that is called the hash code for k, this integer need not be in the range [0,N-1], and may even be negative.

Treating the Bit representation as an Integer

To any data type X that is represented using at most as many bits as our integer hashcodes, we can simply take as a hash code for X an integer interpretation of bits.

For example, the hash code for key 314 could simply be 314. The hash code for a floating point number 3.14 could be based upon an interpretation of the bits of the floating point representation as an integer.

Downloaded from EnggTree.com

EnggTree.com

For a type whose bit representation is longer than a desired hash code. For example python relies on 32 bit hash codes. If a floating point number uses a 64-bit representation, its bits cannot be viewed directly as a hash code.

A better approach is to combine in some way the high-order and low-order portions of a 64-bit key to form a 32 bit hash code, which takes all the original bits into consideration.

Polynomial Hash codes.

A polynomial a that takes the components $(x_0, x_1 \ldots x_{n-1})$ for an object x as coefficients. This hash code is therefore called a polynomial hash code.

$$x_0 a^{n-1} + x_1 a^{n-2} + x_2 a^{n-3} \ldots + x_{n-2} a + x_{n-1}$$

By Horner's rule this polynomial can be computed as

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \ldots + a(x_2 + a(x_1 + a x_0))))$$

A polynomial hash code uses multiplication by different powers as a way to spread out the influence of each component across the resulting hash code.

Cycleshift Hash codes.

A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.

Eg. A 5 bit cyclic shift of the 32 bit value 00111101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 10110010110101010001010100000111. In python a cyclic shift of bits can be accomplished through careful use of the bitwise operators << and >>, taking care of truncate results to 32 bit integers.

```
def hash_code(s)
    mask = (1<<32) - 1
    h = 0
    for c in s:
        h = (h<<5 & mask) | (h>>27)
        h += ord(c)
    return h
```

Downloaded from EnggTree.com

	Collisions	
Shift	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

comparison of collision behavior for the cyclic-shift hash code as applied a list of 230,000 English words.

Hash codes in python.

The standard mechanism for computing hash codes in python is a built-in function with signature hash(x) that returns an integer value that serves as the hash code for object x. Only immutable datatypes are deemed hashtable in python. This restriction is meant to ensure that a particular object's hash code remain constant during that objects life span.

Among python's built in data types, the immutable int, float, str, tuple and frozenset classes produce robust hash codes via the hash function.

A problem could occur if a key were inserted into the hash table, yet a later search were performed for that key based on a different hash code than that which it had when inserted, the wrong bucket would be searched.

Instances of user defined classes are treated as unhashable by default, with a typeerror raised by the hash function. However a function that computes hash codes can be implemented in the form of a special method named hash within a class. The returned hash code should reflect the immutable attributes of an instance.

(eg) A color class that maintains three numeric red, green & blue components might implement the method as

```
def hash(self):
    return hash(self.red, self.green, self.blue)
```

Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

An important rule to obey is that if a class defines equivalence through __eq__ then any implementation of __hash__ must be consistent in that if x == y then hash(x) == hash(y).

(eg) 5 == 5.0 as true it ensures hash(5) & hash(5.0) are the same.

Compression Functions

The hash code for a key k will typically not be suitable for immediate use with a bucket array, because the integer hash code may be negative or may exceed the capacity of the bucket array. Thus once we have determined an integer hash code for a key object k, there is still the issue of mapping the integer into the range [0,N-1]. This computation known as a compression function.

* Division Method
* MAD Method.

Division method.

A simple compression function is the division method, which maps an integer i to i mod N where N, the size of the bucket array is a fixed positive integer.

(Eg). If we insert keys with hash codes {20,205,205,210,600} into a bucket array of size 100, then each hash code will collide with three others. But if we use a bucket array of size 101, then there will be no collisions.

If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is 1/N

Choosing N to be a prime number is not always enough, however, for if there is repeated pattern of hash codes of the form pN+q for several different p's, then there will still be collisions.

Downloaded from EnggTree.com
CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

The MAD method.

A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys, is the Multiply-Add and Divide (MAD) method. This method maps an integer i to

[(ai+b) mod p] mod N.

where N is the size of the bucket array, p is a prime number larger than N, and a & b are integers chosen at random from the interval [0, p-1] with a>0.

This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a good hash function, that is one such that the probability any two different keys collide is 1/N.

Collision-Handling Schemes.

The main idea of a hash table is to take a bucket array A and a hash function h and use them to implement a map by storing each item (k,v) in the bucket A[h(k)].

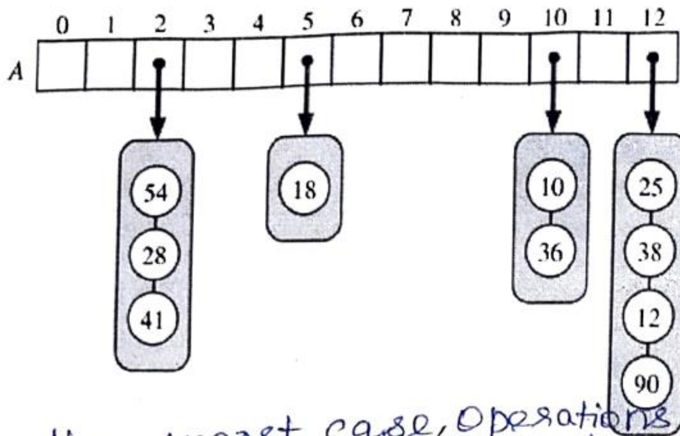
The simple idea is challenged, however, when we have two distinct keys k1 and k2. such that h(k1) = h(k2). The existence of such collisions prevent us from simply inserting a new item (k,v) directly into the bucket A[h(k)]. It also complicates our procedure for performing insertion, search and deletion operations.

Separate chaining

A simple and efficient way for dealing with collisions is to have each bucket A[j] store its own secondary container [illegible] items (k,v) such that h(k) = j. A natural choice for the secondary container is a small map instance implemented using a list. The collision resolution rule is known as separate chaining

Downloaded from EnggTree.com

EnggTree.com



A hash table of size 13 storing 10 items with integer keys with collisions resolved by separate chaining. The Compression function is h(k) = k mod 13.

In the worst case, operations on an individual bucket take time proportional to the size of the bucket. Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N, the expected size of a bucket is n/N. Therefore if given a good hash function, the core map operations run in O([n/N]). The ratio λ = n/N called the load factor of the hash table, should be bounded by a small constant preferably below 1. As long as λ is O(1), the core operations on the hash table run in O(1) expected time.

Open Addressing.

The separate chaining has one slight disadvantage. It requires the use of an auxiliary data structure a list to hold items with colliding keys. If space is at a premium, then we can use the alternative approach of always storing each item directly in a table slot. This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions.

There are several variants of this approach, collectively referred to as open addressing schemes. Open addressing requires that the load factor is always at most 1 and that items are stored directly in the cells of the bucket array itself.

Linear probing and its variants.

A simple method for collision handling with open addressing is linear probing.

In this approach if we try to insert an item (k, v) into a bucket A[j] that is already occupied, where

Downloaded from EnggTree.com

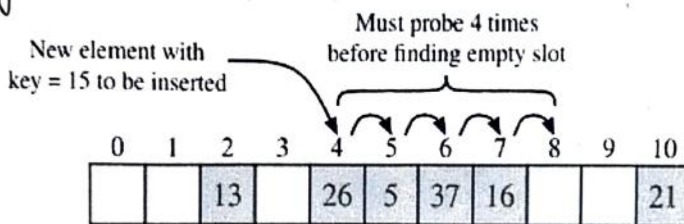
CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

j = h(k) then we next try A[(j+1) mod N]. If A[(j+1) mod N] is also occupied, then we try A[(j+2) mod N] and so on, until we find an empty bucket that can accept the new item. Once this bucket is located we simply insert the item there.

This collision resolution strategy requires that we change the implementation when searching for an existing key, the first step of all getitem, setitem or delitem operations.

In particular, to attempt to locate an item with key equal to k, we must examine consecutive slots starting from A[h(k)], until we either find an item with that key or we find an empty bucket.

(Eg) Insertion into a hash table with integer keys linear probing. The hash function is h(k) = k mod 11.



(Fig 1)

Hash tables.

To implement a deletion, we cannot simply remove a found item from its slot in the array.

For example, after the insertion of key 15 in fig (1) if the item with key 37 were trivially deleted, a subsequent search for 15 would fail because that search would start by probing at index 4, then index 5 and then index 6 at which an empty cell is found. A typical way to get around this difficulty is to replace a deleted item with a special available marker object.

We modify our search algorithm, so that the search for a key k will skip over cells containing the available marker and continue probing until reaching the desired item or an empty bucket. Additionally our algorithm for setitem should remember an available cell encountered during the search for k, since this is a valid place to put a new item (k,v), if no existing item is found.

Open addressing scheme can save space, linear probing suffers tends to cluster the items of a map into contiguous runs, which may even overlap that cause searches to slowdown considerably.

Downloaded from EnggTree.com

EnggTree.com

Linear probing Example.

A list of size 20 (m = 20) we want to put some elements in linear probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}.

x	A[(j+i) mod N] .... i=0,1,2...
96	i=0 A[(96+0) mod 20] = 16.
48	i=0 A[(48+0) mod 20] = 8.
63	i=0 A[(63+0) mod 20] = 3
29	i=0 A[(29+0) mod 20] = 9
87	i=0 A[(87+0) mod 20] = 7
77	i=0 A[(77+0) mod 20] = 17
48	i=0 A[(48+0) mod 20] = 8 i=1 A[(48+1) mod 20] = 9 i=2 A[48+2 mod 20] = 10.
65	i=0 A[(65+0) mod 20] = 5
69	i=0 A[(69+0) mod 20] = 9 i=1 A[(69+1) mod 20] = 10 i=2. A[(69+2) mod 20] = 11
94	i=0 A[(94+0) mod 20] = 14
61	i=0 A[(61+0) mod 20] = 1

Hash Table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
.	61		63		65		87	48	29	48	69			94		96	77		

Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

Quadratic probing.

Quadratic probing iteratively tries the buckets $A[(h(k)+f(i)) \bmod N]$, for $i = 0, 1, 2 \ldots$ where $f(i) = i^2$, until finding an empty bucket. The quadratic probing strategy complicates the removal operation, but it does avoid the kinds of clustering patterns that occur with linear probing. It creates its own kind of clustering called secondary clustering, where the set of filled array cells still has a non-uniform pattern, even if we assume that the original hash codes are distributed uniformly.

An open addressing strategy that does not cause clustering of the kind produced by linear probing or the kind produced by quadratic probing is the double hashing strategy. we choose a secondary hash function h', and if h maps some key k to a bucket $A[h(k)]$ that is already occupied, then we iteratively try the buckets $A[h(k)+f(i)) \bmod N]$ next, for $i = 1, 2, 3 \ldots$ where $f(i) = i \cdot h'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero, a common choice is $h'(k) = q - (k \bmod q)$ for some prime number $q < N$. also N should be prime.

Another approach to avoid clustering with open addressing is to iteratively try buckets $A[(h(k)+f(i)) \bmod N]$ where $f(i)$ is based on a pseudo random number generator providing a repeatable but somewhat arbitrary sequence of subsequent probes that depends upon bits of the original hash code. This approach used in python's dictionary class.

Downloaded from EnggTree.com

EnggTree.com

Quadratic probing Example.

List of size 20 (m = 20). Put some elements in quadratic probing fashion. The elements are {96, 48, 63, 29, 87, 77, 48, 65, 69, 94, 61}

x	A [(h(k)+f(i)) mod N i = 0,1,2 f(i) = i^2
96	i=0 A [(96+0) mod 20] = 16
48	i=0 A [(48+0) mod 20] = 8
63	i=0 A [(63+0) mod 20] = 3
29	i=0 A [(29+0) mod 20] = 9
87	i=0 A [(87+0) mod 20] = 7
77	i=0 A [(77+0) mod 20] = 17
48	i=0 A [(48+0) mod 20] = 8 i=1 A [(48+1^2) mod 20] = 9 i=2 A [(48+2^2) mod 20] = 12
65	i=0 A [(65+0) mod 20] = 5
69	i=0 A [(69+0) mod 20] = 9 A [(69+1^2) mod 20] = 10 ~~A [(69+2^2 mod 20] =~~
94	i=0 A [(94+0) mod 20] = 14.
61	i=0 A [(61+0) mod 20] = 1

Hash Table.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	61		63		5		87	48	29	69		48		94		96.	77			

Downloaded from EnggTree.com

EnggTree.com

Load factors, Rehashing and Efficiency

Load factor is a measure that decides when to increase the Hashtable capacity to maintain the search and insert operation complexity of O(1).

It is defined as $\lambda = n/N$ where N is the total size of the hash table and n is the preferred number of entries which can be inserted before a increment in size of the underlying datastructure is required.

In the hash table schemes $\lambda = n/N$ kept below 1.

with separate chaining as λ gets very close to 1, the probability of a collision greatly increases, which adds overhead to our operations.

Experiments and average-case analyses suggest $\lambda < 0.9$ for hash tables with separate chaining.

With open addressing the load factor λ grows beyond 0.5 and starts approaching 1, clusters of entries in the bucket array start to grow. These clusters causing the probing strategies to "bounce around" the bucket array for a considerable amount of time before they find an empty slot.

Experiments suggest that we should maintain $\lambda < 0.5$ for an open addressing scheme with linear probing and perhaps only a bit higher for other open addressing schemes.

Rehashing.

Rehashing means hashing again, when the load factor increases to more than its pre-defined value.

If an insertion causes the load factor of a hash table to go above the specified threshold, then it is common to resize the table, and to insert all objects into this new table. We need not define a new hash code for each object, we do need to reapply a new compression function that takes into consideration the size of the new table.

Each rehashing will generally scatter the items throughout the new bucket array. When rehashing to a new table, it is a good requirement for the new

Downloaded from EnggTree.com

array's size to be atleast double the previous size. Indeed, if we always double the size of the table with each rehashing operation, then we can amortize the cost of rehashing all the entries in the table against the time used to insert them in the first place.

Efficiency of Hash tables.

To store n entries, the expected number of keys in a bucket would be [n/N], which is O(1) if n is O(N)

The cost associated with a periodic rehashing to resize a table after occasional insertions or deletions can be accounted for separately, leading to an additional O(1) amortized cost for setitem & getitem

In the worst case, a poor hash function could map every item to the same bucket this would result in linear-time performance for the core map operations with separate chaining or with any open addressing model in which the secondary sequence of probes depends only on the hash code.

Downloaded from EnggTree.com

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

EnggTree.com

Unit IV
Tree Structures.

Tree ADT - Binary Tree ADT - Tree traversals - Binary search trees - AVL trees - Heaps - multiway search trees.

Tree Definitions and Properties.

A tree is an Abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. We typically call the top element the root of the tree.

Formal Tree Definition.

A Tree T as a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties.

* If T is nonempty, it has a special node, called the root of T, that has no parent.
* Each node v of T different from the root has a unique parent node w, every node with parent w is a child of w.

Other Node Relationships.

Two nodes that are children of the same parent are siblings. A node v is external if v has no children. A node v is internal if it has one or more children. External nodes are also known as leaves.

A node u is an ancestor of a node v if u=v or u is an ancestor of the parent of v. Conversely, the node v is descendant of a node u if u is an ancestor of v.

The subtree of T rooted at a node v is the tree consisting of all the descendants of v in T

Downloaded from EnggTree.com

Edges and paths in Trees.

An edge of tree T is a pair of nodes (u, v) such that u is the parent of v or vice versa.

A path of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.

Ordered Trees.

A tree is ordered if there is a meaningful linear order among the children of each node (ie) identify the children of a node as being the first, second, third and so on.

The Tree Abstract Data type.

We define a tree ADT using the concept of a position as an abstraction for a node of a tree. An element is stored at each position, and positions satisfy parent-child relationships that define the tree structure.

A position object for a tree supports the method.

p.element() : Return the element stored at position p.

The tree ADT then supports the following accessor methods, allowing a user to navigate the various positions of a tree.

T.root() : Return the position of the root of Tree T, or None if T is empty.

T.is_root(p) : Return True if position p is the root of Tree T

T.parent(p) : Return the position of the parent of position p, or None if p is the root of T

T.num_children(p) : Return the number of children of position p.

T.children(p) : Generate an iteration of the children of position p

T.is_leaf(p) : Return True if position p does not have any children.

len(T) : Return the number of positions that are contained in Tree T

T.is_empty() : Return True if tree T does not contain any positions.

Downloaded from EnggTree.com

EnggTree.com

Binary Trees.

A binary tree is an ordered tree with the following properties:

* Every node has atmost two children.
* Each childnode is labeled as being either a left child or a right child.
* A left child precedes a right child in the order of children of a node.

The subtree rooted at a left or right child of a internal node v is called a left subtree or right subtree respectively of v.

A binary tree is proper (or full binary trees) if each node has either zero or two children.

A binary tree is not proper is improper.

A Recursive Binary Tree Definition.

A binary Tree is either empty or consists of

* A node r, called the root of T, that stores an element
* A binary tree (possibly empty), called the left subtree of T
* A binary tree (possibly empty) called the right subtree of T

The Binary Tree Abstract Data Type

As an ADT, a binary tree is a specialization of a tree that supports three additional accessor methods.

T.left(P) : Return the position that represents the left child of P, or None if P has no left child.

T.right(P) : Return the position that represents the right child of P, or None if P has no right child.

T.Sibling(P) : Return the position that represents the sibling of P, or None if P has no sibling.

The Binary Tree Abstract Base class in python.

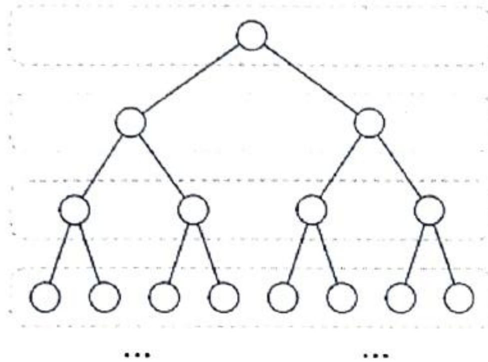
We define a new binary Tree class associated with the binary tree ADT. We rely on inheritance to define the BinaryTree class based upon the existing Tree class.

Downloaded from EnggTree.com

EnggTree.com

Properties of Binary Trees.

Binary trees have properties dealing with relationships between their heights and number of nodes. We denote the set of all nodes of a tree T at the same depth d as leve d of T. In a binary tree, level 0 has at most one node (root), level 1 has atmost two nodes, level 2 has at most four nodes and so on. In general, level d has atmost 2^d nodes.



We can see that the maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree.

Proposition:

Let T be a nonempty binary tree, and let n, n_E, n_I & h denote the number of nodes, number of external nodes, number of internal nodes and height of T respectively. Then T has the following properties.

1. $h+1 \le n \le 2^{h+1} - 1$
2. $1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1) - 1 \le h \le n-1$

Also if T is proper, then T has the following properties.

1. $2h+1 \le n \le 2^{h+1} - 1$
2. $h+1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1) - 1 \le h \le (n-1)/2$

Downloaded from EnggTree.com

EnggTree.com

Relating Internal nodes to External nodes in a proper binary Tree.

Proposition:

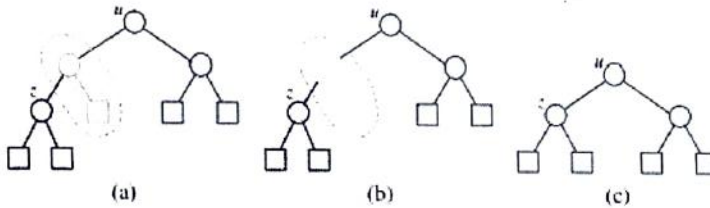
In a nonempty proper binary tree T, with n_E external nodes and n_I internal nodes, we have $n_E = n_I + 1$

Justification.

We justify this proposition by removing nodes from T and dividing them up into two piles an internal node pile and external node pile, until T becomes empty. We consider two cases.

Case 1: If T has only one node v, we remove v and place it on the external-node pile. Thus the external-node pile has one node and the internal node pile is empty.

Case 2: Otherwise T has more than one node, we remove from T an external node w and its parent v, which is an internal node.


(a) (b) (c)

Downloaded from EnggTree.com

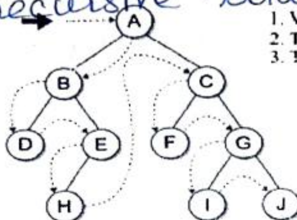
EnggTree.com

Tree Traversals.

Tree traversal iterates through a collection, one item at a time, in order to access or visit each item.

Preorder Traversal.

A tree traversal must begin with the root node, since that is the only access into the tree. After visiting the root node, we can then traverse the nodes in its left subtree followed by the nodes in its right subtree. Since every node is the root of its own subtree and, we can repeat the same process on each node, resulting in a recursive solution.



The logical ordering of nodes with a preorder Traversal.

Consider the binary tree. The dashed lines show the logical order the nodes would be visited during the traversal.

Algorithm.

```
def preordertrav(subtree):
    if subtree is Not None:
        Print(subtree.data)
        preordertrav(subtree.left)
        preordertrav (subtree.right)
```

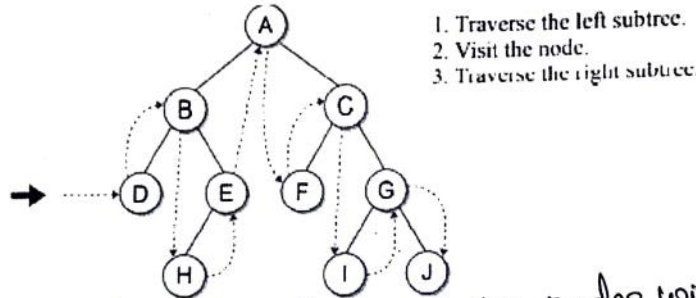
The subtree argument will either be a null reference or a reference to the root of a subtree in the binary tree. If the reference is not None, the node is first visited and then the two sub trees are traversed. The subtree argument will be a null reference when the binary tree is empty or we attempt to follow a non existent link for one or both of the children.

Downloaded from EnggTree.com

EnggTree.com

Inorder Traversal.

Inorder traversal which we first traverse the left subtree and then visit the node followed by the traversal of the right subtree.



The logical ordering of the nodes with an inorder traversal.

Algorithm.

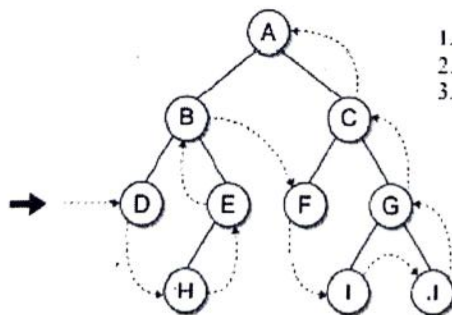
```
def inordertrav(subtree):
    if subtree is not None:
        inordertrav(subtree.left)
        print(subtree.data)
        inordertrav(subtree.right).
```

Postorder traversal.

In a postorder traversal, which can be viewed as the left and right subtrees of each node are traversed before the node is visited.

Algorithm.

```
def postordertrav(subtree):
    if subtree is not None:
        postordertrav(subtree.left)
        postordertrav(subtree.right)
        print(subtree.data).
```



The logical ordering of the nodes with a postorder traversal

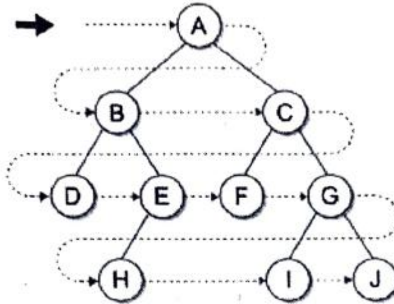
Downloaded from EnggTree.com

EnggTree.com

Breadth - First traversal.

The preorder, inorder, and postorder traversals are the examples of depth first traversal.

In breadth first traversal, the nodes are visited by level from left to right.



The logical ordering of the nodes with a BFT

Recursion cannot be used to implement a breadth-first traversal, since the recursive calls must follow the links that lead deeper into the tree.

We use queue to implement breadth-first traversal. The process starts by saving the root node and in turn priming the iterative loop. During each iteration, we remove a node from the queue, visit it and then add its children to the queue. The loop terminates after all nodes have been visited.

```
def breadthfirsttrav(bintree):
    Queue q
    q.enqueue(bintree).
    while not q.isempty():
        node = q.dequeue()
        print(nod.data).
        if node.left is not None:
            q.enqueue(node.left)
        if node.right is not None:
            q.enqueue(node.right).
```

Downloaded from EnggTree.com

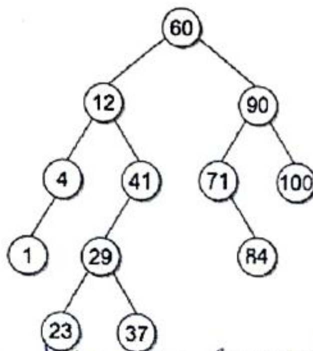
EnggTree.com

Binary Search Tree.

A binary search tree (BST) is a binary Tree in which each node contains a search key within its payload and the tree is structured such that for each interior node V.

* All keys less than the key in node V are stored in the left subtree of V
* All keys greater than the key in node V are stored in the right subtree of V.

A Binary search trees storing integer search keys.



Consider the binary search tree, which contains integers search keys. The root node contains key value 60 and all keys in the root's left subtree are less than 60 and all of the keys in the right subtree are greater than 60.

Partial implementation of the map ADT - BST

```
class BSTMap:
    def __init__(Self):
        self._root = None
        Self._size = 0
    def __len__(Self):
        return self._size
    def __iter__(Self):
        return BSTMapIterator(Self._root)
```

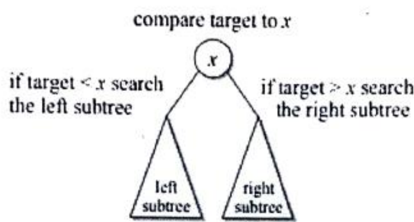
```
class BSTMapNode:
    def __init__(Self, key, Value):
        Self.key = key
        Self.Value = Value
        Self.left = None
        Self.right = None
```

Downloaded from EnggTree.com

Searching.

In a BST if we want to search the tree to determine if it contains a given key or to locate a specific element. If the binary search Tree contains the target key, then there will be a unique path from the root to the node containing that key. If the root contains the target value, our search is over, otherwise if the target is not in the root, we must decide which of two possible paths to take. We know that key in the root node is larger than the keys in its leftsubtree and smaller than the keys in its right subtree. Thus if the target is less than the roots key we move left and we move right if its greater. We repeat the comparison on the root node of the subtree and take the appropriate Path. This process is repeated until target is located or we encounter a null child link.

414



The structure of BST is based on the search keys.

Suppose we want to search for key value 29 in the BST from. we begin by comparing the target to 60. Since the target is less than 60, we move left. The target is then compared to 12. This time we move right since the target is larger than 12. Next the target is compared to 41, resulting in a move to the left. Finally when we examine the left child of node 41, we find the target and report a successful search.

If the target is not in the tree, then we continue the search which will fall off the tree, thus reaching a null child link during the search for a target key indicates an unsuccessful search.

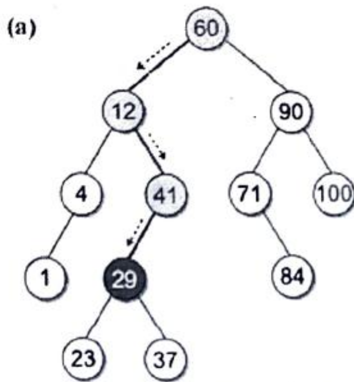
The BST operations can be implemented iteratively or with the use of recursion. The recursive method has two base cases. The target is contained in the current node or a null child link is encountered. when a base case is reached, the method

Downloaded from EnggTree.com CHENNAI INSTITUTE DEPARTMENT OF SCIENCE AND HUMANITIES

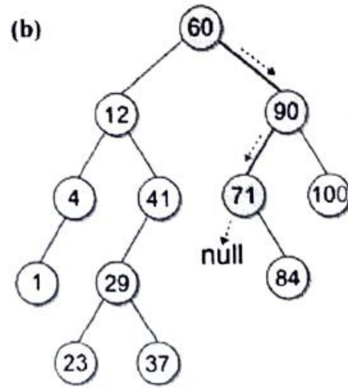
EnggTree.com

returns either a reference to the node containing the key or None, back through all of the recursive calls. The latter indicates the key was not found in the tree. The recursive call is made by passing the link to either the left or right subtree depending on the relationship between the target and the key in the current node.

Searching a BST



successful search



unsuccessful search.

Algorithm.

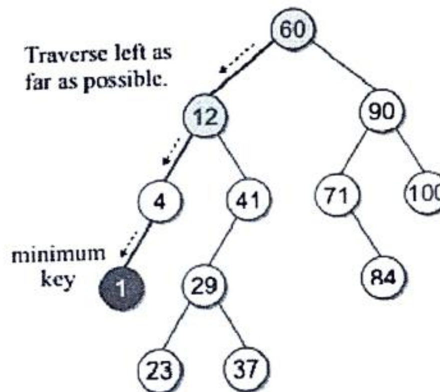
```
class BSTMap :
# ...
   # Determines if the map contains the given key.
  def __contains__( self, key ):
    return self._bstSearch( self._root, key ) is not None

   # Returns the value associated with the key.
  def valueOf( self, key ):
    node = self._bstSearch( self._root, key )
    assert node is not None, "Invalid map key."
    return node.value

   # Helper method that recursively searches the tree for a target key.
  def _bstSearch( self, subtree, target ):
    if subtree is None :          # base case
      return None
    elif target < subtree.key :  # target is left of the subtree root.
      return self._bstSearch( subtree.left )
    elif target > subtree.key :  # target is right of the subtree root.
      return self._bstSearch( subtree.right )
    else :                        # base case
      return subtree
```

Downloaded from EnggTree.com

EnggTree.com

Min and Max Values.

To find the min and max value is similar to a search that can be performed on a binary search tree is finding the minimum or maximum key values. In BST we know the minimum value is either in the root or in a node to its left. suppose the root node does not have a left child, in this case the root would contain the smallest key value, since all the keys to the right are larger than the root. In the same method the maximum key value can be found.



Finding Minimum key in a BST

```
class BSTMap :
# ...
  # Helper method for finding the node containing the minimum key.
  def _bstMinumum( self, subtree ):
    if subtree is None :
      return None
    elif subtree.left is None :
      return subtree
    else :
      return self._bstMinimum( subtree.left )
```

Downloaded from EnggTree.com

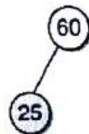
EnggTree.com

Insertions.

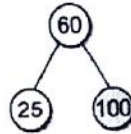
when a binary search tree is constructed, the keys are added one at a time. As the keys are inserted, a new node is created for each key and linked into its proper position with in the tree. suppose we want to build a binary search tree from the keylist [60, 25, 100, 35, 17, 80] by inserting the keys in the order they are listed.

Building a binary tree by inserting the keys [60, 25, 100, 35, 17, 80]

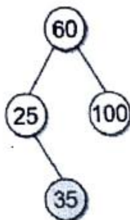
(a) Insert 60.

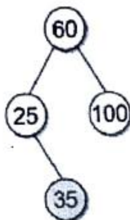


(b) Insert 25.

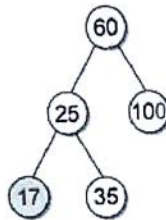


(c) Insert 100.

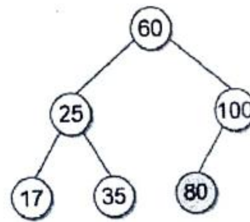




(d) Insert 35.



(e) Insert 17.

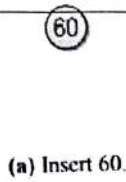


(f) Insert 80.

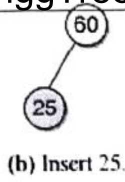
we start by inserting value 60. A node is created and its datafield set to that value. Since the tree is initially empty, this first node becomes the root of the tree. Next we insert value 25, since it is smaller than 60, it has to be inserted to the left of the root, which means it becomes the left child of th root. next we insert 100 in the right of the root, next we need to insert 35, the root already has both its left and right children. when new keys are inserted, we do not modify the data fields of existing nodes or the links between existing nodes. Thus 35 is inserted into our current tree and still maintain the search tree property.

working through this example, if we want to insert the new keys for eg 30 into the tree we built. Now it calls bstSearch() method and search for key 30, The search will lead us to node 35 and then fall off the tree when attempting to follow its leftchild link.

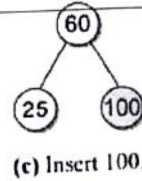
Downloaded from EnggTree.com



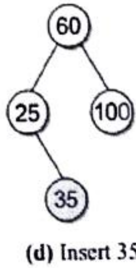
(a) Insert 60.



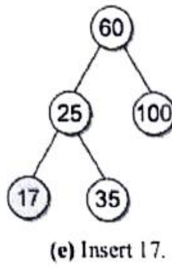
(b) Insert 25.



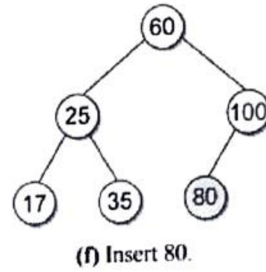
(c) Insert 100.



(d) Insert 35.

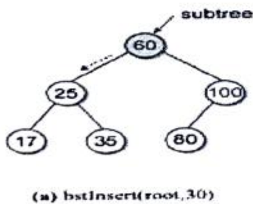


(e) Insert 17.

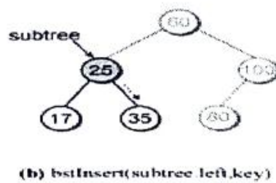


(f) Insert 80.

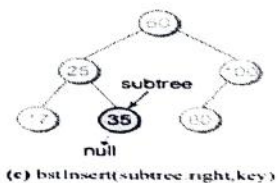
```
class BSTMap :
# ...
  # Adds a new entry to the map or replaces the value of an existing key.
  def add( self, key, value ):
    # Find the node containing the key, if it exists.
    node = self._bstSearch( key )
    # If the key is already in the tree, update its value.
    if node is not None :
      node.value = value
      return False
    # Otherwise, add a new entry.
    else :
      self._root = self._bstInsert( self._root, key, value )
      self._size += 1
      return True

  # Helper method that inserts a new item, recursively.
  def _bstInsert( self, subtree, key, value ):
    if subtree is None :
      subtree = _BSTMapNode( key, value )
    elif key < subtree.key :
      subtree.left = self._bstInsert( subtree.left, key, value )
    elif key > subtree.key :
      subtree.right = self._bstInsert( subtree.right, key, value )
    return subtree
```



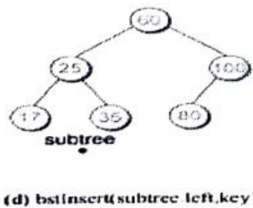
(a) bstInsert(root,30)



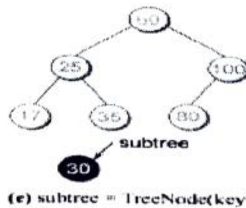
(b) bstInsert(subtree.left,key)



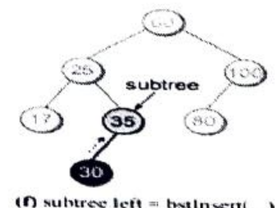
(c) bstInsert(subtree.right,key)



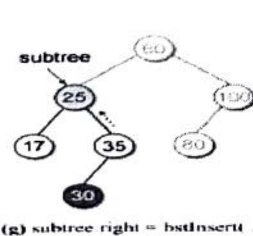
(d) bstInsert(subtree.left,key)



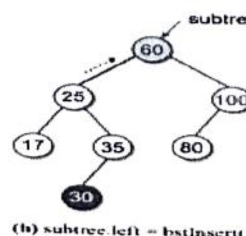
(e) subtree = TreeNode(key)



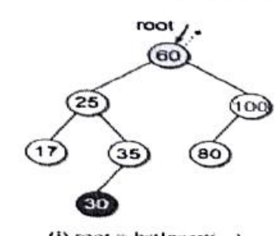
(f) subtree.left = bstInsert(...)



(g) subtree.right = bstInsert(...)



(h) subtree.left = bstInsert(...)



(i) root = bstInsert(...)

Downloaded from EnggTree.com

EnggTree.com

Deletions.

Removing an element from a BST is a bit more complicated than searching for an element or inserting a new element into the tree. A deletion involves searching for the node that contains the target key and then unlinking the node to remove it from the tree.

There are three cases to consider once the node has been located.

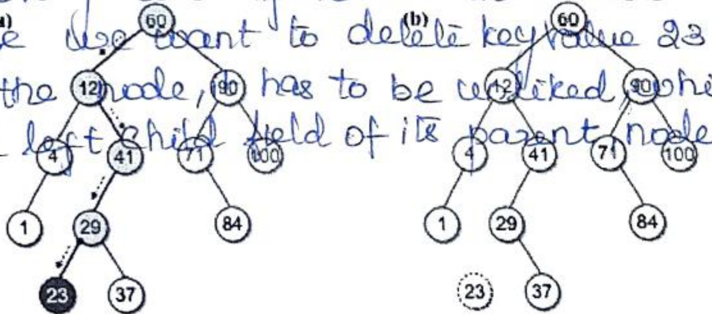
* The node is a leaf
* The node has a single child.
* The node has two children.

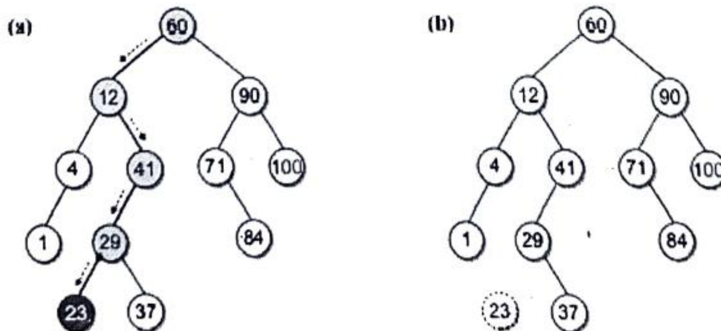
The first step in removing an element is to find the node that contains the key. We search the location to delete a element. Once the node is located, it has to be unlinked to remove it from the tree.

The bstremove() method is used to remove the nodes.

Removing a leafnode.

Removing a leaf node is the easiest among the three cases. Suppose we want to delete key value 23 from the BST, after finding the node, it has to be unlinked, which can be done by setting the left child field of its parent node 29 to None.





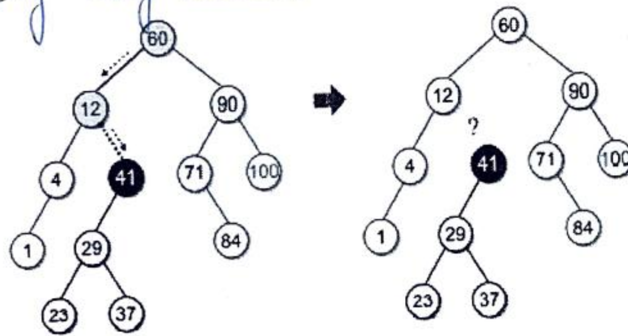
Downloaded from EnggTree.com

EnggTree.com

Removing an Interior Node with one child.

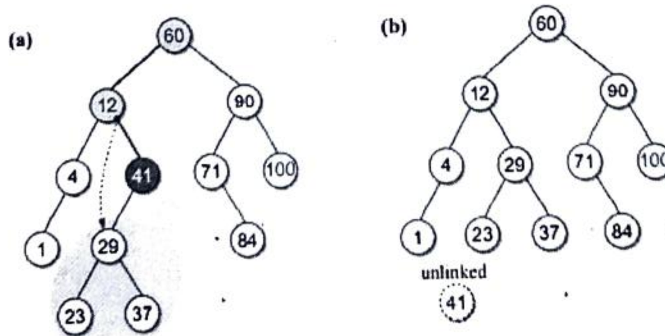
If the node to be removed has a single child, it can be either the left or right child. Suppose we want to delete key value 41 from the BST, the node 41 has a subtree linked as the left child. If we were to simply return None back to the parent, the node 41 be removed, but we would also lose all of its descendants..

To remove node 41, we will have to do something with its descendants. Since Node 41 contains a single child, all of its descendants will either have keys that are smaller than 41 or all of them will be larger. Node 41 is the right child of node 12, all of the descendants of node 41 must also be larger than 12, thus we can set the link in the right child field of node 12 to reference node 29. Now node 29 becomes right child of node 12 and all of the descendants of node 41 will be properly linked without losing any nodes.



Incorrectly unlinking the interior node.

being deleted. Selecting the child field of the parent to change is automatically handled by the assignment performed upon return of the recursive call. All we have to do is return the appropriate child link in the node being deleted.



Downloaded from EnggTree.com

EnggTree.com

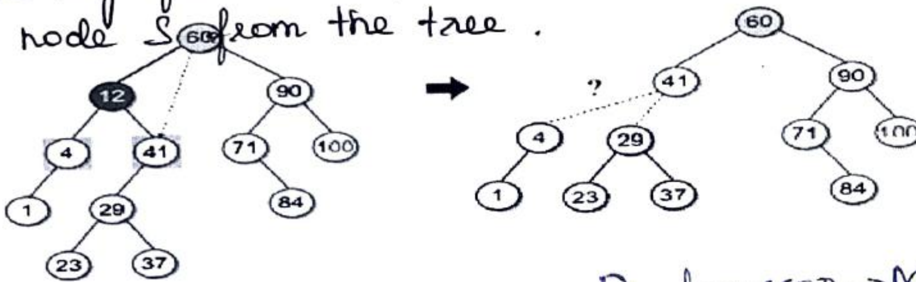
Removing an Interior Node with Two Children.

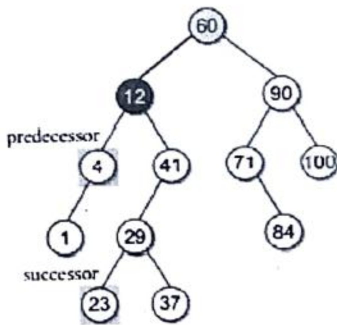
The most difficult case is when the node to be deleted has two children. suppose we want to remove node 12 from the BST, node 12 has two children, both of which are the roots of their own subtrees.

The keys in a BST are arranged such than an inorder traversal produces a sorted key sequence. Thus each node has a logical predecessor and successor. For node 12, its predecessor is node 4 and its successor is node 23.

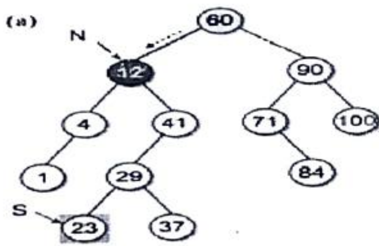
Removing an interior node with two children require 3 steps.

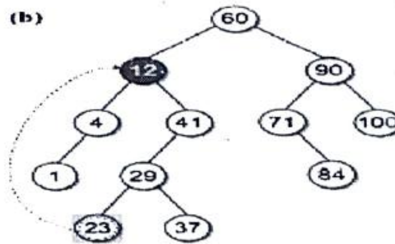
* Find the logical successor S, of the node to be deleted, N.
* Copy the key from node S to node N.
* Remove node S from the tree.

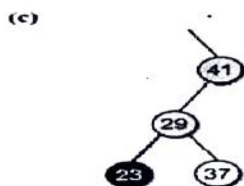


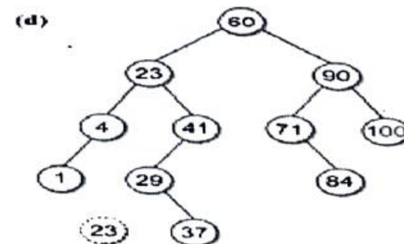


Predecessor → Maximum value in the left subtree.
successor → Minimum value in its right subtree.









Downloaded from EnggTree.com

EnggTree.com

```
class BSTMap :
# ...
  # Removes the map entry associated with the given key.
  def remove( self, key ):
    assert key in self, "Invalid map key."
    self._root = self._bstRemove( self._root, key )
    self._size -= 1

  # Helper method that removes an existing item recursively.
  def _bstRemove( self, subtree, target ):
    # Search for the item in the tree.
    if subtree is None :
      return subtree
    elif target < subtree.key :
      subtree.left = self._bstRemove( subtree.left, target )
      return subtree
    elif target > subtree.key :
      subtree.right = self._bstRemove( subtree.right, target )
      return subtree
    # We found the node containing the item.
    else :
      if subtree.left is None and subtree.right is None :
        return None
      elif subtree.left is None or subtree.right is None :
        if subtree.left is not None :
          return subtree.left
        else :
          return subtree.right
      else
        successor = self._bstMinimum( subtree.right )
        subtree.key = successor.key
        subtree.value = successor.value
        subtree.right = self._bstRemove( subtree.right, successor.key )
        return subtree
```

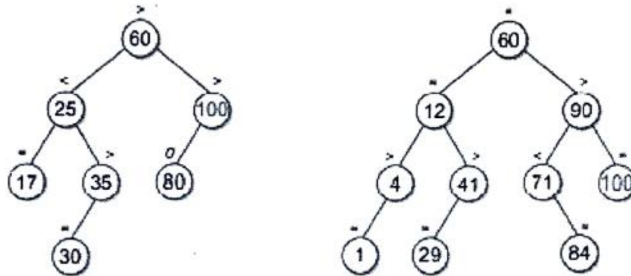
Downloaded from EnggTree.com

EnggTree.com

AVL Trees.

The Binary search Tree provides a convenient structure for storing and searching data collections. The efficiency of the search, insertion and deletion operations depend on the height of the tree.

The AVL tree, which was invented by G.M. Adelson Velskii & Y.M. Landis in 1962, improves on the BST by always guaranteeing the tree is height balanced, which allows for more efficient operations. A binary tree is balanced if the heights of the left and right subtrees of every node differ by at most 1



with each node in an AVL tree, we associate a balance factor, which indicates the height difference between the left and right branch. The balance factor can be one of three states.

lefthigh: when the leftsubtree is higher than the right subtree.
equalhigh: when the two subtrees have equal height.
right high: when the right subtree is higher than the left subtree.

The balance factor are illustrated by the symbols >, =, < [> → for a left high state, = → for the equal high state < = for the right high state]

The search and traversal operations are the same with an AVL tree as with a BST. The insertion and deletion operations have to be modified in order to maintain the balance property of the tree as newkeys are inserted and existing one removed.

The height is sufficient for providing O(log n) time operations even in the worst case.

Downloaded from EnggTree.com

EnggTree.com

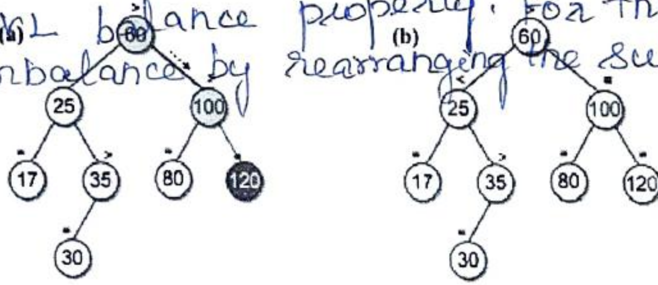
Insertions.

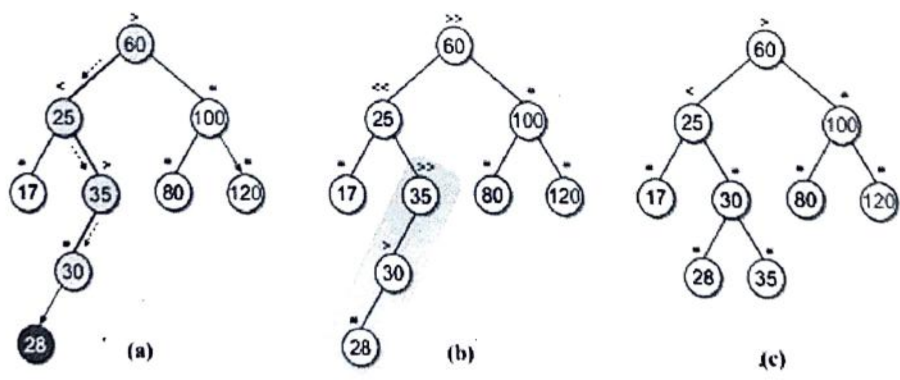
Inserting a key into a AVL tree begins with the same as BST. When a new key is inserted into an AVL tree, the balance property of the tree must be maintained. If the insertion of the new key causes any of the subtrees to become unbalanced, they will have to be rebalanced.

Some insertions are simple. For example, suppose we want to add key 120 to the sample AVL tree, the key is inserted as the right child of node 100. The tree remains balanced since the insertion does not change the height of any subtree but it does cause a change in the balance factors.

After the key is inserted, the balance factors have to be adjusted in order to determine if any subtree is out of balance.

If we add key 28 to the AVL, the new node is inserted as the left child of nod 30, when the balance factors are recalculated, we can see all the subtrees along the path that above node 30 are now out of balance, which violates the AVL balance property. For this example we can correct the imbalance by rearranging the subtree rooted at node 35.





Downloaded from EnggTree.com

Rotations.

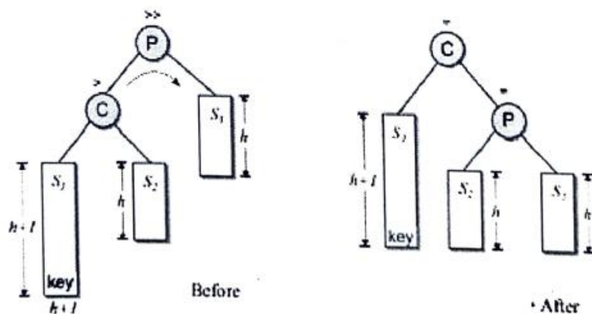
Multiple subtree can become unbalanced after inserting a new key, all of which have roots along the insertion path. But only one will have to be rebalanced, the one deepest in the tree and closest to the new node. After inserting the key, the balance factors are adjusted during the unwinding of the recursion. The first subtree encountered that is out of balance has to be rebalanced. The root node of this subtree is known as the pivot node.

An AVL subtree is rebalanced by performing a rotation around the pivot node. This involves rearranging the links of the pivot node, its children, and possibly one of its grandchildren.

There are 4 possible cases.

case 1:

This case when the balance factor of the pivot node (P) is left high before the insertion and the new key is inserted into the left child (C) of the pivot node. To rebalance the subtree the pivot node has to be rotated right over its left child. The rotation is accomplished by changing the links such that P becomes the right child of C and the right child of C becomes the left child of P.



Downloaded from EnggTree.com

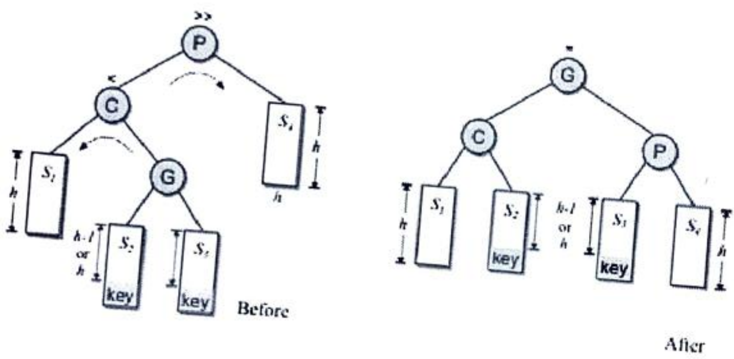
EnggTree.com

Case 2:

This case involves three nodes. The pivot (P), the left child of the pivot (C), and the right child (G) of C. For this case to occur, the balance factor of the pivot is left high before the insertion and the new key is inserted into either the right subtree of C. It requires 2 rotations.

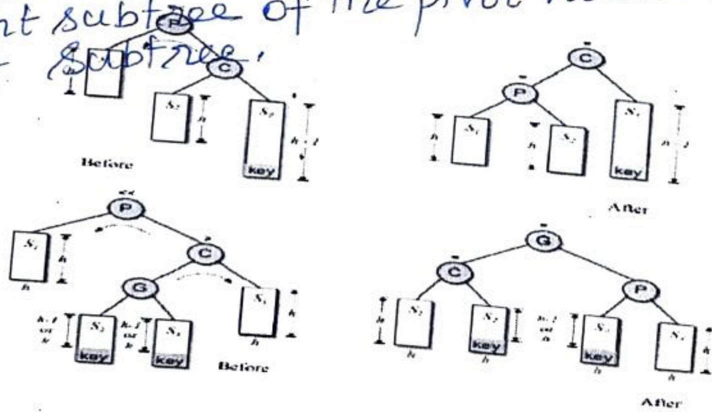
Node C has to rotated left over node V and the pivot node has to be rotated right over its left child.

The link modifications required to accomplish this rotation include setting the right child of G as the new left child of the pivot node, changing the left child of G to become the right child of C, and setting C to the new left child of G.



Case 3 and 4:

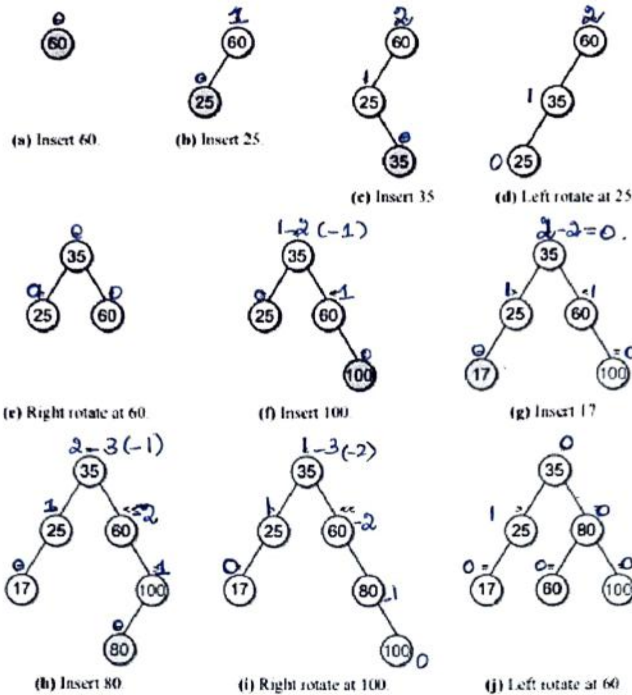
The third case is a mirror image of the first case and the fourth case is a mirror image of the second case. The difference is the new key is inserted in the right subtree of the pivot node or a descendant of its right subtree.



Downloaded from EnggTree.com

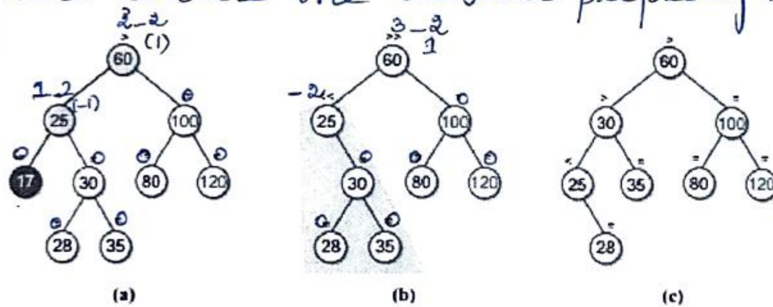
EnggTree.com

Construction of an AVL Tree (60, 25, 35, 100, 17, 80)



Deletions.

when an entry is removed from an AVL tree, we must ensure the balance property is maintained.



For example if we remove the key 17 from the AVL tree, the tree is unbalanced. A left rotation has to be performed pivoting on node 25 to correct the imbalance.

CHENNAI INSTITUTE OF TECHNOLOGY DEPARTMENT OF SCIENCE AND HUMANITIES

Downloaded from EnggTree.com

Implementation.

```
# Constants for the balance factors.
LEFT_HIGH = 1
EQUAL_HIGH = 0
RIGHT_HIGH = -1

# Implementation of the Map ADT using an AVL tree.
class AVLMap :
  def __init__( self ):
    self._root = None
    self._size = 0

  def __len__( self ):
    return self._size

  def __contains__( self, key ):
    return self._bstSearch( self._root, key ) is not None

  def add( self, key, value ):
    node = self._bstSearch( key )
    if node is not None :
      node.value = value
      return False
    else :
      (self._root, tmp) = self._avlInsert( self._root, key, value )
      self._size += 1
      return True

  def valueOf( self, key ):
    node = self._bstSearch( self._root, key )
    assert node is not None, "Invalid map key."
    return node.value

  def remove( self, key ):
    assert key in self, "Invalid map key."
    (self._root, tmp) = self._avlRemove( self._root, key )
    self._size -= 1

  def __iter__( self ):
    return _BSTMapIterator( self._root )

# Storage class for creating the AVL tree node.
class _AVLMapNode :
  def __init__( self, key, value ):
    self.key = key
    self.value = value
    self.bfactor = EQUAL_HIGH
    self.left = None
    self.right = None
```

Helper functions to performing the AVL tree rotations.

```
class AVLMap :
# ...
  # Rotates the pivot to the right around its left child.
  def _avlRotateRight( self, pivot ):
    C = pivot.left
    pivot.left = C.right
    C.right = pivot
    return C

  # Rotates the pivot to the left around its right child.
  def _avlRotateLeft( self, pivot ):
    C = pivot.right
    pivot.right = C.left
    C.left = pivot
    return C
```


Downloaded from EnggTree.com

Helper function used to rebalance AVL subtrees.

```
class AVLMap :
# ...
  # Rebalance a node when its left subtree is higher.
  def _avlLeftBalance( self, pivot ):
    # Set L to point to the left child of the pivot.
    C = pivot.left

    # See if the rebalancing is due to case 1.
    if C.bfactor == LEFT_HIGH :
      pivot.bfactor = EQUAL_HIGH
      C.bfactor = EQUAL_HIGH
      pivot = _avlRotateRight( pivot )
      return pivot

    # Otherwise, a balance from the left is due to case 3.
    else :
      # Change the balance factors.
      if G.bfactor == LEFT_HIGH :
        pivot.bfactor = RIGHT_HIGH
        C.bfactor = EQUAL_HIGH
      elif G.bfactor == EQUAL_HIGH :
        pivot.bfactor = EQUAL_HIGH
        C.bfactor = EQUAL_HIGH
      else : # G.bfactor == RIGHT_HIGH
        pivot.bfactor = EQUAL_HIGH
        C.bfactor = LEFT_HIGH

      # All three cases set G's balance factor to equal high.
      G.bfactor = EQUAL_HIGH

      # Perform the double rotation.
      pivot.left = _avlRotateLeft( L )
      pivot = _avlRotateRight( pivot )
      return pivot
```

Insert an entry into an AVL tree.

```
class AVLMap :
# ...
  # Recursive method to handle the insertion into an AVL tree. The
  # function returns a tuple containing a reference to the root of the
  # subtree and a boolean to indicate if the subtree grew taller.
  def _avlInsert( self, subtree, key, newitem ):
    # See if we have found the insertion point.
    if subtree is None :
      subtree = _AVLTreeNode( key, newitem )
      taller = True

    # Is the key already in the tree?
    elif key == subtree.data :
      return (subtree, False)

    # See if we need to navigate to the left.
    elif key < subtree.data :
      (subtree, taller) = _avlInsert( subtree.left, key, newitem )
      # If the subtree grew taller, see if it needs rebalancing.
      if taller :
        if subtree.bfactor == LEFT_HIGH :
          subtree.right = _avlLeftBalance( subtree )
          taller = False
        elif subtree.bfactor == EQUAL_HIGH :
          subtree.bfactor = LEFT_HIGH
          taller = True
        else :     # RIGHT HIGH
          subtree.bfactor = EQUAL_HIGH
          taller = False
```


Downloaded from EnggTree.com

```
# Otherwise, navigate to the right.
else key > subtree.data :
  (node, taller) = _avlInsert( subtree.right, key, newitem )
   # If the subtree grew taller, see if it needs rebalancing.
  if taller :
    if subtree.bfactor == LEFT_HIGH :
      subtree.bfactor = EQUAL_HIGH
      taller = False
    elif subtree.bfactor == EQUAL_HIGH :
      subtree.bfactor = RIGHT_HIGH
      taller = True
    else :    # RIGHT_HIGH
      subtree.right = _avlRightBalance( subtree )
      taller = False

# Return the results.
return (subtree, taller)
```


Downloaded from EnggTree.com

EnggTree.com

Heaps

Heap datastructure allows us to perform both insertions and removals in logarithmic time.

The Heap Data structure.

A heap is a binary tree T that stores a collection of items at its positions and that satisfies relational property and structural property.

relational property - It defined in terms of the way keys are stored in T

structural property - It defined in terms of the shape of T itself.

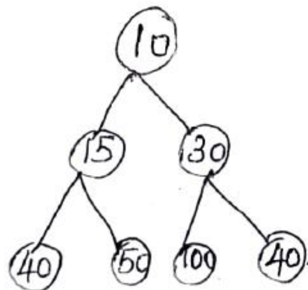
Heap order property.

In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent.

As a consequence of the heap-order property, the keys encountered on a path from the root to a leaf of T are in nondecreasing order. Also a minimum key is always stored at the root of T.

Complete Binary Tree property.

A heap T with height h is a complete binary tree if levels 0, 1, 2, ... h-1 of T have the maximum number of nodes possible and the remaining nodes at level h reside in the leftmost possible positions at that level.



Downloaded from EnggTree.com

EnggTree.com

The height of a heap.

Let h denote the height of T. Insisting that T be complete also has an important consequence.

A heap T storing n entries has height $h = \lfloor \log n \rfloor$.

Justification.

T is complete. The number of nodes in levels 0 through h-1 of T is precisely $1+2+4+\dots+2^{h-1} = 2^h - 1$, and that the number of nodes in level h is at least 1 and at most 2^h. Therefore

$n \geq 2^h - 1 + 1 = 2^h$ and $n \leq 2^h - 1 + 2^h = 2^{h+1} - 1$.

By taking the logarithm of both sides

$2^h \leq n$, height $h \leq \log n$.

by rearranging the terms and taking algorithms of both sides of inequality $n \leq 2^{h+1} - 1$ we see that

$\log(n+1) - 1 \leq h$.

h is an integer, these two inequalities imply that

$h = \lfloor \log n \rfloor$

The heap, which is in the form of a tree, is stored in the array, and its elements are indexed in the following order.

* The root element will be at the 0th position of the list (ie) heap[0]
* For any other node, say heap[i], we have the following
  - The parent node is given by heap[(i-1)//2]
  - The leftchild node is given by heap[(2*i)+1]
  - The right-child node is given by heap[(2*i)+2]

Max Heap: In a max-Heap the key present at the root node must be greatest among the key present at all of its children. The same property must be recursively true for all subtrees in that Binary Tree.

Downloaded from EnggTree.com

EnggTree.com

MinHeap: In a min-heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all subtrees in that BinaryTree.

Construction of Max heap.

10, 51, 2, 18, 4, 31, 13, 5, 23, 64, 29

① Insert the first value

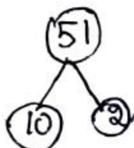
(10)

2. Insert 51

51 ≤ 10 (swap)
(?)

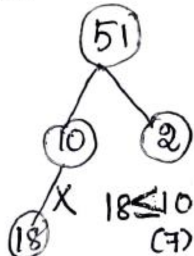
Insert 2

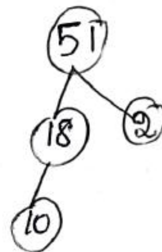
③



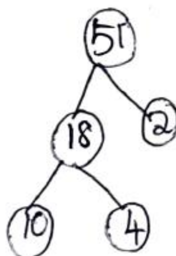
2 ≤ 51

④ Insert 18

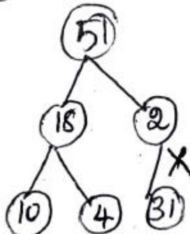




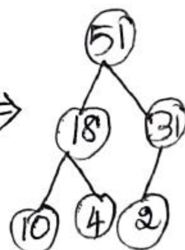
⑤ Insert 4



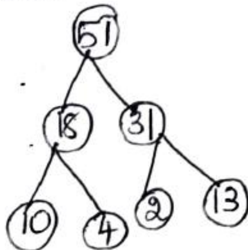
⑥ Insert 31



31 ≤ 2 swap ⇒
(?)

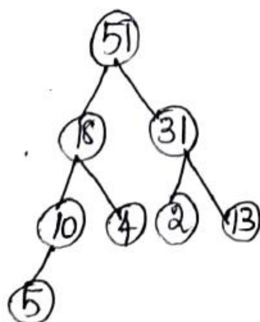


⑦ Insert 13



13 ≤ 31

⑧ Insert 5

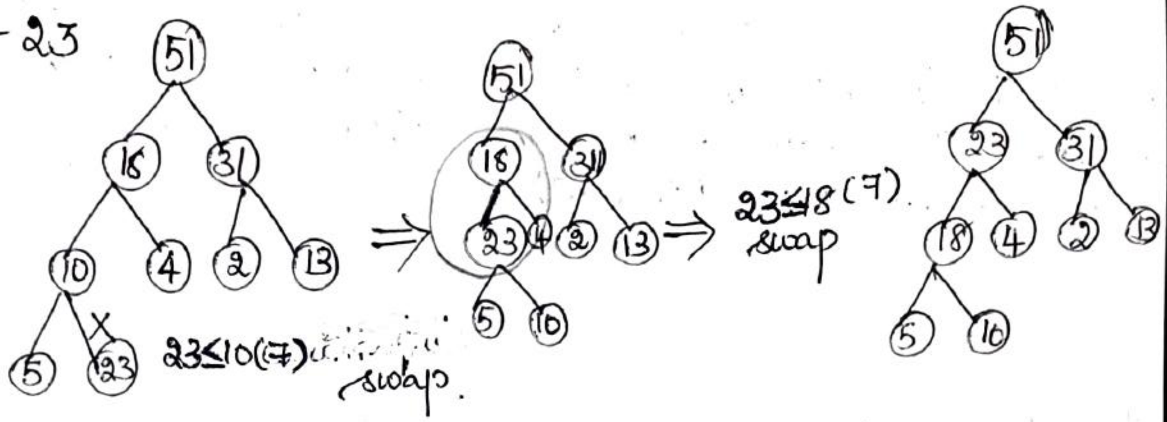


5 ≤ 10.

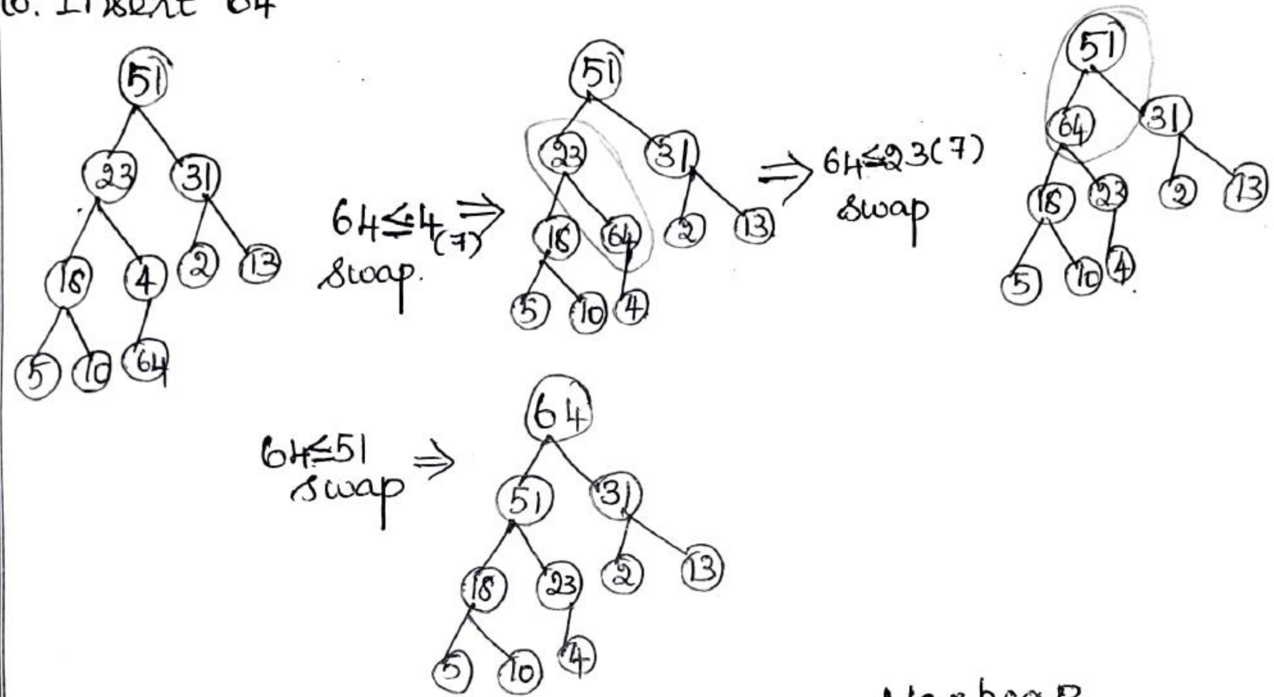
Downloaded from EnggTree.com

EnggTree.com

9. Insert 23

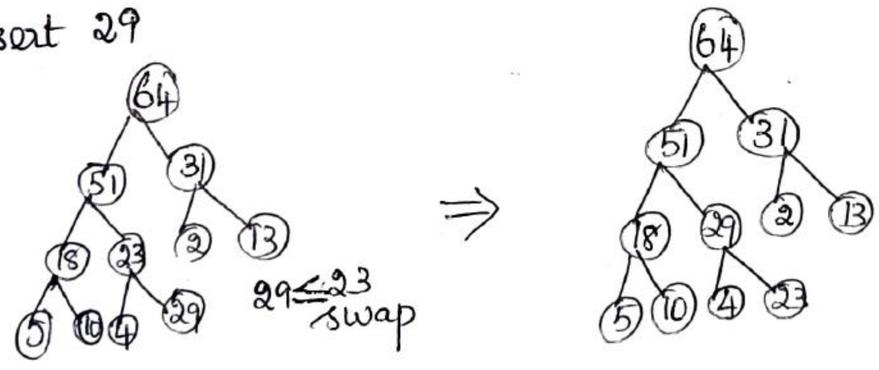


10. Insert 64



Maxheap

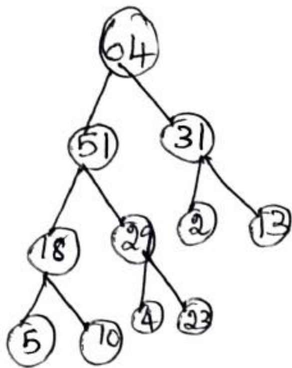
11 Insert 29



Downloaded from EnggTree.com

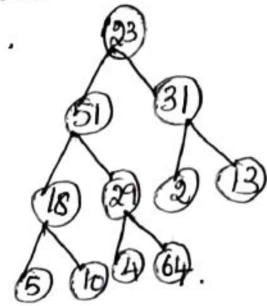
EnggTree.com

Deletions.



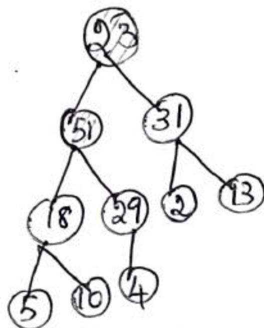
Delete 64

Step 1. swap the first and last items in the heap.



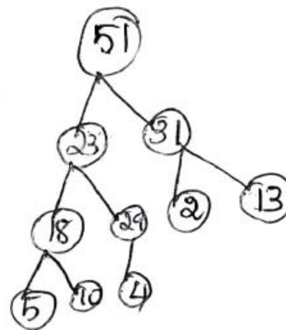
Step 2

Remove the last item from the heap

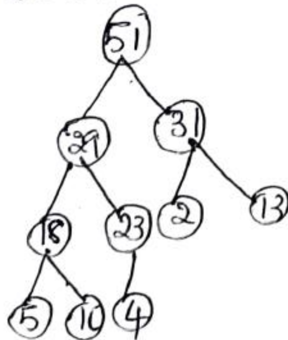


Implement the heap property in all the nodes.

51 ≤ 23 (fails swap)



29 ≤ 23 (fails, swap)



Downloaded from EnggTree.com

EnggTree.com

```
class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a binary heap."""
  #------------------------------ nonpublic behaviors ------------------------------
  def _parent(self, j):
    return (j-1) // 2

  def _left(self, j):
    return 2*j + 1

  def _right(self, j):
    return 2*j + 2

  def _has_left(self, j):
    return self._left(j) < len(self._data)      # index beyond end of list?

  def _has_right(self, j):
    return self._right(j) < len(self._data)     # index beyond end of list?

  def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

  def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
      self._swap(j, parent)
      self._upheap(parent)                      # recur at position of parent

  def _downheap(self, j):
    if self._has_left(j):
      left = self._left(j)
      small_child = left                        # although right may be smaller
      if self._has_right(j):
        right = self._right(j)
        if self._data[right] < self._data[left]:
          small_child = right
      if self._data[small_child] < self._data[j]:
        self._swap(j, small_child)
        self._downheap(small_child)             # recur at position of small child
```

Downloaded from EnggTree.com

EnggTree.com

```
#------------------------------ public behaviors ------------------------------
def __init__(self):
  """Create a new empty Priority Queue."""
  self._data = [ ]

def __len__(self):
  """Return the number of items in the priority queue."""
  return len(self._data)

def add(self, key, value):
  """Add a key-value pair to the priority queue."""
  self._data.append(self._Item(key, value))
  self._upheap(len(self._data) − 1)          # upheap newly added position

def min(self):
  """Return but do not remove (k,v) tuple with minimum key.

  Raise Empty exception if empty.
  """
  if self.is_empty( ):
    raise Empty('Priority queue is empty.')
  item = self._data[0]
  return (item._key, item._value)

def remove_min(self):
  """Remove and return (k,v) tuple with minimum key.

  Raise Empty exception if empty.
  """
  if self.is_empty( ):
    raise Empty('Priority queue is empty.')
  self._swap(0, len(self._data) − 1)          # put minimum item at the end
  item = self._data.pop( )                    # and remove it from the list;
  self._downheap(0)                           # then fix new root
  return (item._key, item._value)
```


Downloaded from EnggTree.com

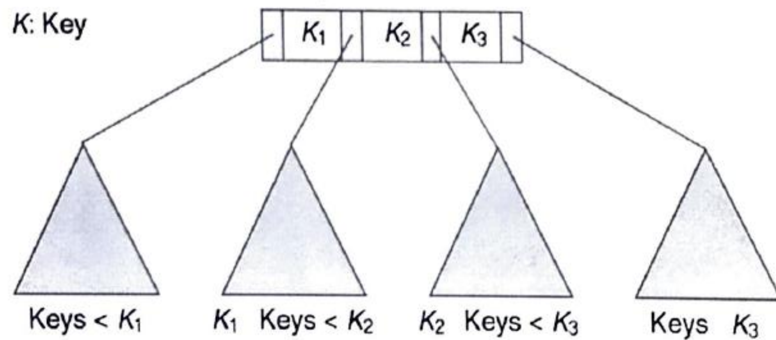
EnggTree.com

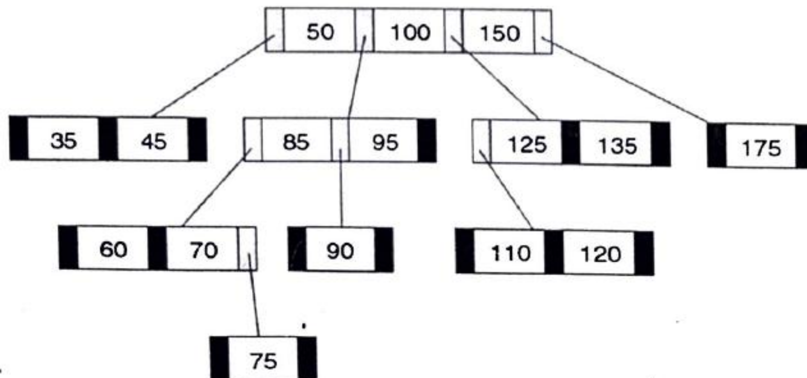
Multiway search Trees.

The multiway search trees are generalised versions of binary trees, where each node contains multiple elements. In an m-way tree of order m, each node contains a maximum of m-1 elements and m children.

Properties.

1. Each node has 0, 1. . . . m subtrees.
2. A node with k<m subtrees, Contain k-1 keys.
3. The key values of the first subtree are all less than the key value.
4. The data entries are ordered
5. All subtrees are m-way trees.





Downloaded from EnggTree.com

EnggTree.com

Unit V

Graph ADT – Representation of graph – Graph traversals – DAG – Topological ordering – shortest paths – Minimum spanning Tree.

Graph ADT.

A graph is a collection of vertices and edges we model the abstraction as a combination of three data types: Vertex, Edge and Graph. Vertex (V) is a set of nodes, Edge (E) is a collection of pair of vertices.

The graph ADT includes the following methods.

* Setvertex
* getvertex
* add Edge
* getEdges

Applications of Graphs.

* Representing relationships between components in electronic circuits
* Transportation network – highway N/w, Flight N/w.
* computer N/w – LAN, Internet
* Databases – Representing ER diagram.

Graph Representation.

* Adjacency Matrix
* Adjacency list
* Adjacency set

Adjacency matrix.

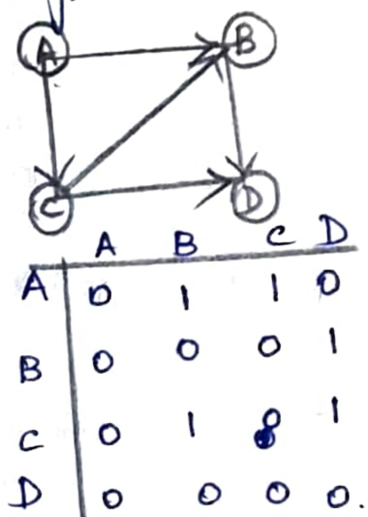
To represent graphs, we need the number of vertices, the number of edges and also their interconnections

Downloaded from EnggTree.com

EnggTree.com

Adjacency Matrix Representation

One simple way to represent a graph is to use a 2D array. This is known as Adjacency matrix representation. For each edge (u,v), we set A[u][v] to True. Otherwise the entry in the array is false. (eg).



	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	1	0	1
D	0	0	0	0.

```
class Graph(object):
    # Initialize the matrix
    def __init__(self, size):
        self.adjMatrix = []
        for i in range(size):
            self.adjMatrix.append([0 for i in range(size)])
        self.size = size

    # Add edges
    def add_edge(self, v1, v2):
        if v1 == v2:
            print("Same vertex %d and %d" % (v1, v2))
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    # Remove edges
    def remove_edge(self, v1, v2):
        if self.adjMatrix[v1][v2] == 0:
            print("No edge between %d and %d" % (v1, v2))
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v1] = 0

    def __len__(self):
        return self.size

    # Print the matrix
    def print_matrix(self):
        for row in self.adjMatrix:
            for val in row:
                print('{:4}'.format(val),end='')
            print("\n")

def main():
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.add_edge(4,1)
    g.print_matrix()


if __name__ == '__main__':
    main()
```

Downloaded from EnggTree.com

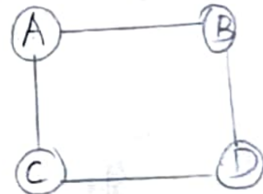
EnggTree.com

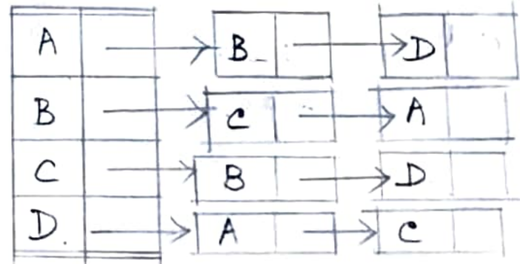
Adjacency list representation.

It represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

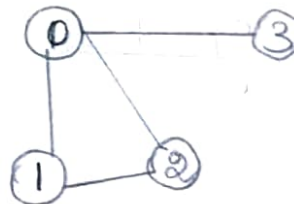
```
# Adjascency List representation in Python
class AdjNode:
    def __init__(self, value):
        self.vertex = value
        self.next = None

class Graph:
    def __init__(self, num):
        self.V = num
        self.graph = [None] * self.V

    # Add edges
    def add_edge(self, s, d):
        node = AdjNode(d)
        node.next = self.graph[s]
        self.graph[s] = node
        node = AdjNode(s)
        node.next = self.graph[d]
        self.graph[d] = node

    # Print the graph
    def print_agraph(self):
        for i in range(self.V):
            print("Vertex " + str(i) + ":", end="")
            temp = self.graph[i]
            while temp:
                print(" -> {}".format(temp.vertex), end="")
                temp = temp.next
            print(" \n")
if __name__ == "__main__":
    V = 5
    # Create graph and edges
    graph = Graph(V)
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(0, 3)
    graph.add_edge(1, 2)
    graph.print_agraph()
```

(Eg)





O/P
vertex 0: → 3 → 2 → 1
Vertex 1 : → 2 → 0
Vertex 2: → 1 → 0
Vertex 3: → 0
Vertex 4 :



Downloaded from EnggTree.com

EnggTree.com

Adjacency set.

It is very much similar to adjacency list, but instead of using linked list, Disjoint sets are used.

Graph Traversals.

To solve problems on graphs, we need a mechanism for traversing the graphs. It can be classified by 2 types
* Depth first search (DFS)
* Breadth first search (BFS)

Depth first search (DFS)

DFS is an algorithm used to traverse or locate a target node in a graph or tree datastructure. It priorities depth and searches along one branch as far as it go until the end of that branch. Once there it backtracks to the first possible divergence from that branch and searches until the end of that branch, repeating the process.

Algorithm.

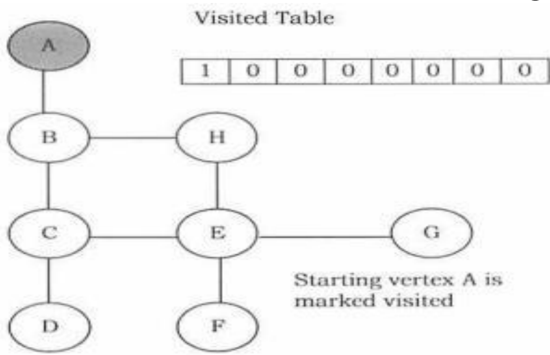
```
def dfs(visited, graph, node):
    if node not in visited:
        print(node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
```

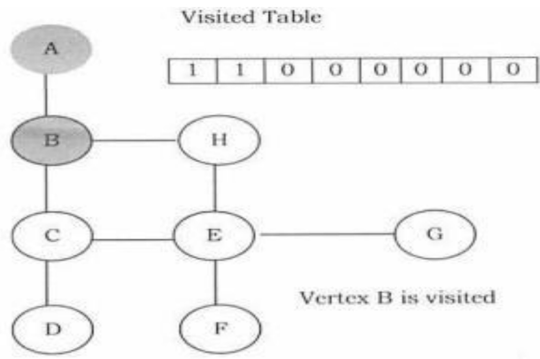
As an example, consider the following graph.

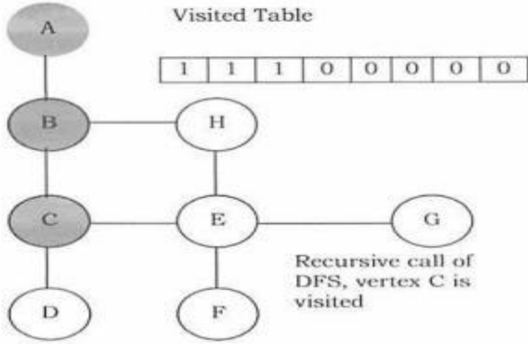
The final generated tree is called the DFS tree and the order in which the vertices are processed is called DFS numbers of the vertices.

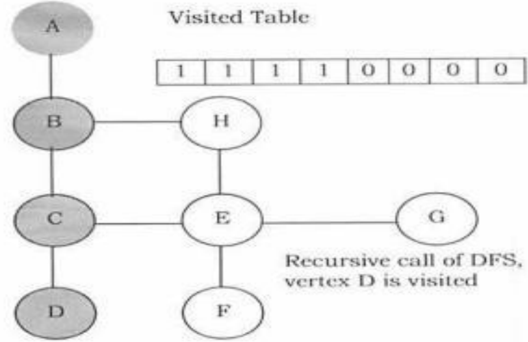
Downloaded from EnggTree.com

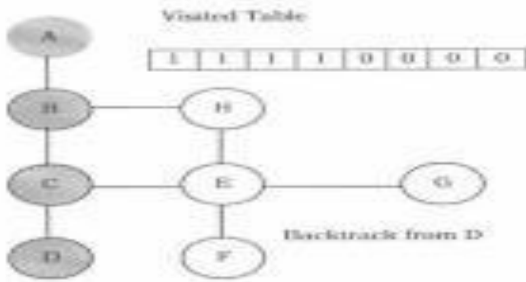
EnggTree.com

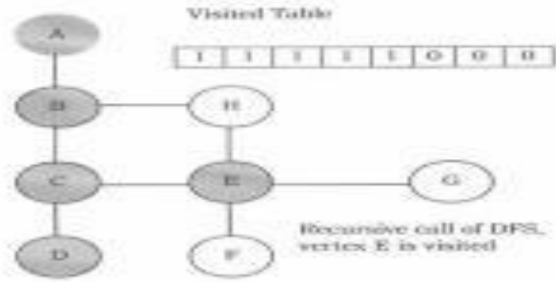


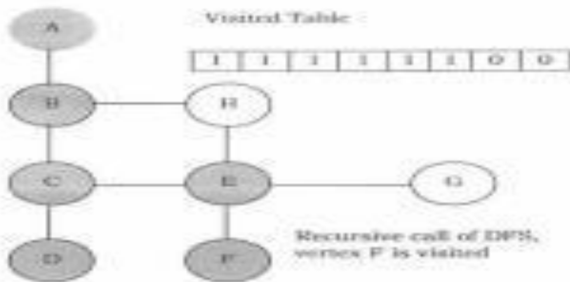


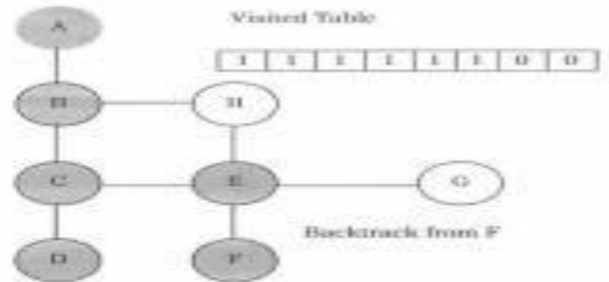


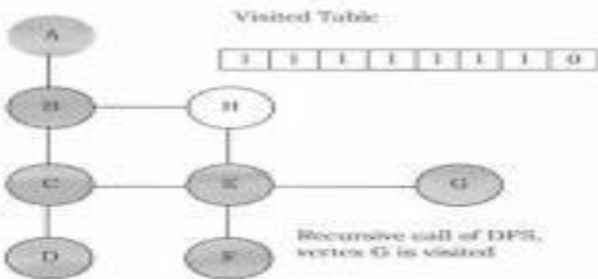


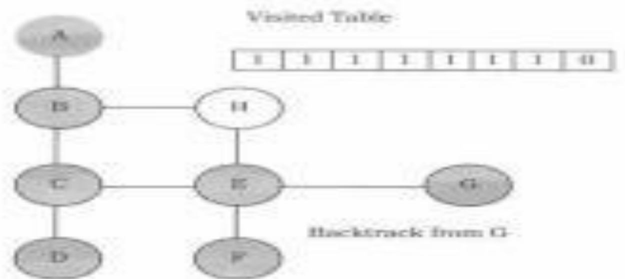




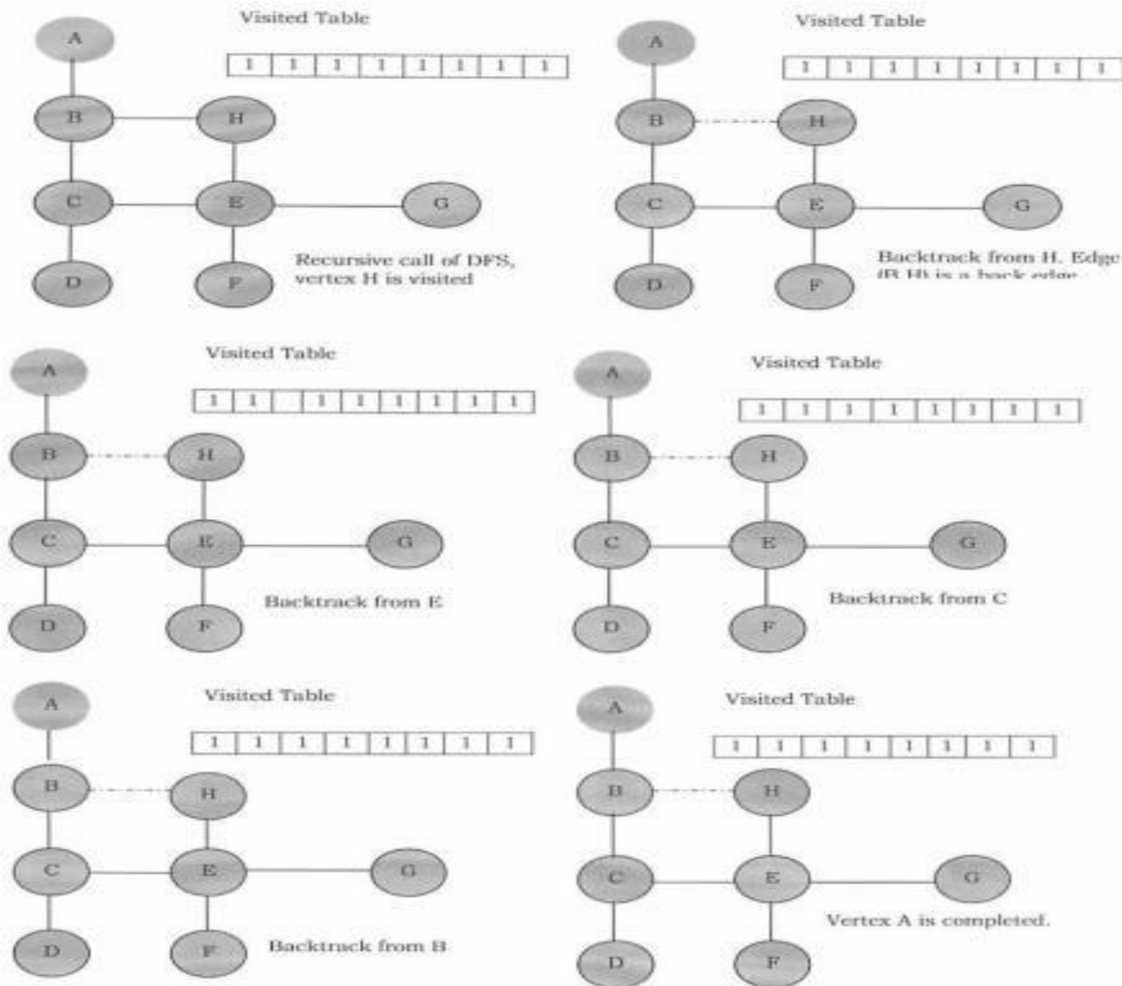








Downloaded from EnggTree.com



From the above diagrams, it can be seen that the DFS traversal creates a tree and we call such tree as DFS tree.

The time complexity of DFS is O(V+E), if we use adjacency list for representing the graphs.

If an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can't be found efficiently, and this gives $O(V^2)$ complexity.

Applications of DFS

* Topological sorting
* Finding connected components
* Finding articulations points (cut vertices) of the graph.
* Solving puzzles and mazes.

Downloaded from EnggTree.com

Breadth first search (BFS)

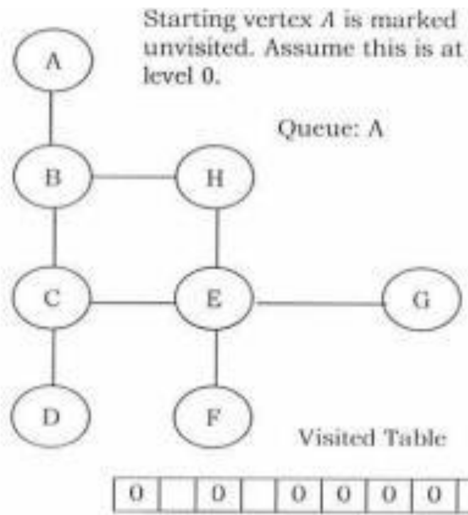
BFS is the process of traversing each node of the graph into two parts visited, Not visited. The purpose of the algorithm is to visit all the vertex while avoiding cycles. BFS uses a queue to represent the visited nodes.

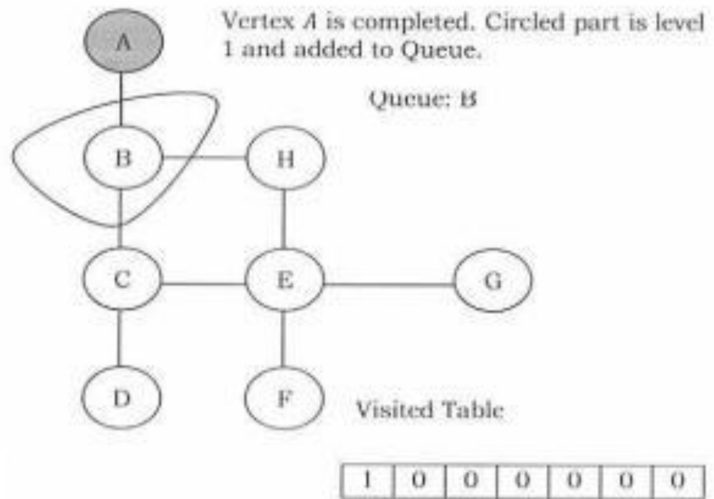
BFS works level by level. Initially BFS starts at a given vertex which is at level 0. In the first stage it visits all vertices at level 1 and so on. BFS continues this process until all the levels of the graph are completed.

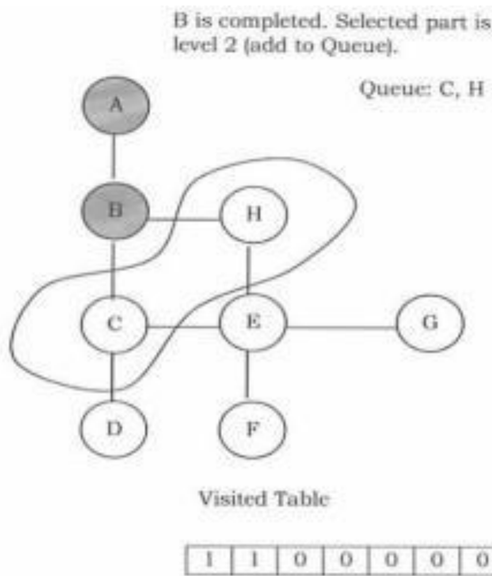
Algorithm

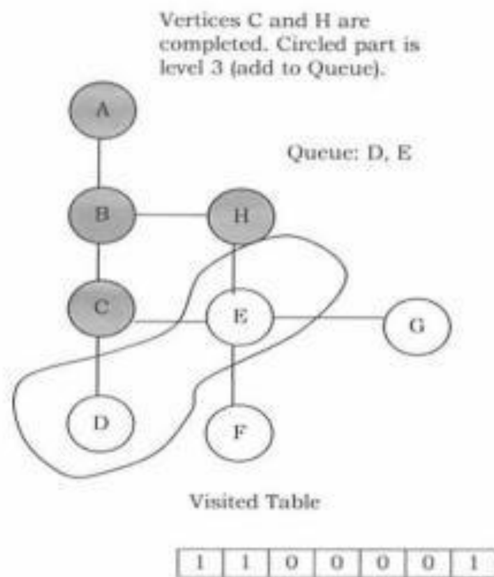
```
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:  # creating loop to visit each node
        m = queue.pop(0)
        print(m, end=" ")
        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour).
```

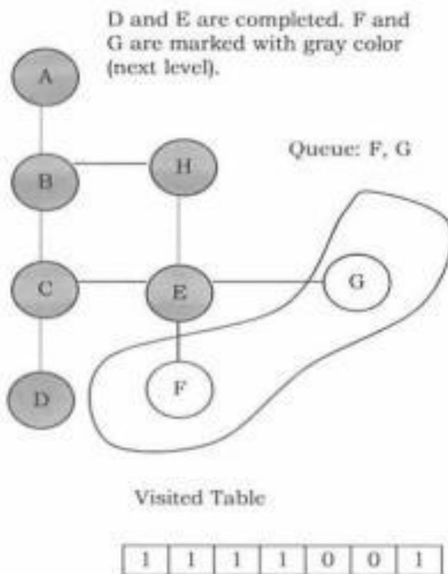
Downloaded from EnggTree.com

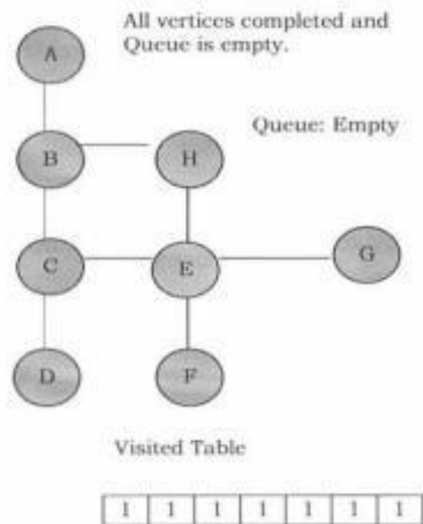












Downloaded from EnggTree.com

EnggTree.com

Topological Sort.

A topological sort is an ordering of vertices in a directed acylic graph, such that if there is a path from Vi to Vj, then Vj appears after Vi in the ordering.

In the graph in figure 1, V1, V2, V5, V4, V3, V7, V6 and V1, V2, V5, V4, V7, V3, V6 are both topological orderings.

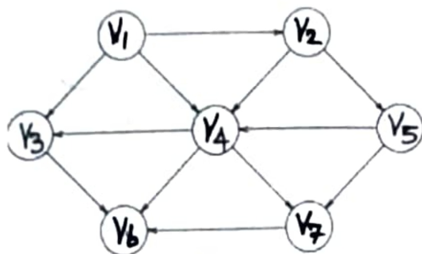
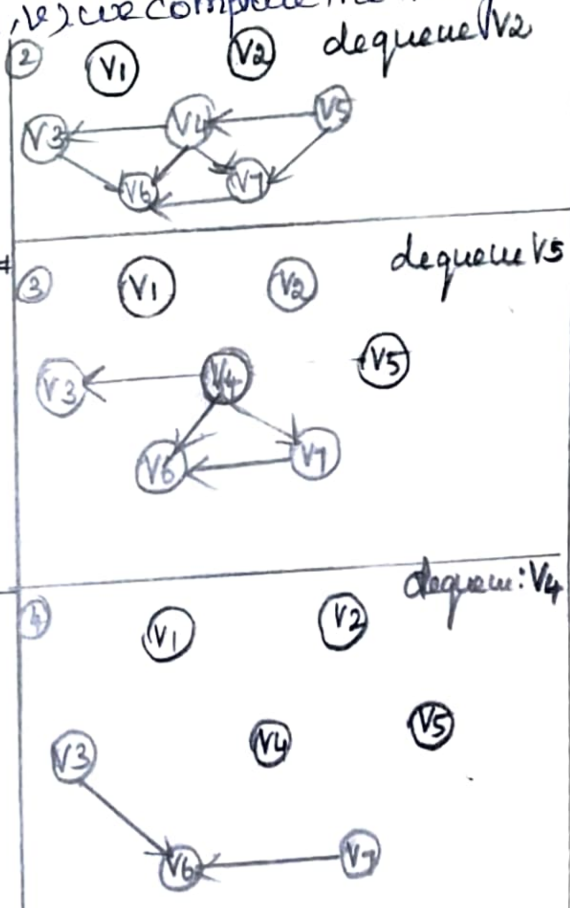


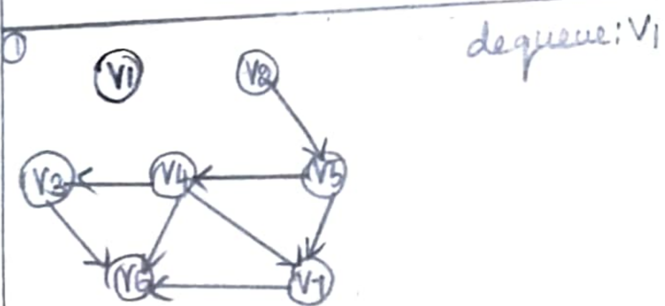
Fig 1.

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges. We can then print this vertex, and remove it, along with its edges from the graph. The same strategy is applied to the rest of the graph.

To formalize this, we define the indegree of a vertex v as the number of edges (u, v). We compute the indegrees of all vertices in the graph.



	Indegree before dequeued #						
Vertex	1	2	3	4	5	6	7
V1	0	0	0	0	0	0	0
V2	1	0	0	0	0	0	0
V3	2	1	1	1	0	0	0
V4	3	2	1	0	0	0	0
V5	1	1	0	0	0	0	0
V6	3	3	3	3	2	1	0
V7	2	2	2	1	0	0	0
Enqueue	V1	V2	V5	V4	V3, V7		V6
Dequeue	V1	V2	V5	V4	V3	V7	V6



Downloaded from EnggTree.com

```
def topologicalsort(G):
    topologicallist=[]
    topologicalqueue=[]
    remainingindegree={}
    nodes=G.getvertices()
    for v in G:
        indegree=v.getIndegree()
        if indegree==0:
            topologicalqueue.append(v)
        else:
            remainingindegree[v]=indegree
    while len(topologicalqueue):
        node=topologicalqueue.pop(0)
        topologicallist.append(node)
    while len(topologicallist):
        node=topologicallist.pop(0)
        print node.getvertexId()
```

Total running time of topological sort is O(V+E)

⑤ dequeue: V7

⑥ dequeue: V6

Downloaded from EnggTree.com

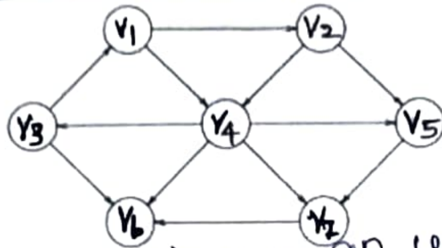
EnggTree.com

Shortest-Path Algorithms

A graph G=(V,E) and a distinguished vertex s, we need to find the shortest path from s to every other vertex in G, is defined by single-source shortest path problem.

* Shortest path in unweighted graph
* Shortest path in weighted graph.

Shortest path in unweighted graph.



The above diagram shows an unweighted graph G. Using some vertex s, which is an input parameter, we would like to find the shortest path from s to all other vertices. We are only interested in the number of edges contained on the path, so there are no weights on the edges. We could assign all edges a weight of 1.

We choose source node S to be V3, we can tell that the shortest path from S to V3 is then a path of length 0. This shows the fig 1

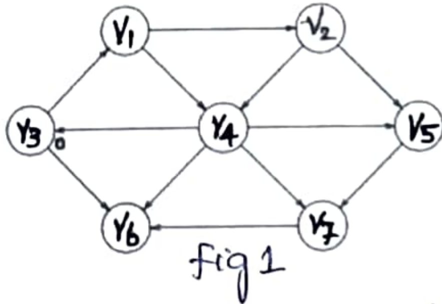


fig 1

V	Known	dV	Pv
V1	F	∞	0
V2	F	∞	0
V3	F	0	0
V4	F	∞	0
V5	F	∞	0
V6	F	∞	0
V7	F	∞	0

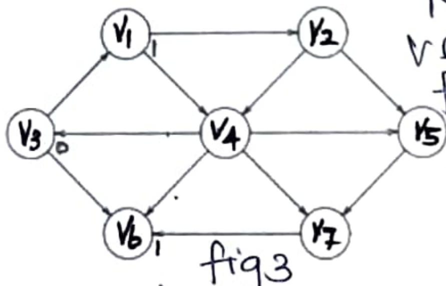


fig 3

Now we can start looking for all vertices that are a distance 1 away from S. These can be found by looking at the vertices that are adjacent to S. V1 and V6 are one edge from S. Shown in fig 2.

Now find vertices whose shortest path from S is exactly 2, by finding all the vertices adjacent to V1 and V6 whose shortest path are not already known.

Downloaded from EnggTree.com

EnggTree.com

This search tells us that the shortest path to reach V4 is 2. fig 3 shows the progress that has been made so far.

Finally we find by examining vertices adjacent to the recently evaluated V2 and V4 that V5 and V7 have a shortest path of three edges.

Fig 4 shows the final result of the algorithm.

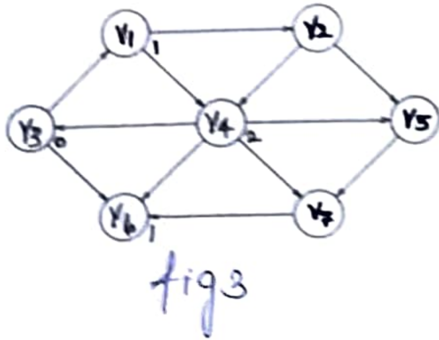


fig3

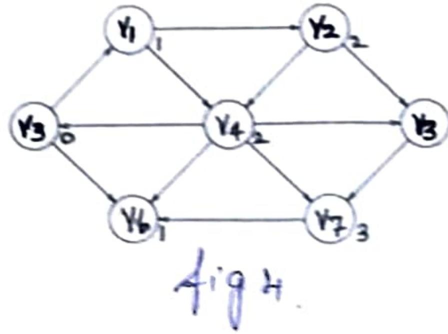


fig 4.

	Initial state			V3 Dequeued			V1 Dequeued			V6 Dequeued		
V	Known	dv	pv	Known	dv	pv	Known	dv	pv	Known	dv	pv
V1	F	∞	0	F	1	V3	T	1	V3	T	1	V3
V2	F	∞	0	F	∞	0	F	2	V1	F	2	V1
V3	F	0	0	T	0	0	T	0	0	T	0	0
V4	F	∞	0	F	∞	0	F	2	V1	F	2	V1
V5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
V6	F	∞	0	F	1	V3	F	1	V3	T	1	V3
V7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q		V3			V1, V6			V6, V2, V4			V2, V4	

	V2 Dequeued			V4 Dequeued			V5 Dequeued			V7 Dequeued		
V	Known	dv	pv	Known	dv	pv	Known	dv	pv	Known	dv	pv
V1	T	1	V1	T	1	V3	T	1	V3	T	1	V3
V2	T	2	V1	T	2	V1	T	2	V1	T	2	V1
V3	T	0	0	T	0	0	T	0	0	T	0	0
V4	F	2	V1	T	2	V1	T	2	V1	T	2	V1
V5	F	3	V2	F	3	V2	T	3	V2	T	3	V2
V6	T	1	V3	T	1	V3	T	1	V3	T	1	V3
V7	F	∞	0	F	3	V4	F	3	V4	T	3	V4
Q		V4, V5			V5, V7			V7			empty	

Algorithm.

```
def bfs():
    queue=[]
    self.start.visited=True
    queue.append(self.start)
    while queue:
        currentnode=queue.pop(0)
        for node in currentnode.neighbors:
            if not node.visited
                node.visited=True
                queue.append(node)
                node.prev=currentnode
                if node=self.end:
                    queue.clear()
                    break;
    Self.trace rou
```

Downloaded from EnggTree.com

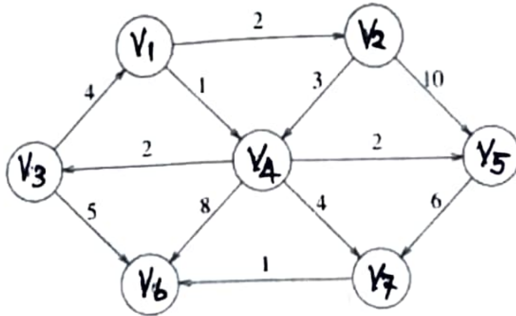
EnggTree.com

Shortest path in weighted graph [Dijkstra's Algorithm]

The shortest path in weighted graph problem was developed by dijkstra. Dijkstra's algorithm is a generalization of the BFS algorithm.

We use the distance table. The shortest distance of the source to itself is zero. The distance table for all other vertices is set to α to indicate that those vertices are not already processed.

The graph is our example. The fig 1 represents the initial configuration.



V	Known	dv	Pv
V_1	F	0	0
V_2	F	∞	0
V_3	F	∞	0
V_4	F	∞	0
V_5	F	∞	0
V_6	F	∞	0
V_7	F	∞	0

Initial configuration.

Assuming that the start node S is V_1. The first Vertex selected is V_1, with path length 0. This vertex is marked known. Now that V_1 is known, some entries are need to be adjusted. The vertices adjacent to V_1 are V_2 and V_4. Both these Vertices get their entries adjusted. in fig 2, Next V_4 is selected and marked known. Vertices V_3, V_5, V_6 and V_7 are adjacent, and it turns out that all require adjusting, shown in fig 3

Next V_2 is selected, V_4 is adjacent but already known. so no work is performed on it. V_5 is adjacent but not adjusted, because the cost of going through V_2 is $2+10=12$ and a path of length 3 is already known. figure 4 shows the table after these vertices are selected.

Downloaded from EnggTree.com

EnggTree.com

The next vertex selected is V5 at cost 3. V7 is the only adjacent vertex, but it is not adjusted, because 3+6>5. Then V3 is selected, and the distance for V6 is adjusted down to 3+5=8. The resulting table shows in figure 5.

Next V7 is selected V6 gets updated down to 5+1=6. The resulting table shows in figure 6.

Finally V6 is selected. The final table is shown in figure 7.

The running time depends on how the vertices are manipulated, which we have yet to consider. The total running time is $O(|E|+|V|^2) = O(|V|^2)$ since it runs in time linear in the number of edges.

V	known	dv	Pv
V1	T	0	0
V2	F	2	V1
V3	F	∞	0
V4	F	1	V1
V5	F	∞	0
V6	F	∞	0
V7	F	∞	0

Fig 1 - V1 is declared known

V	known	dv	Pv
V1	T	0	0
V2	F	2	V1
V3	F	3	V4
V4	T	1	V1
V5	F	3	V4
V6	F	9	V4
V7	F	5	V4

Fig 2 V4 is declared known

V	known	dv	Pv
V1	T	0	0
V2	T	2	V1
V3	F	3	V4
V4	T	1	V1
V5	F	3	V4
V6	F	9	V4
V7	F	5	V4

Fig 3 V2 is declared known

V	known	dv	Pv
V1	T	0	0
V2	T	2	V1
V3	T	3	V4
V4	T	1	V1
V5	T	3	V4
V6	F	8	V3
V7	F	5	V4

Fig 4 V5 & V5 are declared known

V	known	dv	Pv
V1	T	0	0
V2	T	2	V1
V3	T	3	V4
V4	T	1	V1
V5	T	3	V4
V6	F	6	V7
V7	T	5	V4

Fig 5 V7 is declared known

V	known	dv	Pv
V1	T	0	0
V2	T	2	V1
V3	T	3	V4
V4	T	1	V1
V5	T	3	V4
V6	T	6	V7
V7	T	5	V4

Fig 6 V6 is declared known.

Downloaded from EnggTree.com

EnggTree.com

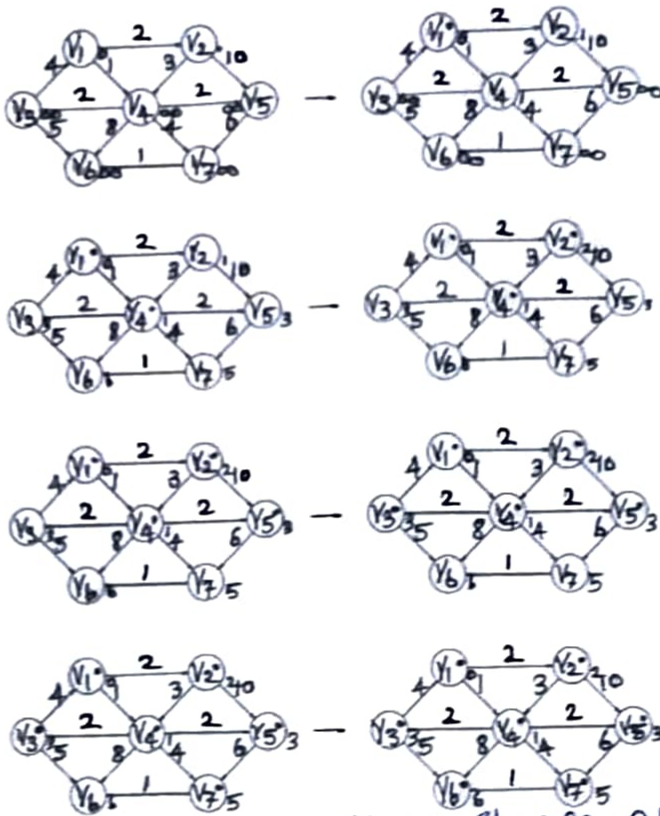


Fig 7 - Stages of Dijkstra's Algorithm.

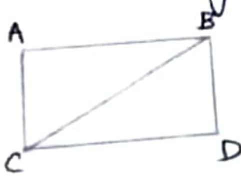
```
#Implementing Dijkstra's Algorithm
def dijkstra(graph, initial):
  visited = {initial : 0}
  path = defaultdict(list)
  nodes = set(graph.nodes)
  while nodes:
    minNode = None
    for node in nodes:
      if node in visited:
        if minNode is None:
          minNode = node
        elif visited[node] < visited[minNode]:
          minNode = node
    if minNode is None:
      break
    nodes.remove(minNode)
    currentWeight = visited[minNode]
    for edge in graph.edges[minNode]:
      weight = currentWeight + graph.distances[(minNode, edge)]
      if edge not in visited or weight < visited[edge]:
        visited[edge] = weight
        path[edge].append(minNode)
  return visited, path
```

Downloaded from EnggTree.com

EnggTree.com

Minimal spanning Tree.

The spanning tree of a graph is a subgraph that contains all the vertices and is also a tree. A graph may have many spanning trees.



For this simple graph we can have multiple spanning trees.

To find the minimum spanning tree in an undirected graph, we assume that the given graphs are weighted graphs.

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connect all the vertices of G with minimum total cost. A minimum spanning tree exisists only if the graph is connected. Two algorithms are used this problem

* Prims Algorithm
* Kruskal's algorithm.

Prims Algorithm.

Prim's Algorithm is almost the same as dijkstra's algorithm. It takes a graph as input and finds the subset of the edges of that graph which

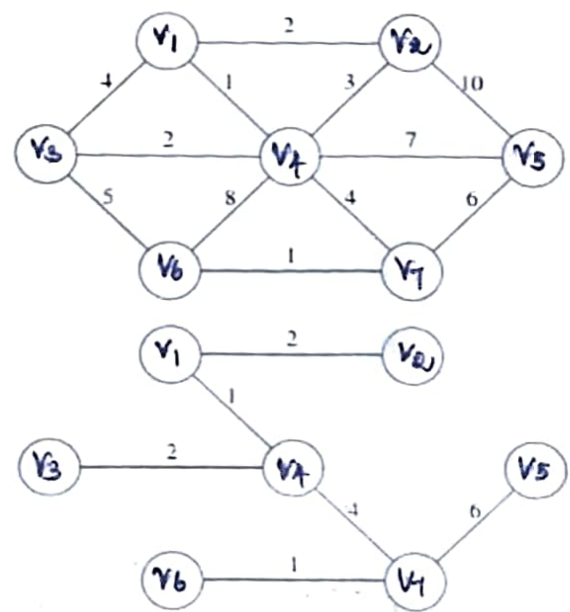
* form a tree that includes every vertex.
* has the minimum sum of weights among all the trees that can be formed from the graph.

Downloaded from EnggTree.com

EnggTree.com

One way to compute a minimum spanning tree is to grow the tree in successive stages. In each stage one node is picked at the root, and we add an edge, and thus an associated vertex, to the tree.

At any point in the algorithm, we can see that we have a set of vertices that have already been included in the tree, the rest of the vertices have not. The algorithm then finds at each stage, a new vertex to add to the tree by choosing the edge(u,v) such that the cost of (u,v) is the smallest among all edges where u is in the tree and v is not.



A graph G and its minimum spanning tree

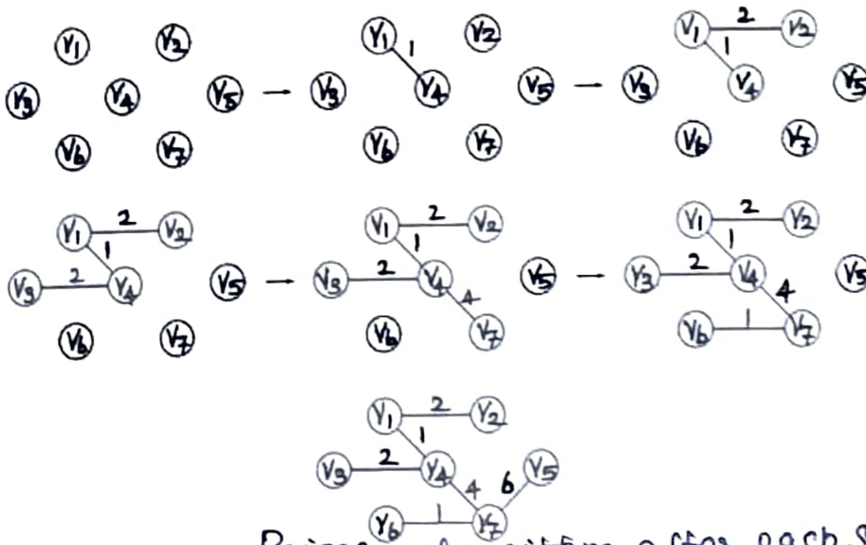
The initial configuration node V_1 is selected and V_2, V_3 and V_4 are updated. The resulting shown in the fig 1. The next vertex selected is V_4. Every vertex adjacent to V_4 except V_1 is examined, because V_1 is known.

Downloaded from EnggTree.com

EnggTree.com

V_2 is unchanged, because it has $d_v = 2$ and the edge cost from V_2 to V_3 is 3, all the rest are updated fig2 shows the resulting table. The next chosen vertex is V_2. This does not affect any distance. Then V_3 is chosen, which affects the distance in V_6, shown in figure 3. Figure 4 results from the selection of V_7, which forces V_6 and V_5 to be adjusted. V_6 and then V_5 are selected, completing the algorithm.

The final table is shown in fig 5. The edges in the spanning tree can be read from the table (V_2, V_1), (V_3, V_4), (V_4, V_1), (V_5, V_7), (V_6, V_7), (V_7, V_4) The total cost is 16.



Prims algorithm after each stage.

V	Known	d_v	P_v
V_1	F	0	0
V_2	F	∞	0
V_3	F	∞	0
V_4	F	∞	0
V_5	F	∞	0
V_6	F	∞	0
V_7	F	∞	0

Initial Table.

V	Known	d_v	P_v
V_1	T	0	0
V_2	F	2	V_1
V_3	F	4	V_1
V_4	F	1	V_1
V_5	F	∞	0
V_6	F	∞	0
V_7	F	∞	0

Table after V_1 is declared known

V	Known	d_v	P_v
V_1	T	0	0
V_2	F	2	V_1
V_3	F	2	V_4
V_4	T	1	V_1
V_5	F	7	V_4
V_6	F	8	V_4
V_7	F	4	V_4

Table after V_4 is declared known.

Downloaded from EnggTree.com

EnggTree.com

V	Known	d_v	P_v
V1	T	0	0
V2	T	2	V1
V3	T	2	V4
V4	T	1	V1
V5	F	7	V4
V6	F	5	V3
V7	F	4	V4

Table after V2 & V3 are declared known

V	Known	d_v	P
V1	T	0	0
V2	T	2	V1
V3	T	2	V4
V4	T	1	V1
V5	F	6	V7
V6	F	1	V7
V7	T	4	V4

Table after V7 is declared known

V	Known	d_v	P_v
V1	T	0	0
V2	T	2	V1
V3	T	2	V4
V4	T	1	V1
V5	T	6	V7
V6	T	1	V7
V7	T	4	V4

Table after V6 & V5 are selected (Prims algorithm Terminates).

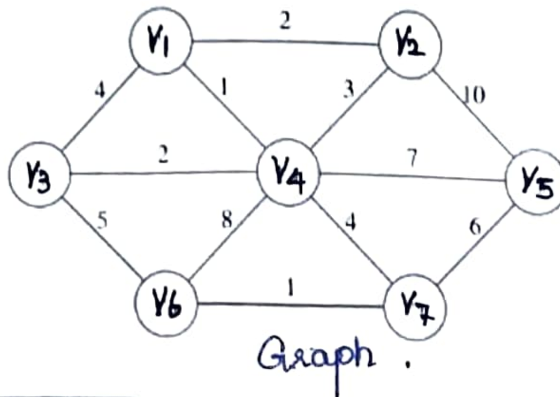
```
def primsAlgo(self):
    visited = [0]*self.vertexNum
    edgeNum=0
    visited[0]=True
    while edgeNum<self.vertexNum-1:
        min = sys.maxsize
        for i in range(self.vertexNum):
            if visited[i]:
                for j in range(self.vertexNum):
                    if ((not visited[j]) and self.edges[i][j]):
                        if min > self.edges[i][j]:
                            min = self.edges[i][j]
                            s = i
                            d = j
        self.MST.append([self.nodes[s], self.nodes[d], self.edges[s][d]])
        visited[d] = True
        edgeNum += 1
```

Downloaded from EnggTree.com

EnggTree.com

Kruskal's Algorithm.

The algorithm starts with v different trees. While constructing the minimum spanning tree, every time Kruskal's algorithm selects an edge that has minimum weight and then adds that edge, if it doesnot create a cycle. so initially, there are |V| single node trees in the forest. Adding an edge merges two trees into one. when the algorithm is completed, there will be only one tree, and that is the minimum spanningtree



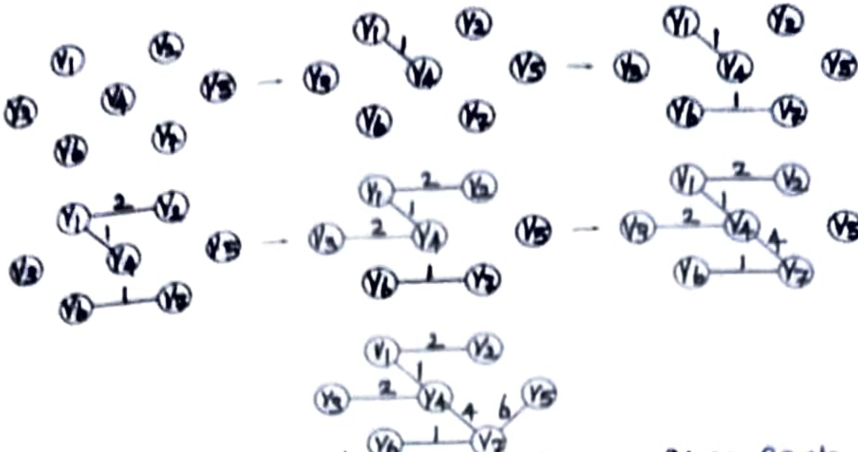
Graph.

Edge	Weight	Action
(V1,V4)	1	Accepted
(V6,V7)	1	Accepted
(V1,V2)	2	Accepted
(V3,V4)	2	Accepted
(V2,V4)	3	Rejected
(V1,V3)	4	Rejected
(V4,V7)	4	Accepted
(V3,V6)	5	Rejected
(V5,V7)	0	Accepted

Action of Kruskal's algorithm on G.

Downloaded from EnggTree.com

EnggTree.com



Kruskal's algorithm after each stage.

```
class Graph:
  def __init__(self, vertex):
    self.V = vertex
    self.graph = []
  def add_edge(self, u, v, w):
    self.graph.append([u, v, w])
  def search(self, parent, i):
    if parent[i] == i:
      return i
    return self.search(parent, parent[i])
  def apply_union(self, parent, rank, x, y):
    xroot = self.search(parent, x)
    yroot = self.search(parent, y)
    if rank[xroot] < rank[yroot]:
      parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
      parent[yroot] = xroot
    else:
      parent[yroot] = xroot
      rank[xroot] += 1
  def kruskal(self):
    result = []
    i, e = 0, 0
    self.graph = sorted(self.graph, key=lambda item: item[2])
    parent = []
    rank = []
    for node in range(self.V):
      parent.append(node)
      rank.append(0)
    while e < self.V - 1:
      u, v, w = self.graph[i]
      i = i + 1
      x = self.search(parent, u)
      y = self.search(parent, v)
      if x != y:
        e = e + 1
        result.append([u, v, w])
        self.apply_union(parent, rank, x, y)
    for u, v, weight in result:
      print("Edge:",u, v,end =" ") print("-",weight)
```

Downloaded from EnggTree.com