QUESTION BANK

UNIT-I

ABSTRACT DATA TYPES

PART-A

1. What is a data structure?

A data structure is a method for organizing and storing data which would allow efficient data retrieval and usage. A data structure is a way of organizing data that considers not only the items stored, but also their relationships to each other.

2. Why do we need data structures?

● Data structures allow us to achieve an important goal: component reuse.

● Once data structure has been implemented, it can be used again and again in various applications.

3. List some common data structures.

● Stacks

● Queues

● Lists

● Trees

● Graphs

● Tables

4. Define ADT (Abstract Data Type)

An abstract data type (ADT) is a set of operations and mathematical abstractions , which can be viewed as how the set of operations is implemented. Objects like lists, sets and graphs, along with their operation, can be viewed as abstract data types, just as integers, real numbers and Booleans.

5. Mention the features of ADT.

a. Modularity
i. Divide program into small functions
ii. Easy to debug and maintain
iii. Easy to modify
b. Reuse
i. Define some operations only once and reuse them in future
c. Easy to change the implementation

6. What is a Linear data structure?

Linear data structures is a continuous arrangement of data elements in the memory. It can be constructed by using array data type. The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

For example Stack, Queue, Tables, List, and Linked Lists.

7.State the difference between arrays and linked lists

Arrays	Linked Lists
Size of an array is fixed	Size of a list is variable
It is necessary to specify the number of elements during declaration.	It is not necessary to specify the number of elements during declaration
Insertions and deletions are somewhat difficult	Insertions and deletions are carried out easily
It occupies less memory than a linked list for the same number of elements	It occupies more memory

8. What are the advantages of modularity?

• It is much easier to debug small routines than large routines.

It is easier for several people to work on a modular program simultaneously

• A well-written modular program places certain dependencies in only one routine, making changes easier.

9. Define class.

A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties

The example for class of car can be :

class Car :

10. Define Object.

The Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

For example "Dog" is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

The example for object of car class can be:

obj = maruti()

Here, obj is an object of class car.

11. What is constructor?

The constructor is a method that is called when an object is created. This method is defined in the class and can be used to initialize basic variables. If you create four objects, the class constructor is called four times. Every class has a constructor, but its not required to explicitly define it.

12. Write short notes on python__init__.

The function init(self) builds your object. Its not just variables you can set here, you can call class methods too. Everything you need to initialize the object(s). Lets say you have a class Plane, which upon creation should start flying. There are many steps involved in taking off: accelerating, changing flaps, closing the wheels and so on.

```
classPlane:
def__init__(self):
self.wings = 2
```

13. Define Inheritance.What are its types?

The inheritance is the process of acquiring the properties of one class(parent class) to another class(child class). In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass. The Parent class is the class which provides features to another class. The parent class is also known as Base class or Superclass. The Child class is the class which receives features from another class. The child class is also known as the Derived Class or Subclass.

There are five types of inheritances, and they are as follows.

- Simple Inheritance (or) Single Inheritance
- Multiple Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

14. What is namespace?

A namespace is **a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it**. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

15. What is divide an conquer?

A divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems solving the sub-problems, and combining them to get the desired output.

To use the divide and conquer algorithm, recursion is used

16. Define recursion.

A recursion is a process of function calling itself.

example:

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

17. Differentiate is shallow and deep copying?

Shallow Copy	Deep Copy
Shallow Copy stores the references of objects to the original memory address.	Deep copy stores copies of the object's value.
Shallow Copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect changes made to the new/copied object in the original object.
Shallow Copy stores the copy of the original object and points the references to the objects.	Deep copy stores the copy of the original object and recursively copies the objects as well.
Shallow copy is faster.	Deep copy is comparatively slower.

18.What are asymptotic notations.

Asymptotic notations are the mathematical notations used to describe the runningtime of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by thealgorithm is linear i.e the best case

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

19. What is OOP?

Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts:

- Class
- Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

20. Define Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

PART-B

1.Explain about Inheritance in detail.

Types of inheritances:

The inheritance is a very useful and powerful concept of object-oriented programming. Using the inheritance concept, we can use the existing features of one class in another class.

The inheritance is the process of acquiring the properties of one class to another class.

In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.

The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.

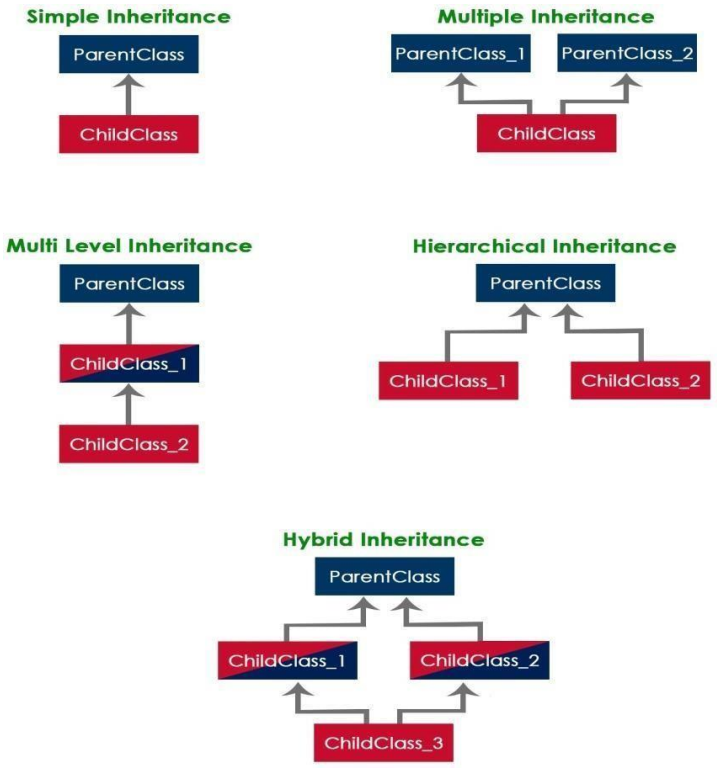
The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

There are five types of inheritances, and they are as follows.

- **Simple Inheritance (or) Single Inheritance**
- **Multiple Inheritance**
- **Multi-Level Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

The following picture illustrates how various inheritances are implemented.



Creating a Child Class

In Python, we use the following general structure to create a child class from a parent class.

Syntax

```
classChildClassName(ParentClassName):
    ChildClass implementation
.
.
```

Let's look at individual inheritance type with an example.

Simple Inheritance (or) Single Inheritance

In this type of inheritance, one child class derives from one parent class. Look at the following example code.

Example

```
classParentClass:

deffeature_1(self):
print('feature_1 from ParentClass is running...')

deffeature_2(self):
print('feature_2 from ParentClass is running...')



classChildClass(ParentClass):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
```

```
obj.feature_3()
```

2. Explain in detail about classes and object with example programs.

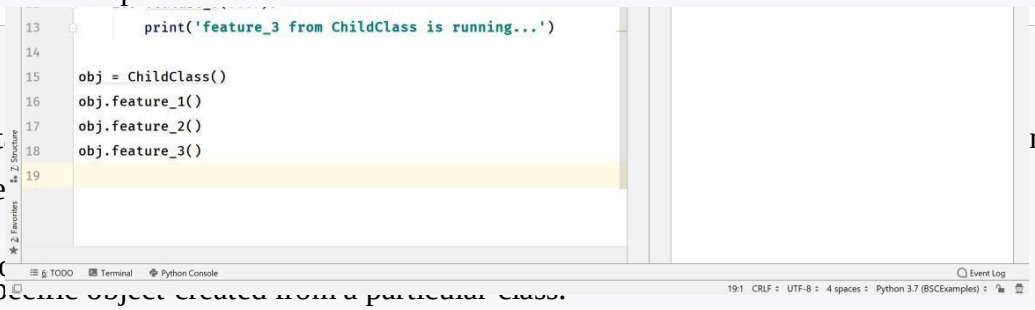
OOPs in Python is a programming approach that focuses on using objects and classes as same as other general programming languages. The objects can be any real-world entities. Python allows developers to develop applications using the OOPs approach with the major focus on code reusability.

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

3. The example for class of parrot can be :



Object

An object (inst r the object is defined. The

The example fo
instance is a specific object created from a particular class.

```
obj = Parrot()
```

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example:
class Parrot:

# class attribute
species = "bird"

instance attribute
def_____init_(self, name, age):
self.name = name self.age = age

instantiate the Parrot classblu
= Parrot("Blu", 10) woo =
Parrot("Woo",

# access the class attributes
print("Blu is a {}".format(blu. class .species))
print("Woo is also a {}".format(woo. class .species))

# access the instance attributes
print("{} is {} years old".format(blu.name, blu.age))
print("{} is {} years old".format(woo.name, woo.age))

3.What is Recursion. Elaborate about Binary Search and File systems.

A recursion is a process of function calling itself.

example:

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element. We can find the element's index position very fast using the binary search algorithm.

There are many searching algorithms but the binary search is most popular among them.

The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.

Let's understand the concept of binary search.

Concept of Binary Search

The divide and conquer approach technique is followed by the recursive method. In this method, a functionis called itself again and again until it found an element in the list.

A set of statements is repeated multiple times to find an element's index position in the iterative method.The **while** loop is used for accomplish this task.

Binary search is more effective than the linear search because we don't need to search each list index. Thelist must be sorted to achieve the binary search algorithm.

Let's have a step by step implementation of binary search.

We have a sorted list of elements, and we are looking for the index position of 45.

[12, 24, 32, 39, 45, 50, 54]

So, we are setting two pointers in our list. One pointer is used to denote the smaller value called **low** and thesecond pointer is used to denote the highest value called **high**.

Next, we calculate the value of the **middle** element in the array.

1. mid = (low+high)/2
2. Here, the low is 0 and

the high is 7.3. mid =

(0+7)/2

4. mid = 3 (Integer)

4.Explian the concept of shallow and deep copying.

Shallow Copy	Deep Copy
Shallow Copy stores the references of objects to the original memory address.	Deep copy stores copies of the object's value.
Shallow Copy reflects changes made to the new/copied object in the original object.	Deep copy doesn't reflect changes made to the new/copied object in the original object.
Shallow Copy stores the copy of the original object and points the references to the objects.	Deep copy stores the copy of the original object and recursively copies the objects as well.
Shallow copy is faster.	Deep copy is comparatively slower.

5.Brieflyexplain about the various asymptotic notations.

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing

of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$.

This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly sma

The time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

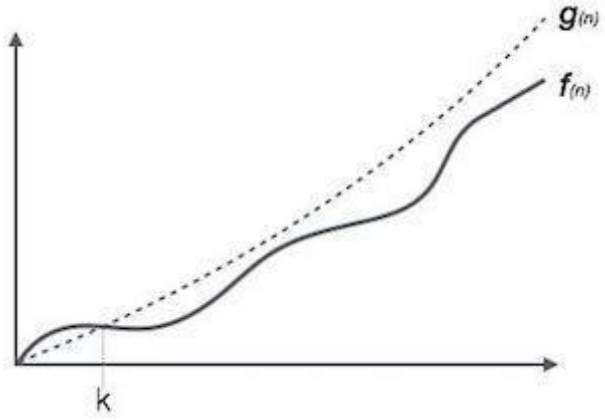
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

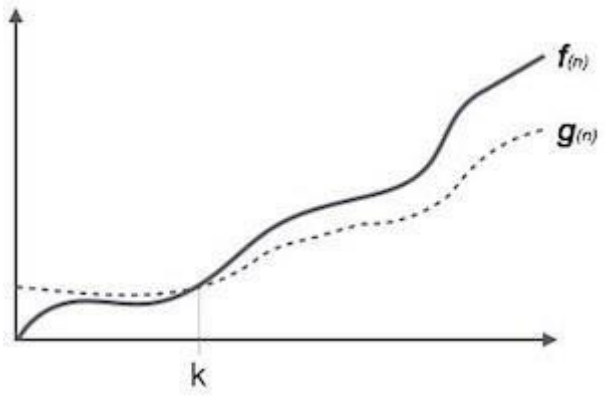


For example, for a function ***f*(n)**

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0. \}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



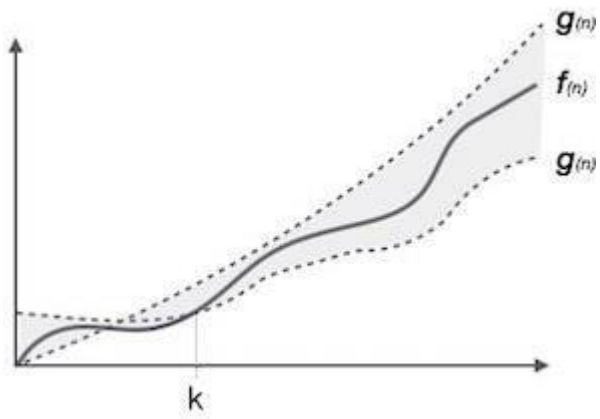
For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper boundof

an algorithm's running time. It is represented as follows −



$\theta(f(n))$ = { $g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$. }

UNIT-II LINEAR STRUCTURES
PART-A

1. What are the types of queues?

• **Linear Queues** – The queue has two ends, the front end and the rear end. The rear end is where we insert elements and front end is where we delete elements. We can traverse in a linear queue in only one direction ie) from front to rear.

• **Circular Queues** – Another form of linear queue in which the last position is connected to the first position of the list. The circular queue is similar to linear queue has two ends, the front end and the rear end. The rear end is where we insert elements and front end is where we delete elements. We can traverse in a circular queue in only one direction ie) from front to rear.

• **Double-Ended-Queue** – Another form of queue in which insertions and deletions are made at both the front and rear ends of the queue.

2. What are the ways of implementing linked list?

The list can be implemented in the following ways:

i. Array implementation

ii. Linked-list implementation

iii. Cursor implementation

3. What are the types of linked lists?

There are three types

i. Singly linked list

ii. Doubly linked list

iii. Circularly linked list

4. When singly linked list can be represented as circular linked list?

In a singly linked list, all the nodes are connected with forward links to the next nodes in the list. The last node has a next field, NULL. In order to implement the circularly linked lists from singly linked lists, the last node's next field is connected to the first node.

5. List down the applications of List.

a. Representation of polynomial ADT

b. Used in radix and bubble sorting

c. In a FAT file system, the metadata of a large file is organized as a linked list of FAT entries.

d. Simple memory allocators use a free list of unused memory regions, basically a linked list with the list pointer inside the free memory itself.

6. What are the advantages of linked list?

a. Save memory space and easy to maintain

b. It is possible to retrieve the element at a particular index

c. It is possible to traverse the list in the order of increasing index.

d. It is possible to change the element at a particular index to a different value,without affecting any other elements.

7. What are the operations performed in list?

The following operations can be performed on a list

i. Insertion

a. Insert at beginning
b. Insert at end
c. Insert after specific node
d. Insert before specific node

ii. Deletion

a. Delete at beginning
b. Delete at end
c. Delete after specific node
d. Delete before specific node

iii. Merging

iv. Traversal

8. What is the need for the header?

Header of the linked list is the first element in the list and it stores the number of elements in the list. It points to the first data element of the list.

9. List the applications of stacks

- Towers of Hanoi
- Reversing a string
- Balanced parenthesis
- Recursion using stack
- Evaluation of arithmetic expressions.

10. List the applications of queues

- Jobs submitted to printer
- Real life line
- Calls to large companies
- Access to limited resources in Universities
- Accessing files from file server

11. Define a stack

Stack is an ordered collection of elements in which insertions and deletions are restricted to one end. The end from which elements are added and/or removed is referred to as top of the stack. Stacks are also referred as piles, push-down lists and last-in-first- out (LIFO) lists.PUSH and POP are the basic operations of a stack.

12. List out the basic operations that can be performed on a stack

The basic operations that can be performed on a stack are

- Push operation
- Pop operation
- Peek operation

- Empty check
- Fully occupied check

13. State the different ways of representing expressions

The different ways of representing expressions are

- Infix Notation
- Prefix Notation
- Postfix Notation

14. Mention the advantages of representing stacks using linked lists than arrays

- It is not necessary to specify the number of elements to be stored in a stack during its declaration, since memory is allocated dynamically at run time when an element is added to the stack
- Insertions and deletions can be handled easily and efficiently
- Linked list representation of stacks can grow and shrink in size without wasting memory space, depending upon the insertion and deletion that occurs in the list
- Multiple stacks can be represented efficiently using a chain for each stack

15. Define a queue

Queue is an ordered collection of elements in which insertions are restricted to one end called the rear end and deletions are restricted to other end called the front end. Queues are also referred as First-In-First-Out (FIFO) Lists.Enqueue and Dequeue are the operations of a queue.

16. State the difference between queues and linked lists

The difference between queues and linked lists is that insertions and deletions may occur anywhere in the linked list, but in queues insertions can be made only in the rear end and deletions can be made only in the front end.

17. Define a Deque

Deque (Double-Ended Queue) is another form of a queue in which insertions and deletions are made at both the front and rear ends of the queue. There are two variations of a deque, namely, input restricted deque and output restricted deque. The input restricted deque allows insertion at one end (it can be either front or rear) only. The output restricted deque allows deletion at one end (it can be either front or rear).

18. Define a priority queue

Priority queue is a collection of elements, each containing a key referred as the priority for that element. Elements can be inserted in any order (i.e., of alternating priority), but are arranged in order of their priority value in the queue. The elements are deleted from the queue in the order of their priority (i.e., the elements with the highest priority is deleted first). The elements with the same priority are given equal importance and processed accordingly.

19. List out the advantages and disadvantages of using a linked list

Advantages of linked list

- It is not necessary to specify the number of elements in a linked list during its declaration .
- Linked list can grow and shrink in size depending upon the insertion and deletion that occurs in the list
- Insertions and deletions at any place in a list can be handled easily and efficiently
- A linked list does not waste any memory space .

disadvantages of using a linked list

• Searching a particular element in a list is difficult and time consuming.

- A linked list will use more storage space than an array to store the same numberof elements.

20. List the basic operations carried out in a linked list

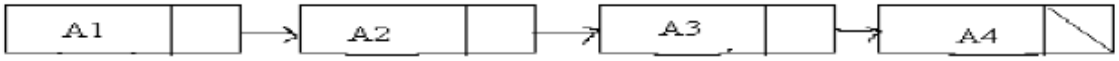
The basic operations carried out in a linked list include:

• Creation of a list
• Insertion of a node
• Deletion of a node
• Modification of a node
• Traversal of the list

21. Define Linked Lists . State the different types of linked lists

Linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this theNext Pointer. The last cell'sNext pointer points to NULL.

The different types of linked list include singly linked list, doubly linked list andcircular linked list.



PART-B

1.Explain in detail about Stack ADT and its operations with Python program.

Stack: It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.

The two basic operations associated with stacks are:

1. Push
2. Pop

While performing push and pop operations the following test must be conducted on thestack.

a) Stack is empty or not b) stack is full or not

1. Push: Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

2. Pop: Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of thestack.

Representation of Stack (or) Implementation of stack:

The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a ***stack overflow*** condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a ***stack underflow*** condition.

1. **push():** When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

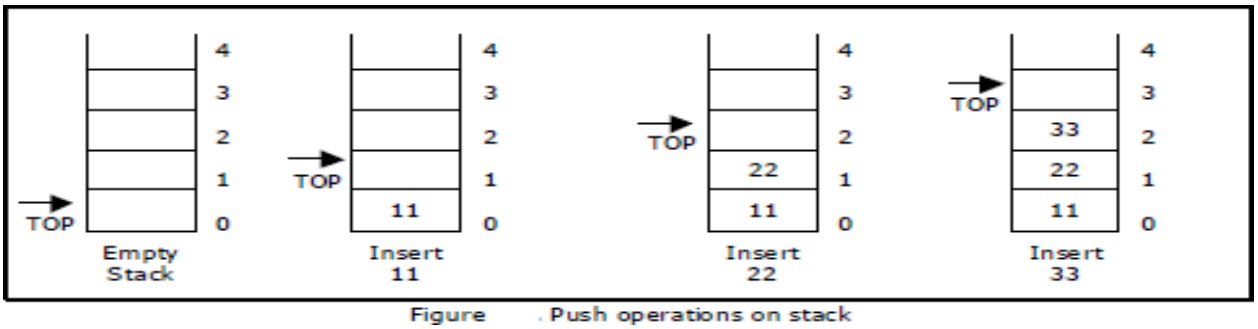


Figure . Push operations on stack

Initially **top=-1**, we can insert an element in to the stack, increment the top value i.e **top=top+1**. We can insert an element in to the stack first check the condition is stack is full or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

1. **Pop():** When an element is taken off from the stack, the operation is performed by pop(). Below figure shows a stack initially with three elements and shows the deletion of elements using pop().

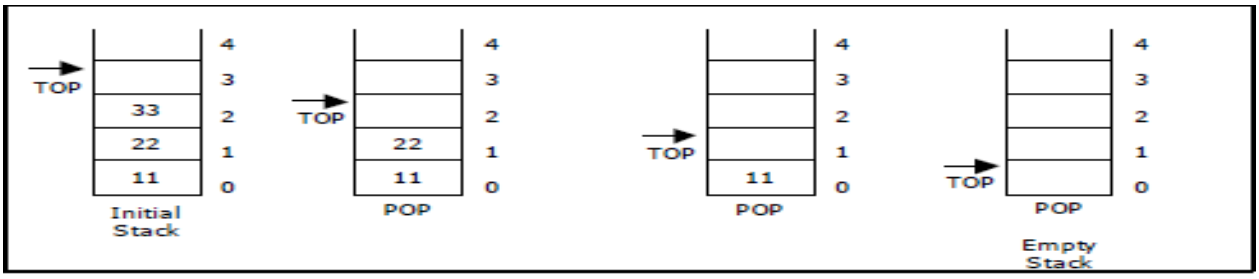


Figure Pop operations on stack

1. **display():** This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.

2. Explain in detail about Queue ADT and its operations with Python program.

Queue: It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear** The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue.

Operations on QUEUE:

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

- **Enqueue or insertion:** which inserts an element at the end of the queue.
- **Dequeue or deletion:** which deletes an element at the start of the queue.

Queue operations work as follows:

1. Two pointers called FRONT and REAR are used to keep track of the first and lastelements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to 0.
3. On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.
4. On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of FRONT to 1.
8. When dequeing the last element, we reset the values of FRONT and REAR to 0.

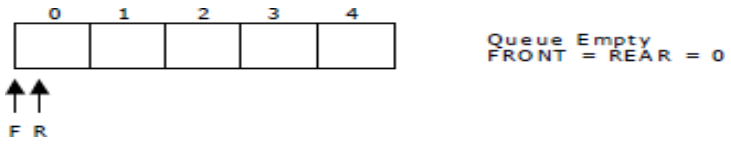
Representation of Queue (or) Implementation of Queue:

The queue can be represented in two ways:

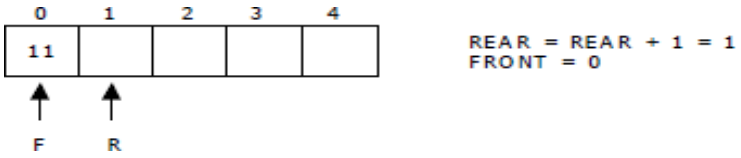
1. Queue using Array
2. Queue using Linked List

1. Queue using Array:

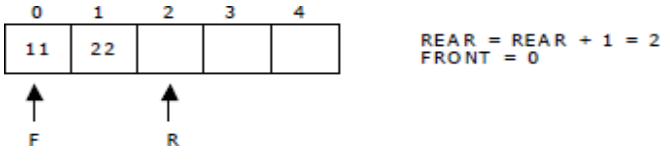
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



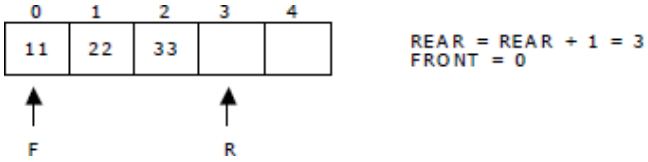
Now, insert 11 to the queue. Then queue status will be:



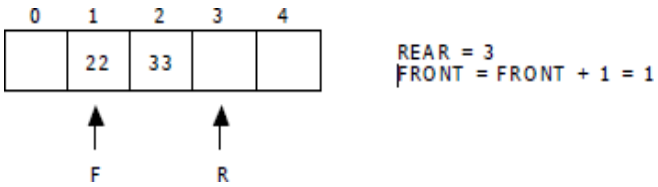
Next, insert 22 to the queue. Then the queue status is:



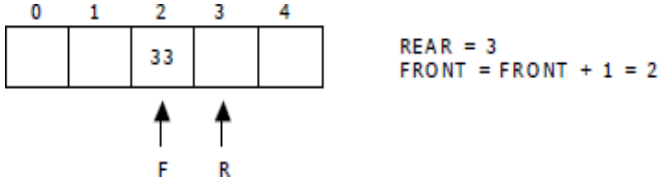
Again insert another element 33 to the queue. The status of the queue is:



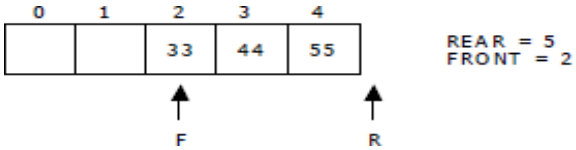
Now, delete an element. The element deleted is the element at the front of the queue.Sothe status of the queue is:



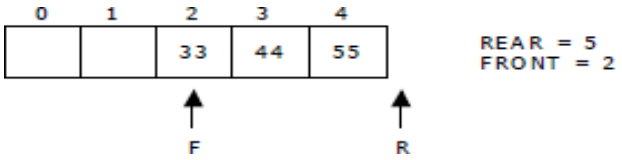
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

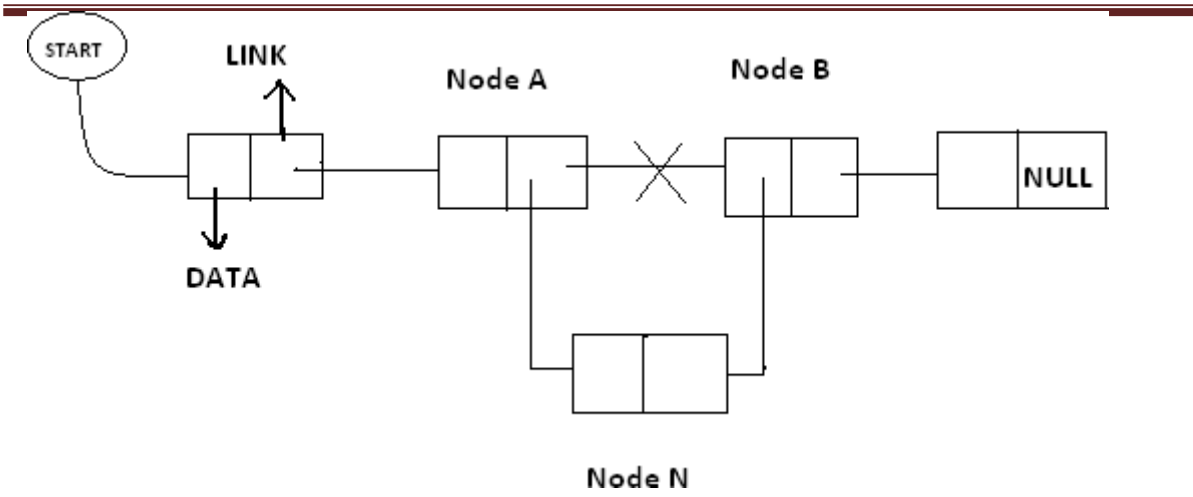


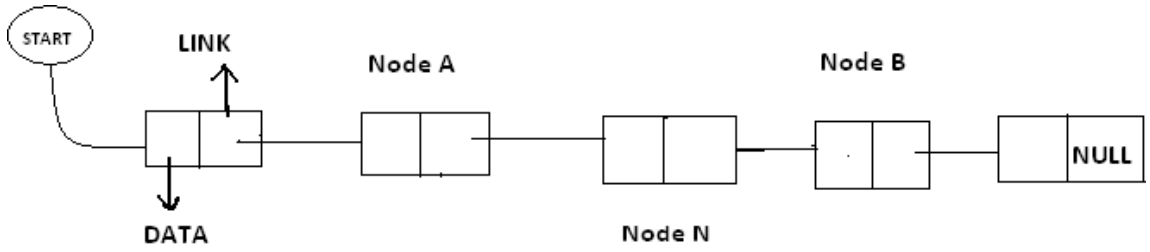
Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:

3. .Discuss in detail about Singly Linked list with program.

Insertion and Deletion of a Single Linked List:

Insertion Let the list be a Linked list with succesive nodes A and B as shown in below figure.suppose a node N id to be inserted into the list between the node A and B.



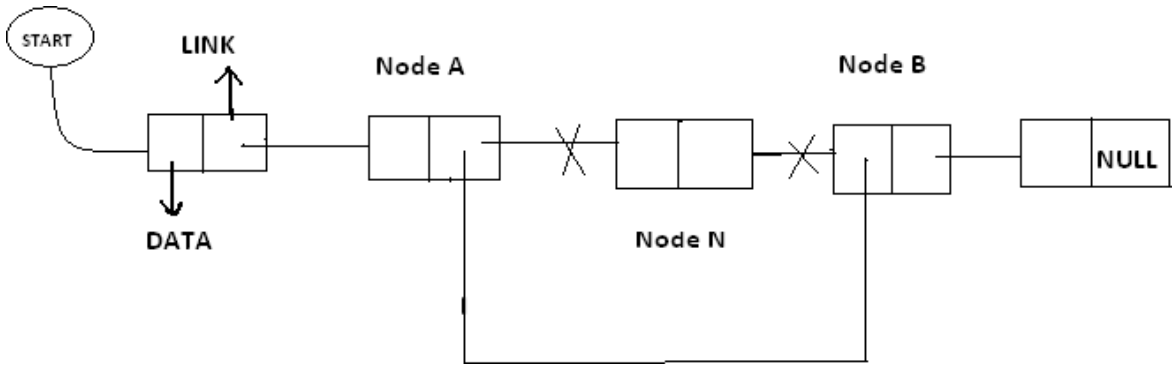


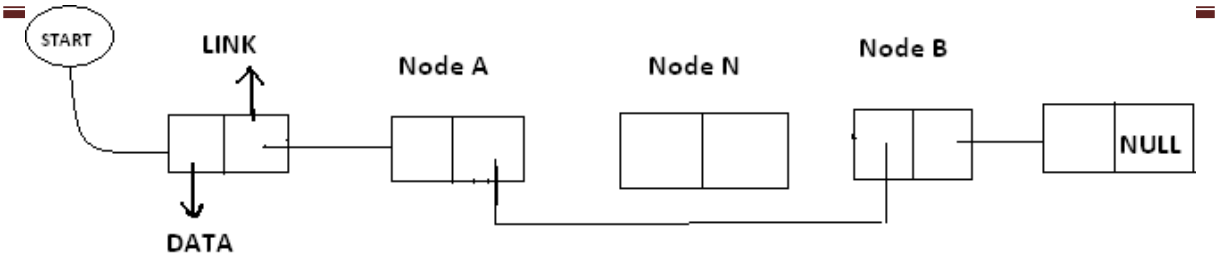
After Insertion

In the New list the Node A points to the new Node N and the new node N points to the node B to which Node A previously pointed.

Deletion:

Let list be a Linked list with node N between Nodes A and B is as shown in the following figure.





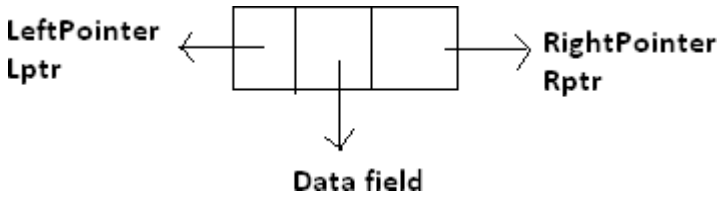
After Deletion Node N in Between Node A abd Node B

In the new list the node N is to be deleted from the Linked List. The deletion occurs as the link field in the Node A is made to point node B this excluding node N from its path.

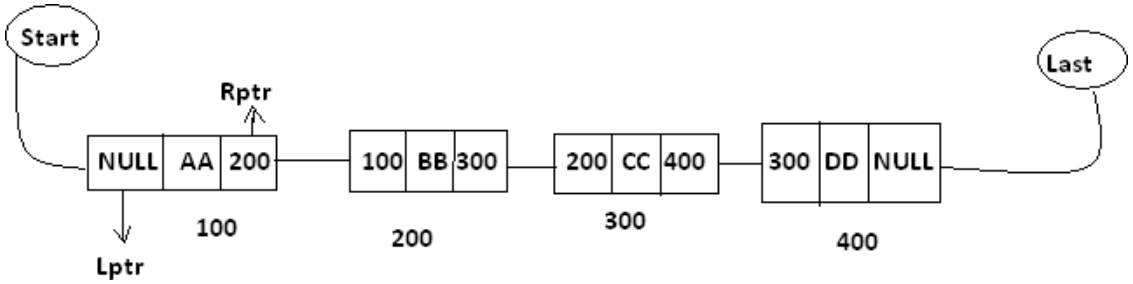
4.Expalin how a Doubly Linked list is implemented with appropriate example programs.

DOUBLE LINKED LIST (Or) TWO WAY LINKED LIST

In certain applications it is very desirable that list be traversed in either forward direction or Back word direction. The property of Double Linked List implies that each node must contain two link fields instead of one. The links are used to denote the preceding and succeeding of the node. The link denoting the preceding of a node is called Left Link. The link denoting succeeding of a node is called Right Link. The list contain this type of node is called a "**Double Linked List**" or "**Two Way List**". The Node structure in the Double Linked List is as follows:

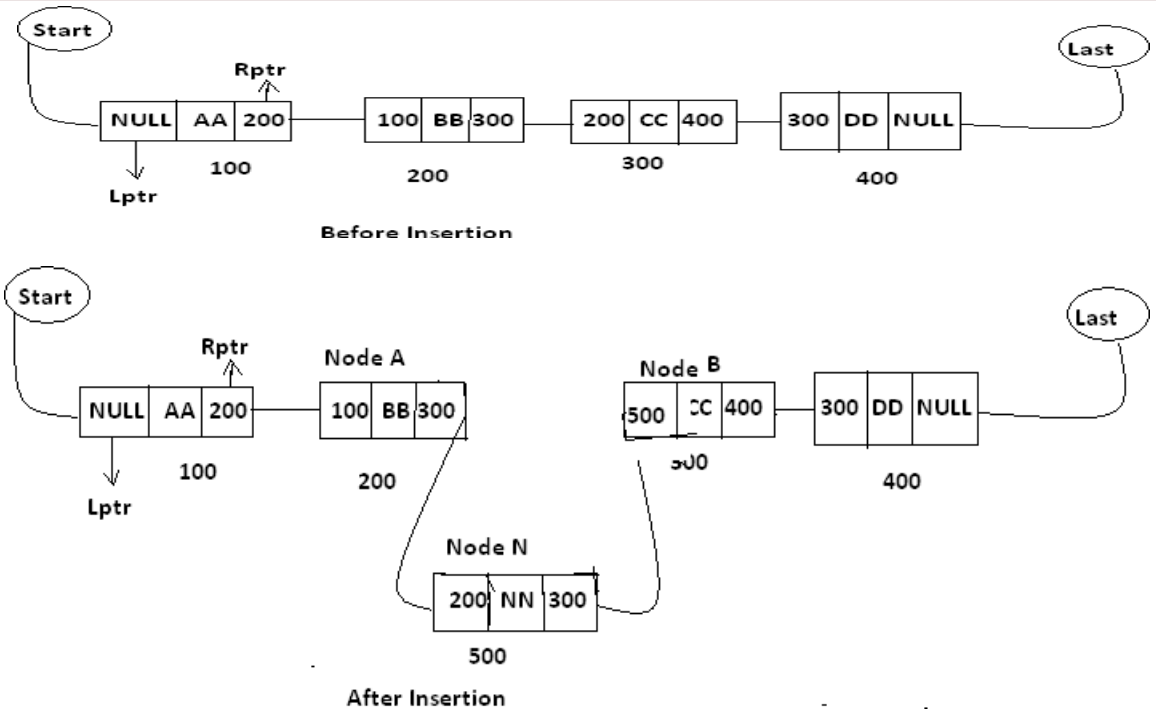


Lptr contains the address of the before node. Rptr contains the address of next node. Data Contains the Linked List is as follows.



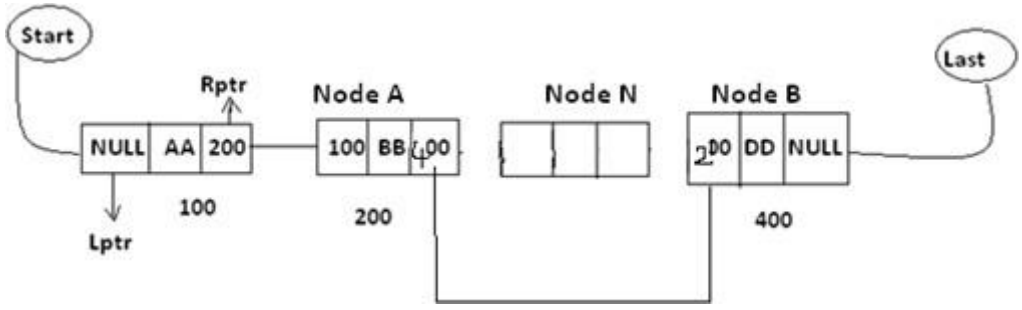
In the above diagram Last and Start are pointer variables which contains the address of last node and starting node respectively.

Insertion in to the Double Linked List: Let list be a double linked list with successive modes A and B as shown in the following diagram. Suppose a node N is to be inserted into the list between the node s A and B this is shown in the following diagram.



As in the new list the right pointer of node A points to the new node N ,the Lptr of the node 'N' points to the node A and Rptr of node 'N' points to the node 'B' and Lpts of node B points the new node 'N'

Deletion Of Double Linked List :- Let list be a linked list contains node N between the nodesA and B as shown in the following diagram.

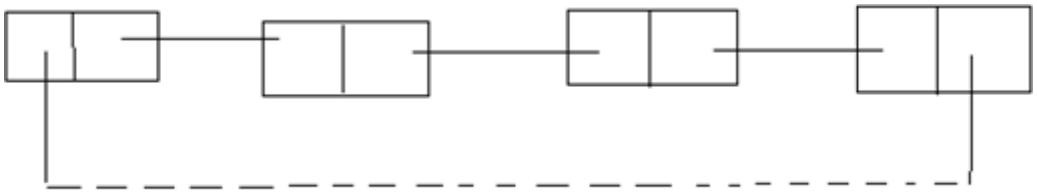


5.Elaborate with example about circular linked list.

Circular Linked List:- Circular Linked List is a special type of linked list in which all the nodes are linked in continuous circle. Circular list can be singly or doubly linked list. Note that,there are no Nulls in Circular Linked Lists. In these types of lists, elements can be added to the back of the list and removed from the front in constant time.

Both types of circularly-linked lists benefit from the ability to traverse the full list beginning at any given node. This avoids the necessity of storing first Node and last node, but we need a special representation for the empty list, such as a last node variable which points to some node in the list or is null if it's empty. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case. Circular linked lists are most useful for describing naturally circular structures, and have the advantage of being able to traverse the list starting at any point. They also allow quick access to the first and last records through a single pointer (the address of the last element)

Circular single linked list:



Circular Single Linked List

Circular linked list are one they of liner linked list. In which the link fields of last node of the list contains the address of the first node of the list instead of contains a null pointer.

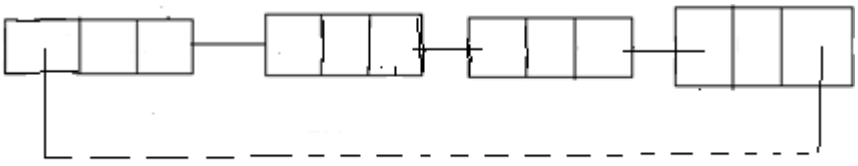
Advantages:- Circular list are frequency used instead of ordinary linked list because in circular list all nodes contain a valid address. The important feature of circular list is as follows.

(1) In a circular list every node is accessible from a given node.

(2) Certain operations like concatenation and splitting becomes more efficient in circular list.

Disadvantages: Without some conditions in processing it is possible to get into an infinite Loop.

Circular Double Linked List :- These are one type of double linked list. In which the rpt field of the last node of the list contain the address of the first node ad the left points of the first node contains the address of the last node of the list instead of containing null pointer.



Circular Double Linked List

UNIT-III
SORTING AND SEARCHING
PART-A

1.Define sorting

Sorting arranges the numerical and alphabetical data present in a list in a specific order or sequence. There are a number of sorting techniques available. The algorithms can be chosen based on the following factors

– Size of the data structure

– Algorithm efficiency

– Programmer's knowledge of the technique.

2. What do you mean by internal and external sorting?

An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive)

3. Define bubble sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

4. How the insertion sort is done with the array?

It sorts a list of elements by inserting each successive element in the previously sorted sublist.

Consider an array to be sorted A[1],A[2],….A[n]

a. Pass 1 : A[2] is compared with A[1] and placed them in sorted order.

b. Pass 2 : A[3] is compared with both A[1] and A[2] and inserted at an appropriate place. This makes A[1], A[2],A[3] as a sorted sub array.

c. Pass n-1 : A[n] is compared with each element in the sub array

A[1],A[2],……A[n-1] and inserted at an appropriate position.

5. What are the steps for selection sort?

 The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.

 Initially, the sorted sublist is empty and the unsorted sublist is the entire input list.

 The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

6. What are the steps in quick sort?

The steps are:

a. Pick an element, called a **pivot**, from the list.

b. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.

c. Recursively apply the above steps to the sub-list of elements with smaller values and separately to the sub-list of elements with greater values.

7. What are the advantages and disadvantages of insertion sort

Advantages

a. Simplest sorting technique and easy to implement

b. It performs well in the case of smaller lists.

c. It leverages the presence of any existing sort pattern in the list

Disadvantages

 Efficiency of O(n) is not well suited for large sized lists

 It requires large number of elements to be shifted

8. Define searching

Searching refers to determining whether an element is present in a given list of elements or not. If the element is present, the search is considered as successful, otherwise it is considered as an unsuccessful search. The choice of a searching technique is based on the following factors

a. Order of elements in the list i.e., random or sorted

b. Size of the list

9. Mention the types of searching

The types are

 Linear search

 Binary search

10. What is meant by linear search?

Linear search or **sequential search** is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

11. What is binary search?

For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the middle element of the array.

If the key match, then a matching element has been found and its index, or position, is returned.
Otherwise, if the search key is less than the middle element, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right.

12. Define hashing function

A hashing function is a key-to-transformation, which acts upon a given key to compute the relative position of the key in an array.
A simple hash function
HASH(KEY_Value)= (KEY_Value)mod(Table-size)

13. What is open addressing?

Open addressing is also called closed hashing, which is an alternative to resolve the collisions with linked lists. In this hashing system, if a collision occurs, alternative cells are tired until an empty cell is found.
There are three strategies in open addressing:

- Linear probing
- Quadratic probing
- Double hashing

14. What are the collision resolution methods?

The following are the collision resolution methods

- Separate chaining
- Open addressing
- Multiple hashing

15. Define separate chaining

It is an open hashing technique. A pointer field is added to each record location, when an overflow occurs, this pointer is set to point to overflow blocks making a linked list.
In this method, the table can never overflow, since the linked lists are only extended upon the arrival of new keys.

16. Define Hashing.

Hashing is the transformation of string of characters into a usually shorter fixed length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the short hashed key than to find it using the original value.

17. What do you mean by hash table?

The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value. Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell.

18. What do you mean by separate chaining?

Separate chaining is a collision resolution technique to keep the list of all elements that hash to the same value. This is called separate chaining because each hash table element is a separate chain (linked list). Each linked list contains all the elements whose keys hash to the same index.

19. What do you mean by open addressing?

Open addressing is a collision resolving strategy in which, if collision occurs alternative cells are tried until an empty cell is found. The cells h0(x), h1(x), h2(x),…. are tried in succession, where hi(x)=(Hash(x)+F(i))mod Tablesize with F(0)=0. The function F is the collision resolution strategy.

20. What do you mean by Probing and linear probing?

Probing is the process of getting next available hash table array cell.

Linear probing is an open addressing collision resolution strategy in which F is a linear function of i, F(i)=i. This amounts to trying sequentially in search of an empty cell. If the table is big enough, a free cell can always be found, but the time to do so can get quite large.

21. What do you mean by quadratic probing?

Quadratic probing is an open addressing collision resolution strategy in which F(i)=i2. There is no guarantee of finding an empty cell once the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.

PART-B

1.Explain in detail Bubble sort with example.

- **Bubble Sort** is a sorting algorithm used to sort list items in ascending order by comparing two adjacent values.
- If the first value is higher than second value, the first value takes the second value position, while second value takes the first value position.(i.e. Swapping is done)
- If the first value is lower than the second value, then no swapping is done.
- This process is repeated until all the values in a list have been compared and swapped if necessary.
- Each iteration is usually called a pass.
- The number of passes in a bubble sort is equal to the number of elements in a list minus one.

Let's understand it by an example -

Example -

We are creating a list of element, which stores the integer numbers

list1 = [5, 3, 8, 6, 7, 2]

Here the algorithm sort the elements -

First iteration

[**5, 3**, 8, 6, 7, 2]

It compares the first two elements and here 5>3 then swap with each other. Now we get new list is -

[**3, 5,** 8, 6, 7, 2]

In second comparison, 5 < 8 then swapping happen -

[3, **5, 8,** 6, 7, 2]

In third comparison, 8>6 then swap -

[3, 5, **6, 8,** 7, 2]

In fourth comparison, 8>7 then swap -
[3, 5, 6, **7, 8**, 2]
In fifth comparison, 8>2 then swap-
[3, 5, 6, 7, **2, 8**]
Here the first iteration is complete and we get the largest element at the end. Now we need to the len(list1) - 1

[**3, 5**, 6, 7, 2, 8] - > [**3, 5**, 6, 7, 2, 8] here, 3<5 then no swap taken place

[3, **5, 6,** 7, 2, 8] - > [3, **5, 6,** 7, 2, 8] here, 5<6 then no swap taken place

[3, 5, **6, 7**, 2, 8] - > [3, 5, **6, 7**, 2, 8] here, 6<7 then no swap taken place

[3, 5, 6, **7, 2**, 8] - > [3, 5, 6, **2, 7**, 8] here 7>2 then swap their position. Now

Second Iteration

3, 5, 6, **2, 7**, 8] - > [3, 5, 6, 2, **7, 8**] here 7<8 then no swap taken place.

Third Iteration

[**3, 5**, 6, 2, 7, 8] - > [**3, 5**, 6, 7, 2, 8] here, 3<5 then no swap taken place

[3, **5, 6,** 2, 7, 8] - > [3, **5, 6,** 7, 2, 8] here, 5<6 then no swap taken place[

[3, 5, **6, 2**, 7, 8] - > [3, 5, **2, 6**, 7, 8] here, 6<2 then swap their positions

[3, 5, 2, **6, 7**, 8] - > [3, 5, 2, **6, 7**, 8] here 6<7 then no swap taken place. Now

[3, 5, 2, 6, **7, 8**] - > [3, 5, 2, 6, **7, 8**] here 7<8 then swap their position.

It will iterate until the list is sorted.

Fourth Iteration -

[**3, 5**, 2, 6, 7, 8] - > [**3, 5,** 2, 6, 7, 8]

[3**, 5, 2**, 6, 7, 8] - > [3, **2, 5**, 6, 7, 8]

[3, 2, **5, 6**, 7, 8] - > [3, 2, **5, 6**, 7, 8]

[3, 2, 5, **6, 7**, 8] - > [3, 2, 5, **6, 7**, 8]

[3, 2, 5, **6, 7**, 8] - > [3, 2, 5, **6, 7**, 8]

Fifth Iteration

[**3, 2,** 5, 6, 7, 8] - > [**2, 3,** 5, 6, 7, 8]

Check the each element and as we can see that our list is sorted using the bubble sort technique.

Program

```
def bubble_sort(list1):
    for i in range(0,len(list1)-1):
        for j in range(len(list1)-1):
            if(list1[j]>list1[j+1]):
                temp = list1[j]
                list1[j] = list1[j+1]
                list1[j+1] = temp
    return list1
list1 = [5, 3, 8, 6, 7, 2]
print("The unsorted list is: ", list1)
# Calling the bubble sort function
print("The sorted list is: ", bubble_sort(list1))
```

Output:

```
The unsorted list is:  [5, 3, 8, 6, 7, 2]
The sorted list is:  [2, 3, 5, 6, 7, 8]
```

2. Discuss about selection sort in detail.

- Selection sort is a sorting we select the smallest element from an unsorted array in each pass and swap with the beginning of the unsorted array.
- This process will continue until all the elements are placed at right place. It is simple and an in-place comparison sorting algorithm.

Working of Selection Sort

Given a list of five elements, the following illustrate how the selection sort algorithm iterates through the values when sorting them.
The following shows the unsorted list

Step 1)

The first value 21 is compared with the rest of the values to check if it is the minimum value.

3 is the minimum value, so the positions of 21 and 3 are swapped. The values with a green background represent the sorted partition of the list.

Step 2)

The value 6 which is the first element in the unsorted partition is compared with the rest of the values to find out if a lower value exists

The value 6 is the minimum value, so it maintains its position.

Step 3)

The first element of the unsorted list with the value of 9 is compared with the rest of the values to check if it is the minimum value.

The value 9 is the minimum value, so it maintains its position in the sorted partition.

Step 4)

The value 33 is compared with the rest of the values.

The value 21 is lower than 33, so the positions are swapped to produce the above new list.

Step 5)

We only have one value left in the unpartitioned list. Therefore, it is already sorted.

The final list is like the one shown.

Selection Sort Program

```
def selection_sort(array):
  length = len(array)
   for i in range(length-1):
      minIndex = i
        for j in range(i+1, length):
    if array[j]<array[minIndex]:
          minIndex = j
```

```
    array[i], array[minIndex] = array[minIndex], array[i]
  return array
array = [21,6,9,33,3]
print("The sorted array is: ", selection_sort(array))
```

3.What is searching? Explain the linear search and binary search with python program and example.

- Searching is the process of selecting particular information from a collection of data based on specific criteria.

There are two types of searching.

1.Linear search

2.Binary search

3.7 LINEAR SEARCH

- The simplest form of searching is the sequential or linear search algorithm.
- This technique iterates over the sequence, one item at a time,until the specific item is found.
- In Python, a target item can be found in a sequence using the' in' operator.

```
if key in theArray :
    print( "The key is in the array." )
else :
    print( "The key is not in the array." )
```

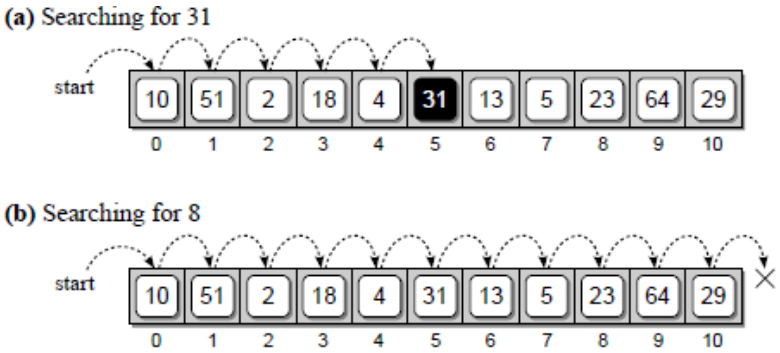


Figure 5.1: Performing a linear search on an unsorted array: (a) the target item is found and (b) the item is not in the array.

- To determine if value **31** is in the array, the search begins with the value in the first element.
- Since the first element does not contain the target value, the next element in sequential order is compared to value 31.

- This process is repeated until the item is found in the sixth position.
- For example, suppose we want to search for value 8 in the sample array.
- The search begins at the first entry as before, but this time every item in the array is compared to the target value.
- It cannot be determined that the value is not in the sequence until the entire array has been traversed.

Finding a Specific Item

Listing 5.1 Implementation of the linear search on an unsorted sequence.

```
1  def linearSearch( theValues, target ) :
2    n = len( theValues )
3    for i in range( n ) :
4       # If the target is in the ith element, return True
5      if theValues[i] == target
6        return True
7
8    return False   # If not found, return False.
```

- A loop is used to traverse through the sequence during which each element is compared against the target value.
- If the item is in the sequence, the loop is terminated and True is returned.
- Otherwise, a full traversal is performed and False is returned after the loop terminates.
- Assuming the sequence contains n items, the linear search has a worst case time of O(n)

- This is an example of a divide and conquer strategy.
- The algorithm starts by middle item of the sorted sequence resulting in one of three possible conditions: the middle item is the target value, the target value is less than the middle item, or the target is larger than the middle item.
- Since the sequence is ordered, we can eliminate half the values in the list when the target value is not found at the middle position.
- Let's have a step by step implementation of binary search.

We have a sorted list of elements, and we are looking for the index position of 45.

[12, 24, 32, 39, 45, 50, 54]

So, we are setting two pointers in our list. One pointer is used to denote the smaller value called **low** and the second pointer is used to denote the highest value called **high**.

Next, we calculate the value of the **middle** element in the array.

mid = (low+high)/2

Here, the low is 0 and the high is 7.

mid = (0+7)/2

mid = 3 (Integer)

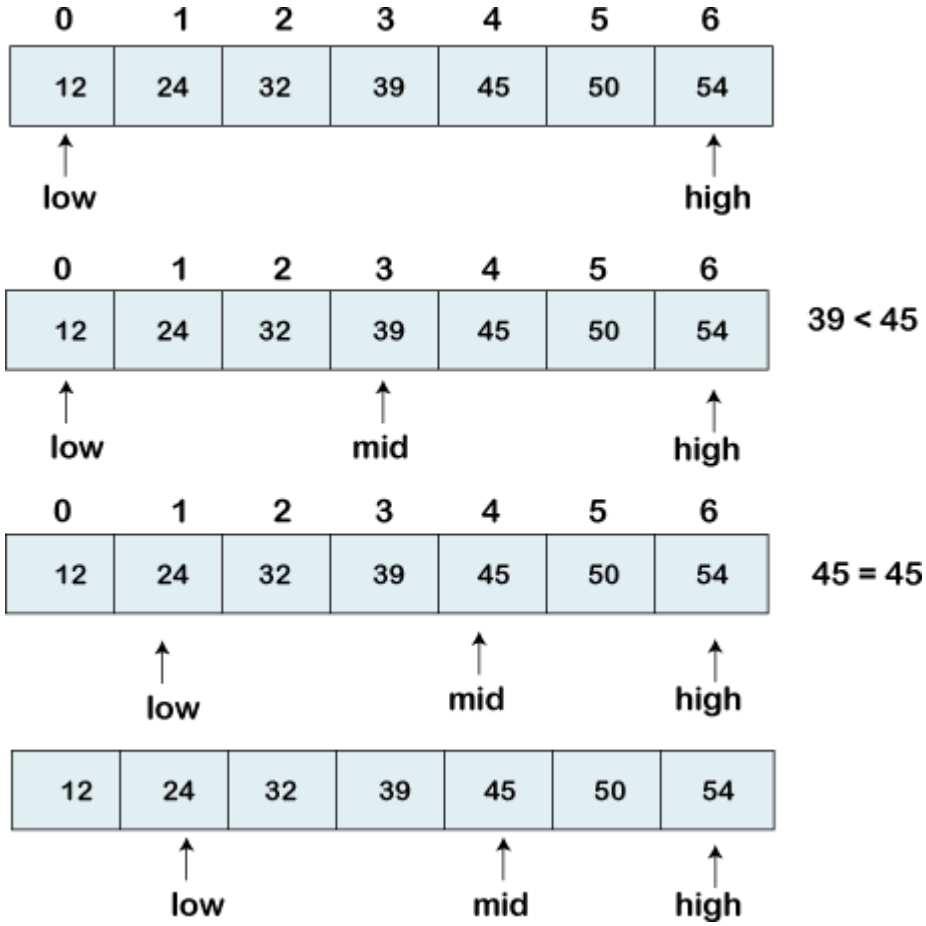
Now, we will compare the searched element to the mid index value. In this case, **32** is not equal to **45.** So we need to do further comparison to find the element.

If the number we are searching equal to the mid. Then return **mid** otherwise move to the further comparison.

The number to be search is greater than the **middle** number, we compare the **n** with the middle element of the elements on the right side of **mid** and set low to **low = mid + 1.**

Otherwise, compare the **n** with the **middle element** of the elements on the left side of **mid** and set **high** to **high = mid - 1.**

- The complexity of the binary search algorithm is **O(1)** for the best case.
- The **O(logn)** is the worst and the average case complexity of the binary search.

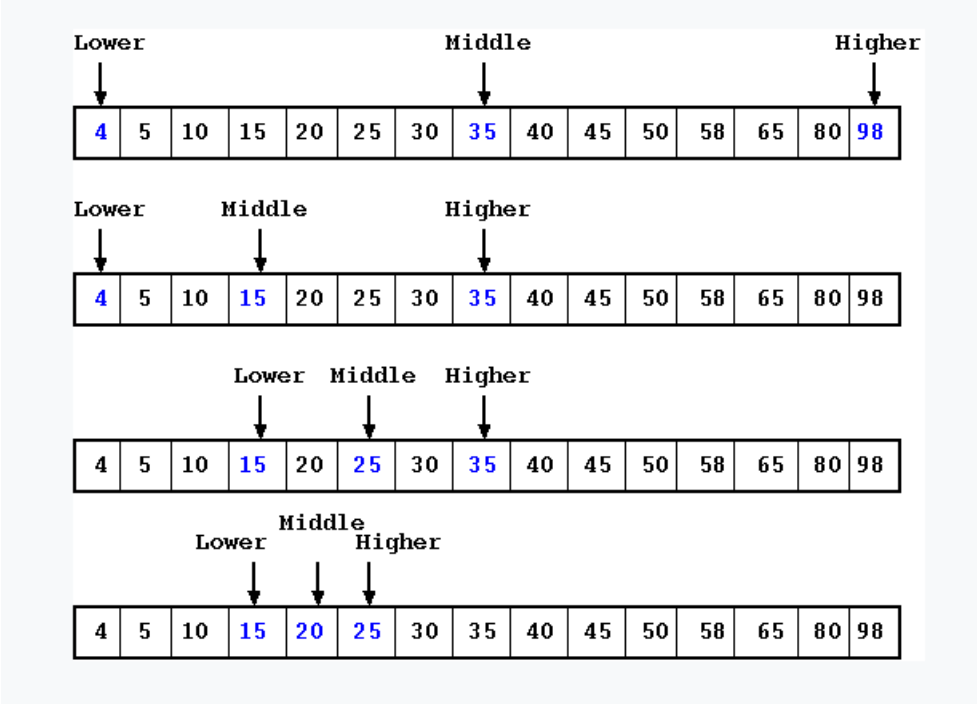

0 1 2 3 4 5 6
12 24 32 39 45 50 54
low
high
0 1 2 3 4 5 6
12 24 32 39 45 50 54
39 < 45
low
mid
high
0 1 2 3 4 5 6
12 24 32 39 45 50 54
45 = 45
low
mid
high
12 24 32 39 45 50 54
low
mid
high

Listing 5.4 Implementation of the binary search algorithm.

```
def binarySearch( theValues, target ) :
    # Start with the entire sequence of elements.
    low = 0
    high = len(theValues) - 1

    # Repeatedly subdivide the sequence in half until the target is found.
    while low <= high :
        # Find the midpoint of the sequence.
        mid = (high + low) // 2
        # Does the midpoint contain the target?
        if theValues[mid] == target :
            return True
        # Or does the target precede the midpoint?
        elif target < theValues[mid] :
            high = mid - 1
        # Or does it follow the midpoint?
        else :
            low = mid + 1

    # If the sequence cannot be subdivided further, we're done.
    return False
```



4. What is hashing? Explain open addressing and separate chaining methods of collision resolution techniques with examples.

(i) Separate Chaining

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

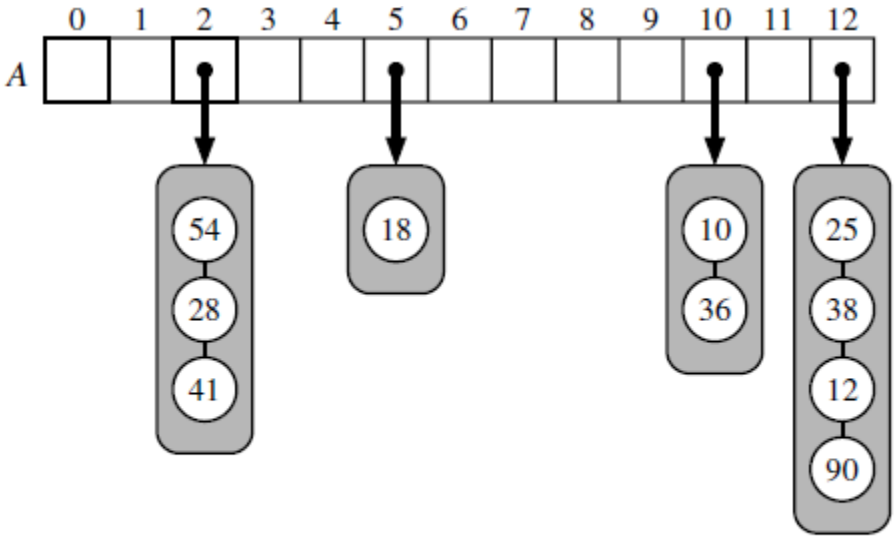


Figure 10.6: A hash table of size 13, storing 10 items with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

- 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.

(ii) Open Addressing

- 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, which we discuss next.
- 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.

5. The keys are stored in an array called a hash table and a hash function is associated with the table.

- If our hash function is good, then we expect the entries to be uniformly distributed in the *N* cells of the bucket array.
- Thus, 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*).
- In the worst case, a poor hash function could map every item to the same bucket. The basic command c = a + b involves two calls to __getiting in the dictionary for the local namespace to retrieve the values identified as a and b, and a call to __setitem__ to store the result associated with name c in that namespace.

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Table 10.2: Comparison of the running times of the methods of a map realized by means of an unsorted list (as in Section 10.1.5) or a hash table. We let n denote the number of items in the map, and we assume that the bucket array supporting the hash table is maintained such that its capacity is proportional to the number of items in the map.

6.Explain about Quick sort. Sort the following integer elements using Quick Sort 40, 20,70,14,60,61,97,30

- **Divide:**
  - If S has a sequence of elements, select a specific element x from S, which is called the pivot. As is common, 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 Of course, if the elements of S are distinct, then E holds just one element— the pivot itself.
- **Conquer**:
  - Recursively sort sequences L and G.
- **Combine:**
  - Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.

Working of Quicksort Algorithm:

1. Select the Pivot Element

- Here, we will be selecting the rightmost element of the array as the pivot element.

2. Rearrange the Array

- Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Here's how we rearrange the array:

a. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.
b. If the element is greater than the pivot element, a second pointer is set for that element

Now, pivot is compared with other elements. If an element smaller than the pivot element is

reached, the smaller element is swapped with the greater element found earlier.

a. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.
b. The process goes on until the second last element is reached.
c. Finally, the pivot element is swapped with the second pointer.

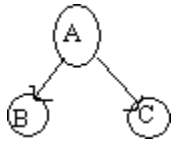
UNIT-IV
TREE STRUCTURES
PART-A

1. Define a tree

A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r, called the root, and zero or more nonempty (sub) trees T1, T2,…,Tk, each of whose roots are connected by a directed edge from r.

2. Define degree of the node

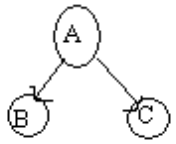
The total number of sub-trees attached to that node is called the degree of the node.



For node A, the degree is 2 and for B and C, the degree is 0.

3. Define leaves

These are the terminal nodes of the tree. The nodes with degree 0 are always the leaves.



Here, B and C are leaf nodes.

4. Define depth and height of a node

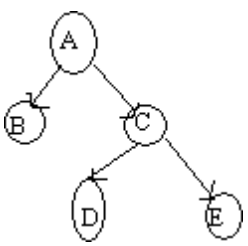
For any node ni, the depth of ni is the length of the unique path from the root to ni. The height of ni is the length of the longest path from ni to a leaf.

5. Define depth and height of a tree

The depth of the tree is the depth of the deepest leaf. The height of the tree is equal to the height of the root. Always depth of the tree is equal to height of the tree.

6. What do you mean by level of the tree?

The root node is always considered at level zero, then its adjacent children are supposed to be at level 1 and so on.
Here, node A is at level 0, nodes B and C are at level 1 and nodes D and E are at level 2.



7. Define a binary tree

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub-tree and right sub-tree.

8. Define a path in a tree

A path in a tree is a sequence of distinct nodes in which successive nodes are connected by edges in the tree.

9. Define a full binary and complete binary tree

A full binary tree is a tree in which all the leaves are on the same level and every non-leaf node has exactly two children.

A complete binary tree is a tree in which every non-leaf node has exactly two children not necessarily to be on the same level.

10. State the properties of a binary tree

- The maximum number of nodes on level n of a binary tree is 2n-1, where n≥1.
- The maximum number of nodes in a binary tree of height n is 2n-1, where n≥1.
- For any non-empty tree, nl=nd+1 where nl is the number of leaf nodes and nd is the number of nodes of degree 2.

11. What is meant by binary tree traversal?

Traversing a binary tree means moving through all the nodes in the binary tree, visiting each node in the tree only once.

12. What are the different binary tree traversal techniques?

- Preorder traversal
- Inorder traversal
- Postorder traversal
- Levelorder traversal

13. What are the tasks performed during inorder traversal?

- Traverse the left sub-tree
- Process the root node
- Traverse the right sub-tree

14. State the merits of linear representation of binary trees.

- Storage method is easy and can be easily implemented in arrays
- When the location of a parent/child node is known, other one can be determined easily
- It requires static memory allocation so it is easily implemented in all programming language

15. Define a binary search tree

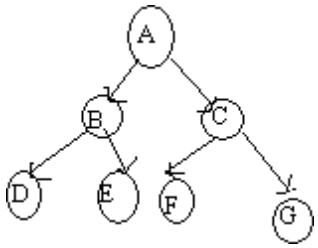
A binary search tree is a special binary tree, which is either empty or it should satisfy the following characteristics:

Every node has a value and no two nodes should have the same value i.e) the values in the binary search tree are distinct

- The values in any left sub-tree is less than the value of its parent node
- The values in any right sub-tree is greater than the value of its parent node

• The left and right sub-trees of each node are again binary search trees

16. Traverse the given tree using Inorder, Preorder and Postorder traversals.
Inorder :



Inorder : D H B E A F C I G J

Preorder: A B D H E C F G I J
Postorder: H D E B F I J G C A

17. Define AVL Tree.

AVL stands for Adelson-Velskii and Landis. An AVL tree is a binary search tree which has the following properties: 1.The sub-trees of every node differ in height by at most one. 2.Every sub-tree is an AVL tree. Search time is O(logn). Addition and deletion operations also take O(logn) time.

18. What do you mean by balanced trees?

Balanced trees have the structure of binary trees and obey binary search tree properties. Apart from these properties, they have some special constraints, which differ from one data structure to another. However, these constraints are aimed only at reducing the height of the tree, because this factor determines the time complexity.
Eg: AVL trees, Splay trees

19. What are the categories of AVL rotations?

Let A be the nearest ancestor of the newly inserted nod which has the balancing factor ±2. Then the rotations can be classified into the following four categories:
Left-Left: The newly inserted node is in the left subtree of the left child of A.
Right-Right: The newly inserted node is in the right subtree of the right child of A.
Left-Right: The newly inserted node is in the right subtree of the left child of A.
Right-Left: The newly inserted node is in the left subtree of the right child of A.

20. What is Heap Data Structure?

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element in existing heap time complexity O(log N).
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the most prior element in the heap, (max or min element for max and min heap).

PART-B

1.What is Binary search tree. Write an algorithm to find an element from binary search tree .

- A ***binary tree*** is an ordered tree with the following properties:
  1. Every node has at most two children.
  2. Each child node is labeled as being either a ***left child*** or a ***right child***.
  3. 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 an internal node *v* is called a ***left subtree*** or ***right subtree***, respectively, of *v*.
- A binary tree is ***proper*** if each node has either zero or two children.
- Some people also refer to such trees as being ***full*** binary trees. Thus, in a proper binary tree, every internal node has exactly two children.
- A binary tree that is not proper is ***improper***.

A Recursive Binary Tree Definition

- we can also define a binary tree in a recursive way such that 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 abstract data type, a binary tree is a specialization of a tree that supports three additional accessor methods:

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

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

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

Properties of Binary Trees

- Binary trees have several interesting 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 ***level*** *d* of *T*.
- In a binary tree, level 0 has at most one node (the root), level 1 has at most two nodes (the children of the root), level 2 has at most four nodes, and so on.

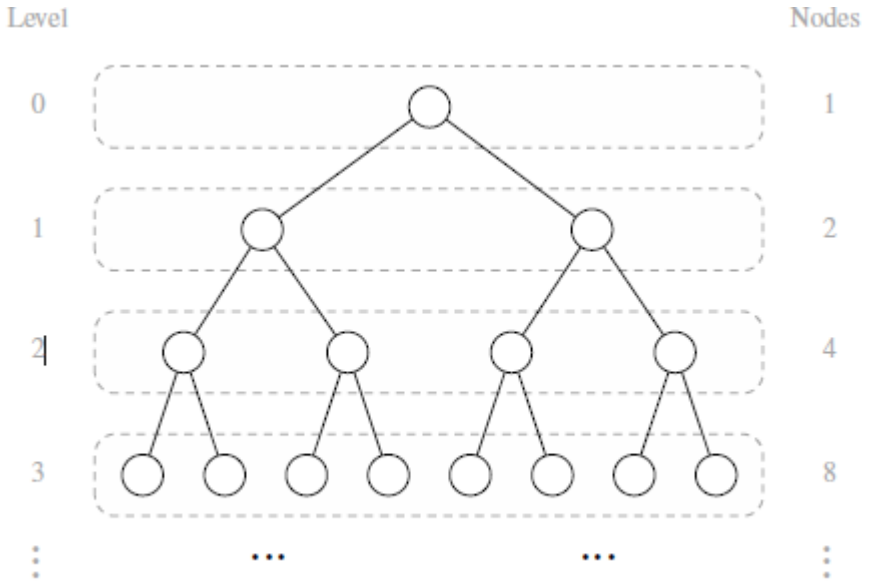


Figure 8.9: Maximum number of nodes in the levels of a binary tree.

Proposition 8.8: *Let T be a nonempty binary tree, and let n, n_E, n_I and 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. $h+1 \leq n \leq 2^{h+1}-1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h-1$
4. $\log(n+1)-1 \leq h \leq n-1$

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

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

2.What are the tree traversal techniques? Explain with an example.

- A ***traversal*** of a tree *T* is a systematic way of accessing, or "visiting," all the positions of *T*.
- There are3 types of traversal
    1. Preorder traversal
    2. Inorder traversal
    3. Post order travsersal

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, we can repeat the same process on each node, resulting in a recursive solution.

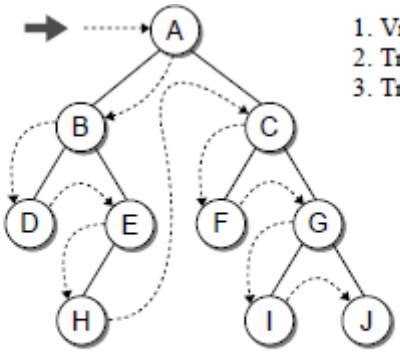


Figure 13.12: The logical ordering of the nodes with a preorder traversal.

- Consider the binary tree in Figure . The dashed lines show the logical order the nodes would be visited during the traversal: A, B, D, E, H, C, F, G, I, J. This traversal is known as a preorder traversal since we first visit the node followed by the subtree traversals.

Algorithm for preorder traversal

Listing 13.2 Preorder traversal on a binary tree.

```
1  def preorderTrav( subtree ):
2    if subtree is not None :
3      print( subtree.data )
4      preorderTrav( subtree.left )
5      preorderTrav( subtree.right )
```

- In the algorithm 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 subtrees are traversed.
- By convention, the left subtree is always visited before the right subtree.
- 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.

Inorder Traversal

- Another traversal that can be performed is the inorder traversal, in which we first traverse the left subtree and then visit the node followed by the traversal of the right subtree.

Listing 13.3 Inorder traversal on a binary tree.

```
1  def inorderTrav( subtree ):
2    if subtree is not None :
3      inorderTrav( subtree.left )
4      print( subtree.data )
5      inorderTrav( subtree.right )
```

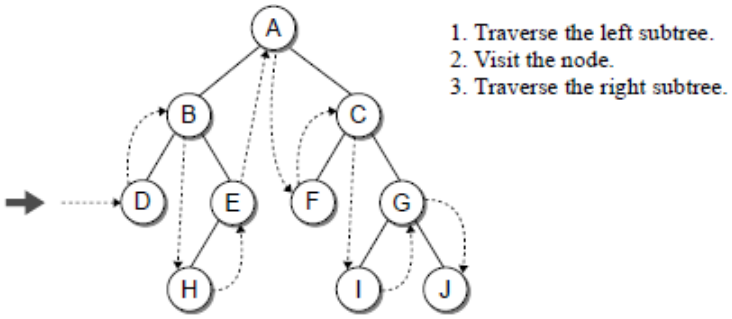


Figure 13.13: The logical ordering of the nodes with an inorder traversal.

- Figure shows the logical ordering of the node visits in the example tree: D, B, H, E, A, F, C, I, G, J

Postorder Traversal

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

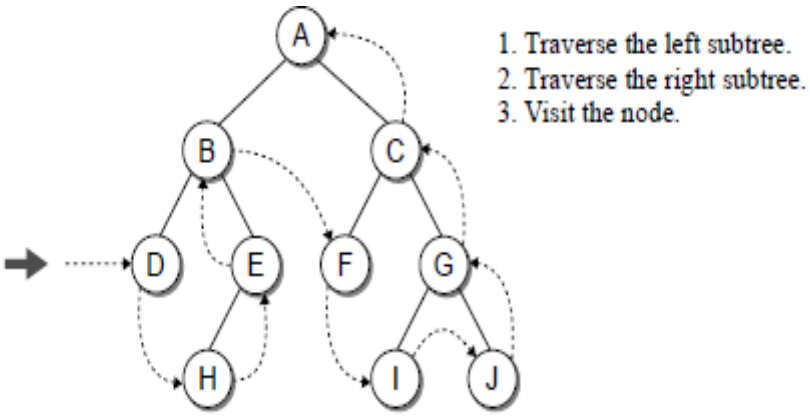


Figure 13.14: The logical ordering of the nodes with a postorder traversal.

- The example tree with the logical ordering of the node visits in a postorder traversal is shown in Figure .
- The nodes are visited in this order: D, H, E, B, F, I, J, G, C, A.
- You may notice that the root node is always visited first in a preorder traversal but last in a postorder traversal.

ALgorithm:

Listing 13.4 Postorder traversal on a binary tree.

```
1  def postorderTrav( subtree ):
2    if subtree is not None :
3      postorderTrav( subtree.left )
4      postorderTrav( subtree.right )
5      print( subtree.data )
```

3.Explain the AVL trees with the rotations used in it .

- The AVL tree, which was invented by G. M. Adel'son-Velskii and Y. M. Landis in 1962, improves on the binary search tree 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:

left high: When the left subtree is higher than the right subtree.
equal high: When the two subtrees have equal height.
right high: When the right subtree is higher than the left subtree.

- The balance factors of the tree nodes in our illustrations are indicated by symbols: > for a left high state, = for the equal high state, and < for a right high state.
- When a node is out of balance, we will use either << or >> to indicate which subtree is higher.

- The search and traversal operations are the same with an AVL tree as with a binary search tree.
- The insertion and deletion operations have to be modified in order to maintain the balance property of the tree as new keys are inserted and existing ones removed.

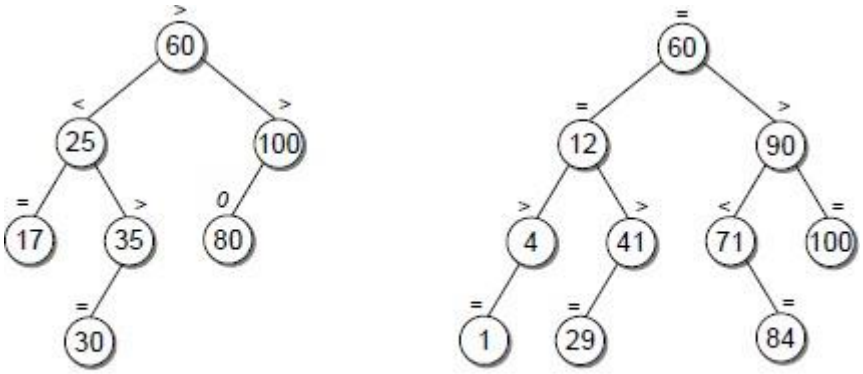


Figure 14.14: Examples of balanced binary search trees.

Insertions

- Inserting a key into an AVL tree begins with the same process used with a binary search tree.
- We search for the new key in the tree and add a new node at the child link where we fall of the tree.
- 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.
- For example, suppose we want to add key 120 to the sample AVL tree from Figure 14.14(a).
- Following the insertion operation of the binary search tree, the new key will be inserted as the right child of node 100, as illustrated in Figure 14.15(a). 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.
- Figure 14.15(b) shows the new balance factors after key 120 is added.

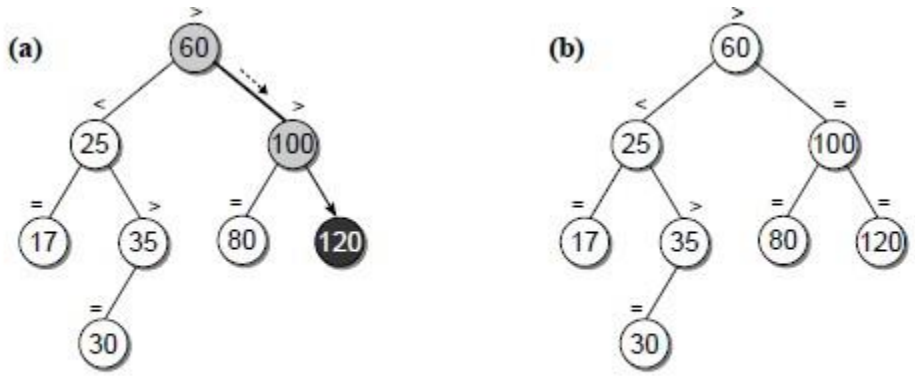


Figure 14.15: A simple insertion into an AVL tree: (a) with key 120 inserted; and (b) the new balance factors.

- Assume we need to add key 28 to the AVL.The new node 28 is inserted as the left child of node 30, as illustrated in Figure 14.16(a).
- When the balance factors are recalculated, as in Figure 14.16(b), we can see all of the subtrees along the path that are 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, as illustrated in Figure 14.16(c).

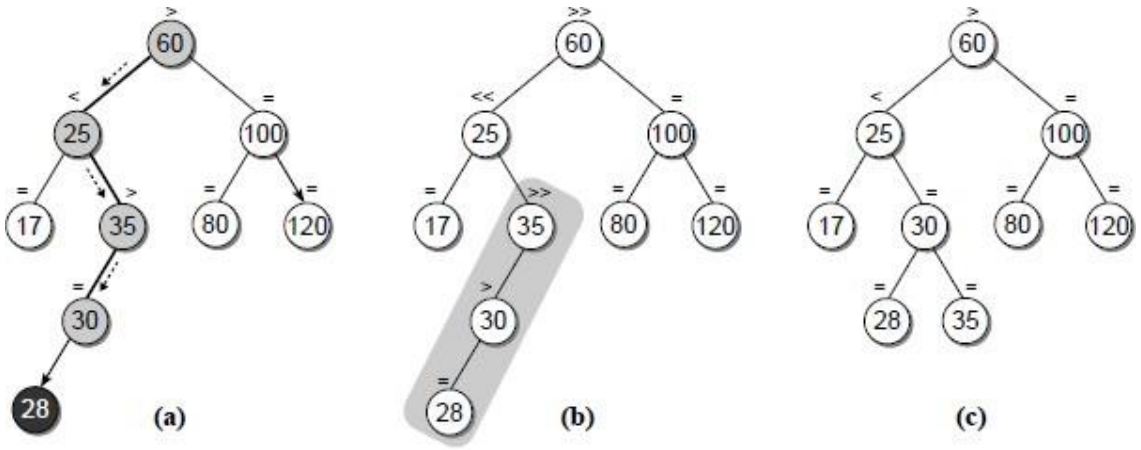


Figure 14.16: An insertion that causes the AVL tree to become unbalanced: (a) the new key is inserted; (b) the balance factors showing an out-of-balance tree; and (c) the subtree after node 35 is rearranged.

Rotations

- 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.
- The root node of this subtree is known as the pivot node.

There are four possible cases:

Case 1: This case, as illustrated in Figure 14.17, occurs 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.

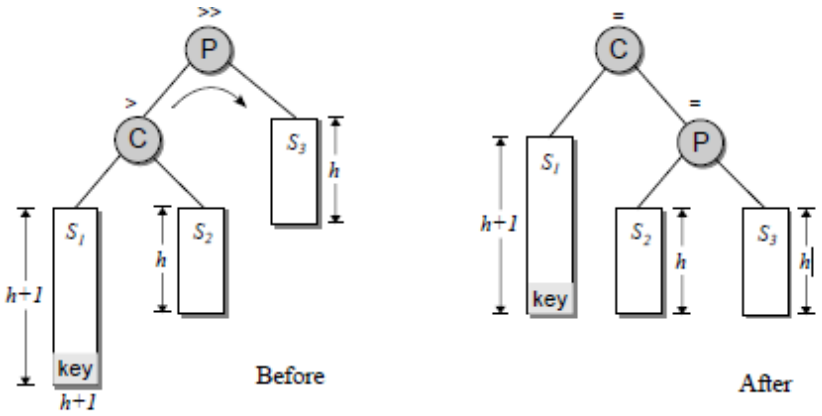


Figure 14.17: Case 1: a right rotation of the pivot node over its left child.

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. This case, which is illustrated in Figure 14.18, requires two rotations. Node C has to be 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 be the new left child of G.

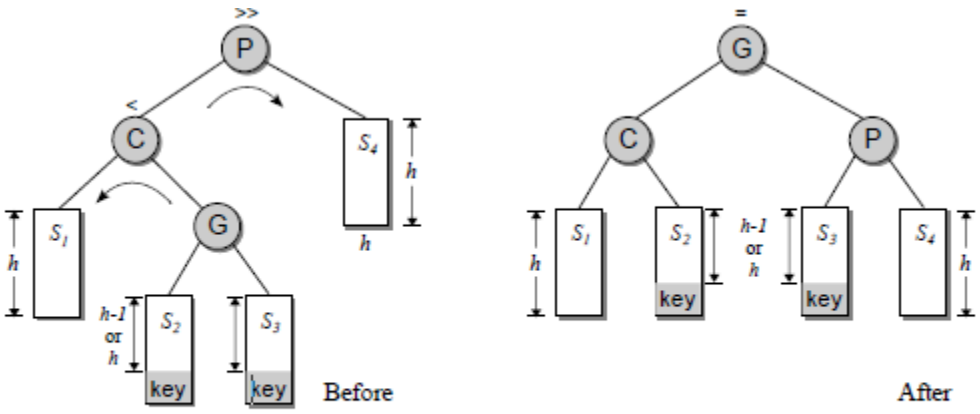


Figure 14.18: Case 2: a double rotation with the pivot's left child rotated left over its right child and the pivot rotated right over its left child.

Cases 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. The two cases are illustrated in Figure 14.19.

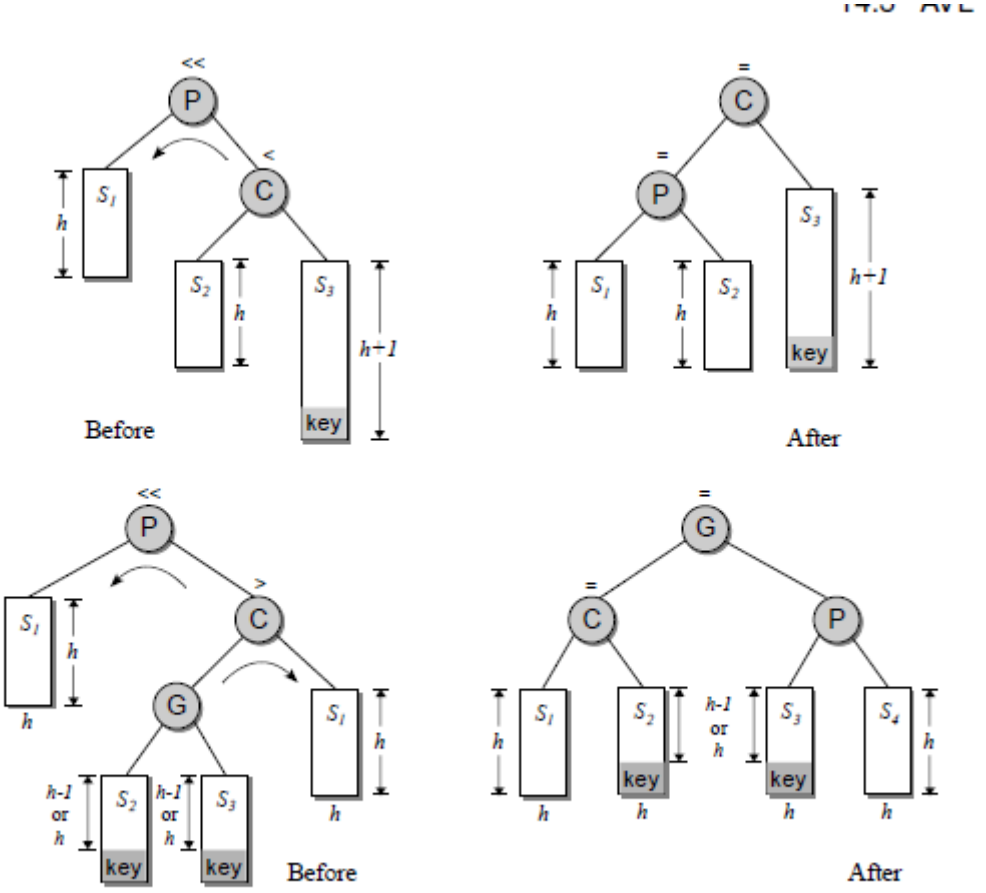


Figure 14.19: Cases 3 (top) and 4 (bottom) are mirror images of cases 1 and 2.

4. Explain the Basic operations performed in a heap.

- A heap is a binary tree *T* that stores a collection of items at its positions and that satisfies two additional properties:
  (i)a relational property defined in terms of the way keys are stored in *T* and a (ii)**structural property** 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 (namely, level i has $2i$ nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible positions at that level.

- A **max-heap** has the property, known as the heap order property, that for each non-leaf node V , the value in V is greater than the value of its two children.
- The **min-heap** has the opposite property. For each non-leaf node V , the value in V is smaller than the value of its two children.

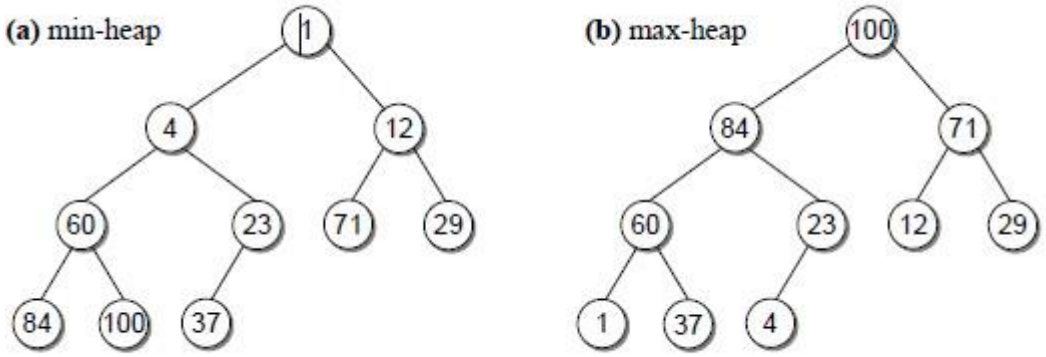


Figure 13.21: Examples of a heap.

Insertions

- When a new value is inserted into a heap, the heap order property and the heap shape property (a complete binary tree) must be maintained.
- Suppose we want to add value 90 to the max-heap in Figure 13.21(b). If we are to maintain the property of the max-heap, there are only two places in the tree where 90 can be inserted, as shown in Figure 13.22(a).
- Contrast this to the possible locations if we were to add value 41 to the max-heap, shown in Figure 13.22(b).

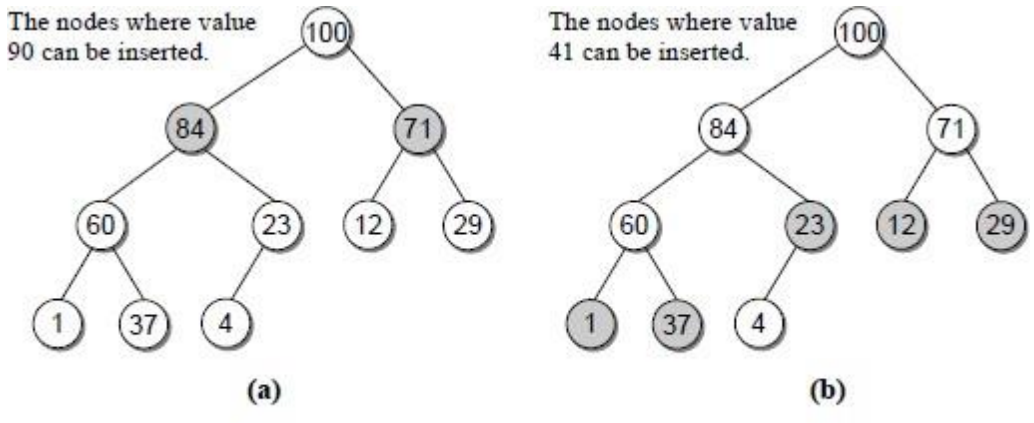


Figure 13.22: Candidate locations in a heap for new values.

- If we insert 90 into the heap, First, we create a new node and fil it with the new value as shown in part (a).
- The node is then attached as a leaf node at the only spot in the tree where the heap shape property can be maintained (part (b)).
- Remember, a heap is a complete tree and in such a tree, the leaf nodes on the lowest level must be filled from left to right.

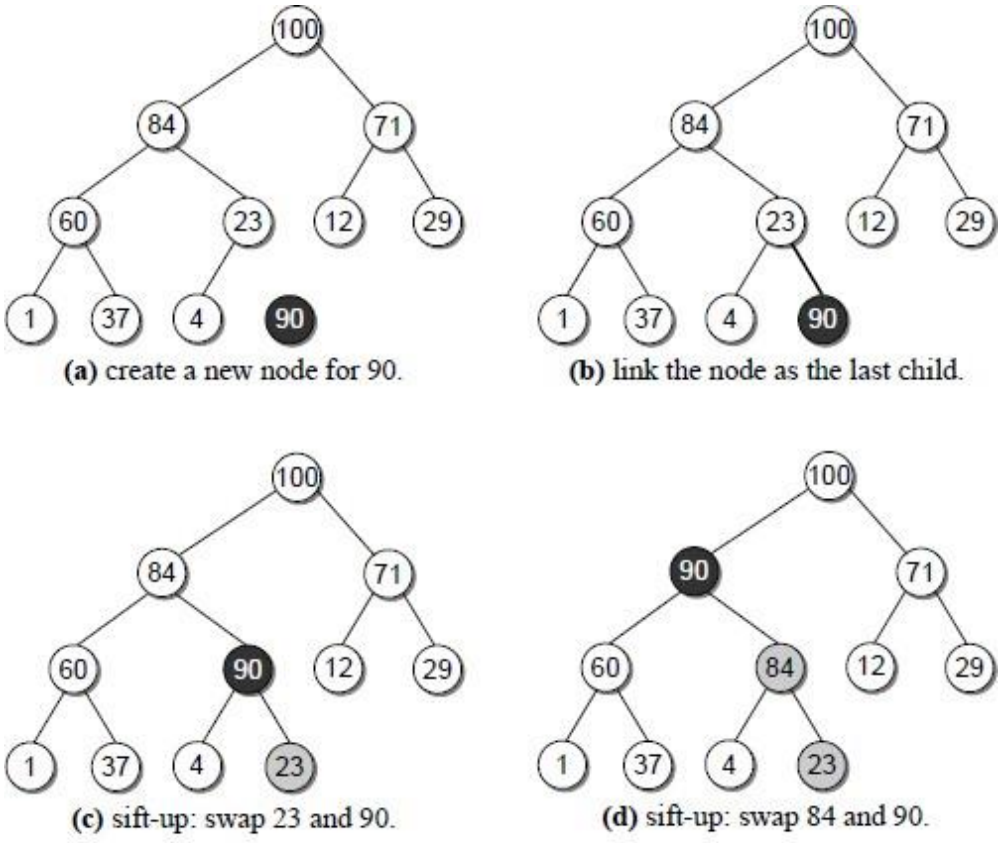


Figure 13.23: The steps to insert value 90 into the heap.

- Now, suppose we add value 41 to the heap, as illustrated in Figure 13.24.
- The new node is created and filled with value 41 and linked into the tree as the left child of node 12.
- When the new value is sifted up, we find values 12 and 41 have to be swapped, resulting in the final placement of the new value.

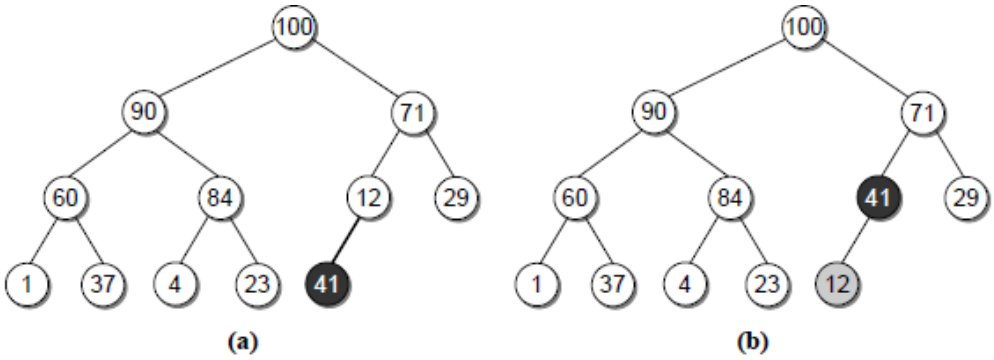


Figure 13.24: Inserting value 41 into the heap: (a) create the new node and link it into the tree; and (b) sift the new value up the tree.

Extractions(Removing)

- When a value is extracted and removed from the heap, it can only come from the root node.
- Thus, in a max-heap, we always extract the largest value and in a min-heap, we always extract the smallest value.
- After the value in the root has been removed, the binary tree is no longer a heap since there is now a gap in the root node, as illustrated in Figure 13.25.

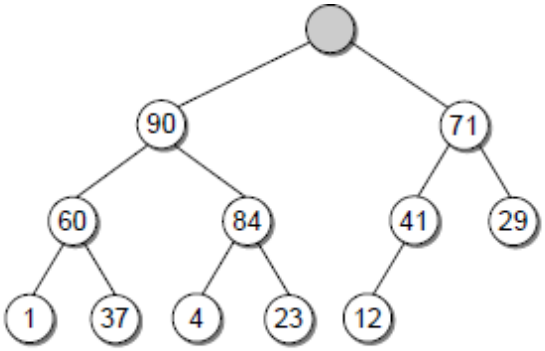


Figure 13.25: Extracting a value from the max-heap leaves a hole at the root node.

5.Write short notes on Multi-way search trees.

- General trees can be used as multiway search trees.Map items stored in a search tree are pairs of the form (*k*,*v*), where *k* is the ***key*** and *v* is the ***value*** associated with the key.
- Let *w* be a node of an ordered tree. We say that *w* is a ***d-node*** if *w* has *d* children.
- We define a multiway search tree to be an ordered tree *T* that has the following properties, which are illustrated in Figure 11.23a:

• Each internal node of *T* has at least two children. That is, each internal node is a *d*-node

such that $d \geq 2$.

• Each internal d-node w of T with children $c1, \ldots, cd$ stores an ordered set of $d-1$ key-value pairs $(k1,v1), \ldots, (kd-1,vd-1)$, where $k1 \leq \cdots \leq kd-1$.

• Let us conventionally define $k0 = -\infty$ and $kd = +\infty$. For each item (k,v) stored at a node in the subtree of w rooted at ci, $i = 1, \ldots, d$, we have that $ki-1 \leq k \leq ki$.

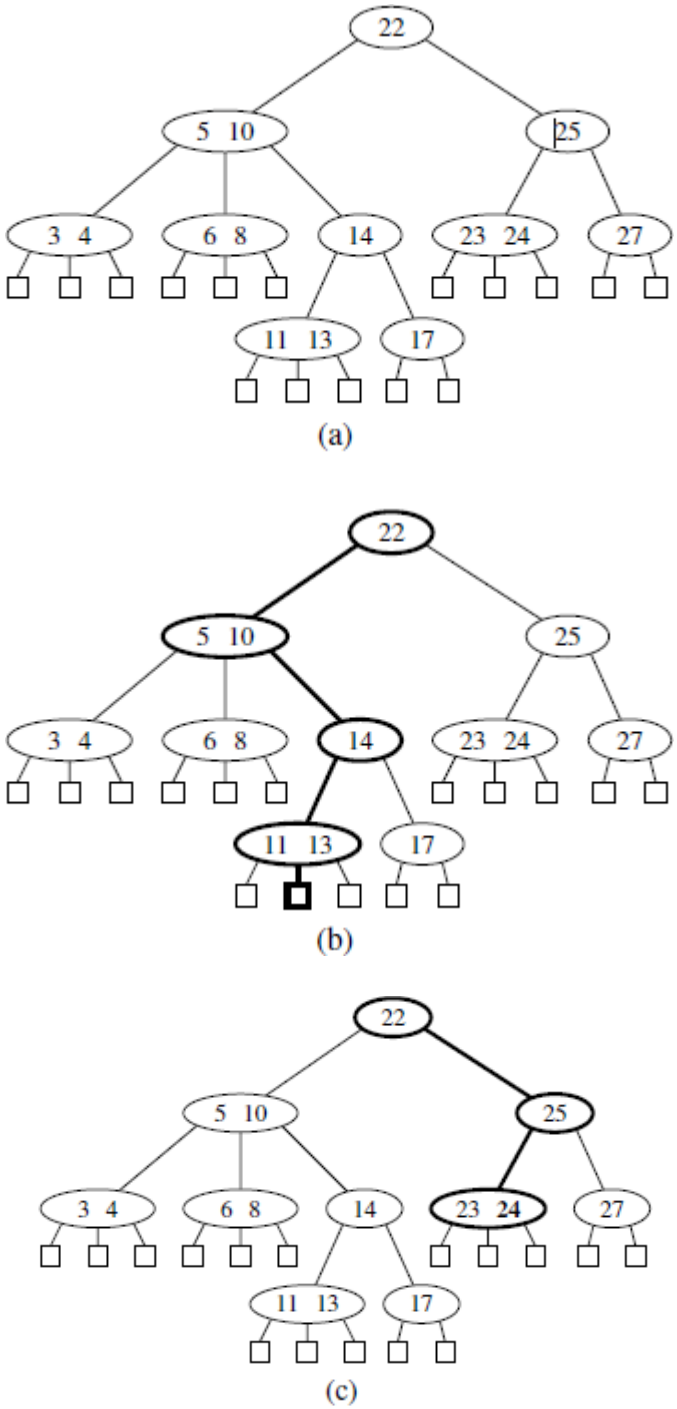


Figure 11.23: (a) A multiway search tree T; (b) search path in T for key 12 (unsuccessful search); (c) search path in T for key 24 (successful search).

Searching in a Multiway Tree

- Searching for an item with key k in a multiway search tree T is simple.
- We perform such a search by tracing a path in T starting at the root. (See Figure 11.23b and c.)
- When we are at a d-node w during this search, we compare the key k with the keys $k1$, . . . ,kd−1 stored at w. If $k = ki$ for some i, the search is successfully completed.

- Otherwise, we continue the search in the child *ci* of *w* such that *ki*−1 < *k* < *ki*.
- If we reach an external node, then we know that there is no item with key *k* in *T*, andthe search terminates unsuccessfully.

UNIT-IV
GRAPH STRUCTURES
PART-A

1. Define Graph.

A graph G consist of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set for edge E to a set of pairs of elements of V. It can also be represented as G=(V, E).

2. Define adjacent nodes.

Any two nodes which are connected by an edge in a graph are called adjacent nodes. For example, if an edge x ε E is associated with a pair of nodes (u,v) where u, v ε V, then we say that the edge x connects the nodes u and v.

3. What are the two traversal strategies used in traversing a graph?

a.Breadth first search

b. Depth first search

4. Name two algorithms two find minimum spanning tree

1. Kruskal'salgorithm
2. Prim's algorithm

5. List the two important key points of depth first search.

i) If path exists from one node to another node, walk across the edge – exploring the edge.

ii) If path does not exist from one specific node to any other node, return to the previous node where we have been before – backtracking.

6. Differentiate BFS and DFS.

No.	DFS	BFS
1.	Backtracking is possible from a dead end	Backtracking is not possible
2.	Vertices from which exploration is incomplete are processed in a	The vertices to be explored are organized as a
3.	Search is done in one particular direction	The vertices in the same level are maintained

7. Define biconnectivity.

A connected graph G is said to be biconnected, if it remains connected after removal of any one vertex and the edges that are incident upon that vertex. A connected graph is biconnected, if it has no articulation points.

8. Define adjacency list.

Adjacency list is an array indexed by vertex number containing linked lists. Each node Vi the ith array entry contains a list with information on all edges of G that leave Vi. It is used to represent the graph related problems.

9. What is DAG?

A DAG is a graph that flows in one direction, where no element can be a child of itself. So most of us are familiar with LinkedLists, trees, and even graphs. A DAG is very similar to the first two, and an implementation of the third.

A DAG will have 4 things:

1. Nodes: A place to store the data.
2. Directed Edges: Arrows that point in one direction (the thing that makes this data structure different)
3. Some great ancestral node with no parents. (Fun fact: Most ancestry trees are actually DAGs and not actually trees because cousins at some point get married to each other.)
4. Leaves: Nodes with no children

10. What is Dynamic programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

11. What is a minimum spanning tree?

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at the lowest total cost.

12. Define indegree and out degree of a graph?

In a directed graph, for any node v, the number of edges, which have v as their initial node, is called the out degree of the node v. Outdegree: Number of edges having the node v as root node is the outdegree of the node v.

13. Name the different ways of representing a graph?

a. Adjacency matrix

b. Adjacency list

14. What do you mean by shortest path?

A path having minimum weight between two vertices is known as shortest path, in which weight is always a positive number.

15. Define adjacency list.

Adjacency list is an array indexed by vertex number containing linked lists. Each node Vi the ith array entry contains a list with information on all edges of G that leave Vi. It is used to represent the graph related problems.

16. What is the use of BFS?

BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph. The shortest distance is the minimum number of edges traversed in order to travel from the start node the specific node being examined.

17. What is topological sort?

It is an ordering of the vertices in a directed acyclic graph, such that: If there is a path from u to v, then v appears after u in the ordering.

18. What is a directed and undirected graph?

A graph in which every edge is directed is called a directed graph.

A graph in which every edge is undirected is called an undirected graph.

19. What is a cycle or a circuit acyclic graph?

A path which originates and ends in the same node is called a cycle or circuit. A simple diagram, which does not have any cycles, is called an acyclic graph.

20. What is meant by strongly connected in a graph?

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

PART-B

1. Explain the various representation of graph with example in detail?

Graph Representations

Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Adjacency List**
3. **Adjacency Multilists**

1.Adjacency Matrix

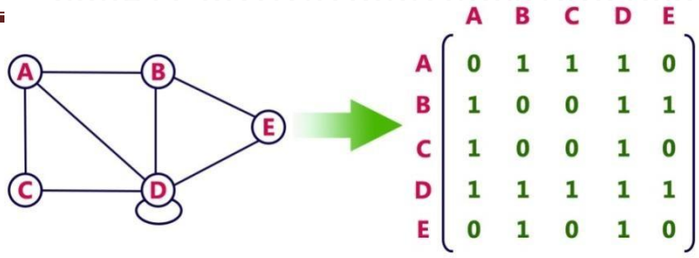
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

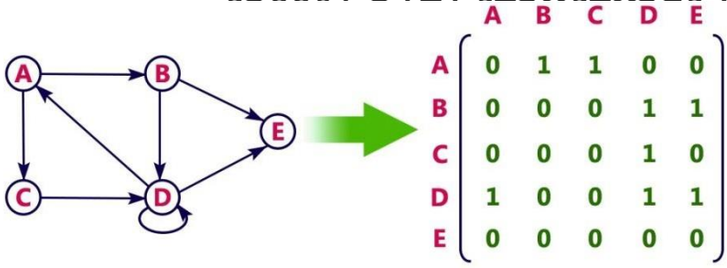
This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff $(v_i, v_j) \in E(G)$ ($\langle v_i, v_j \rangle$ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph



For a Directed graph



The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

Merits of Adjacency Matrix:

From the adjacency matrix, to determine the connection of vertices is easy

The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree

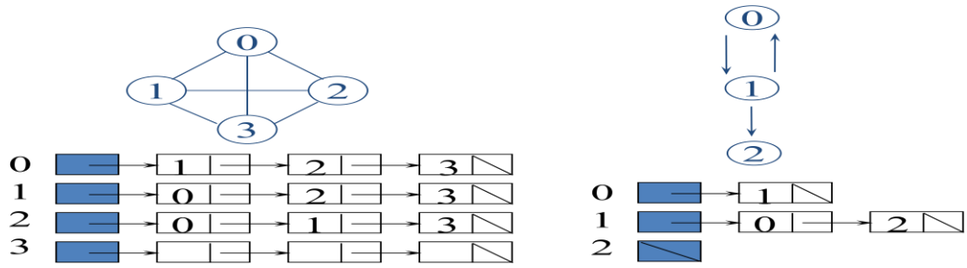
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:

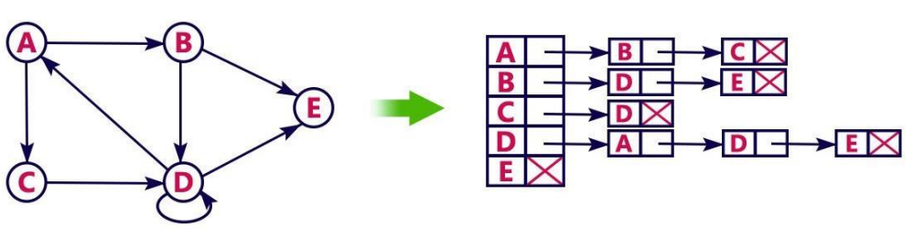


So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. Structure is

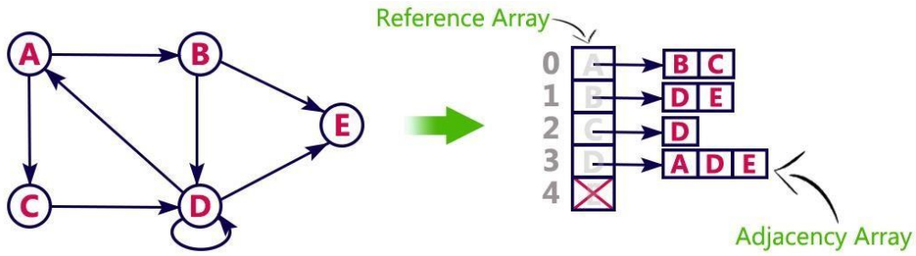
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
   int vertex;
   struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Another type of representation is given below.

example: consider the following directed graph representation implemented using linked list

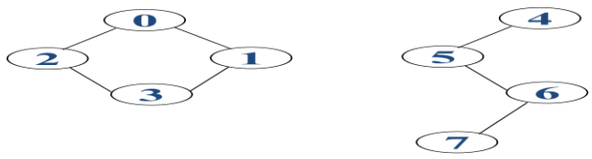


This representation can also be implemented using array.



Sequential representation of adjacency list is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

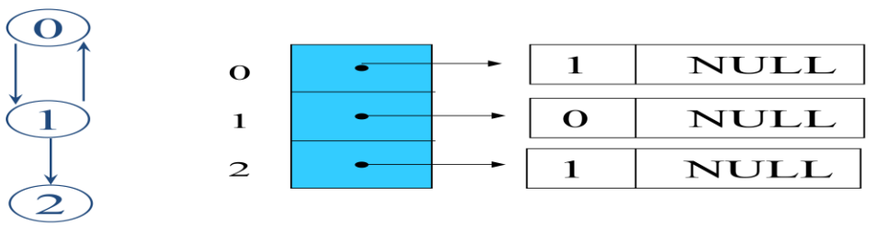


Graph

Instead of chains, we can use sequential representation into an integer array with size n+2e+1. For 0<=i<n, Array[i] gives starting point of the list for vertex I, and array[n] is set to n+2e+1. The adjacent vertices of node I are stored sequentially from array[i].

For an undirected graph with n vertices and e edges, linked adjacency list requires an array of size n and 2e chain nodes. For a directed graph, the number of list nodes is only e. the out degree of any vertex may be determined by counting the number of nodes in its adjacency list. To find in-degree of vertex v, we have to traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.



Determine in-degree of a vertex in a fast way.

3.Adjacency Multilists

In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one on the list for u and the other on tht list for v. As we shall see in some situations it is necessary to be able to determin ie ~ nd enty for a particular edge and mark that edg as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

For adjacency multilists, node structure is

```
typedef struct edge *edge_pointer;
typedef struct edge {
   short int  marked;
   int vertex1, vertex2;
   edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

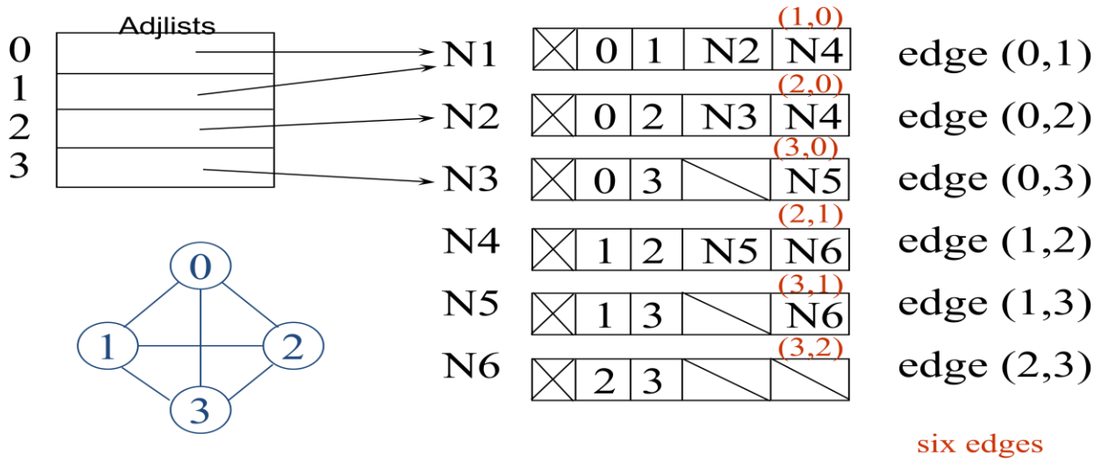
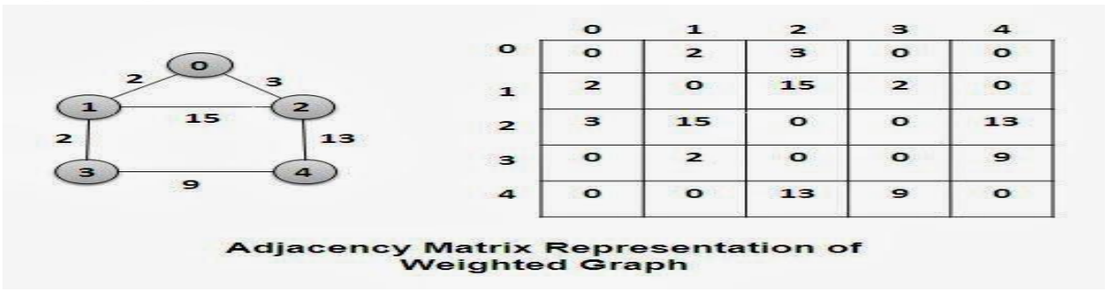


Figure: Adjacency multilists for given graph

4. Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list'nodes by including an additional field weight. A graph with weighted edges is called a network.



2. Explain Graph ADT with example?

Graph Terminology

1. **Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2. **Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1. Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

2. Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

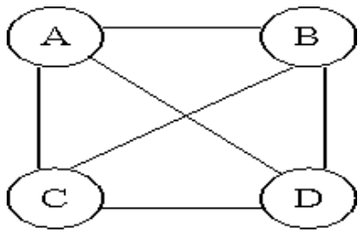
3. Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1. Undirected

Graph

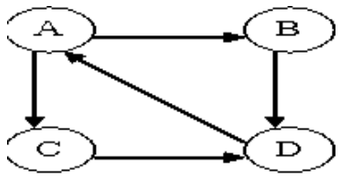
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

3. Directed Graph

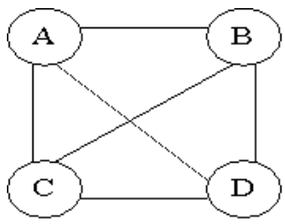
A graph with only directed edges is said to be directed graph.



Directed Graph.

4. Complete Graph

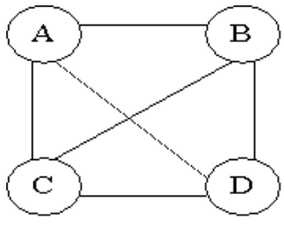
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

5. Regular Graph

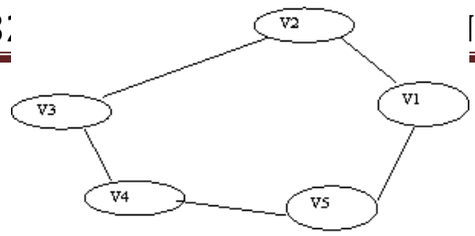
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

6. Cycle Graph

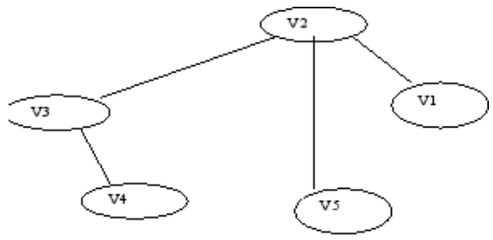
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



A cycle graph

7. Acyclic Graph

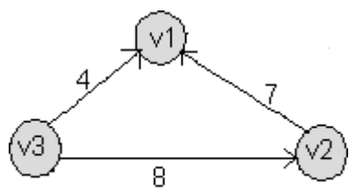
A graph without cycle is called acyclic graphs.



A acyclic graph

8. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Adjacent nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

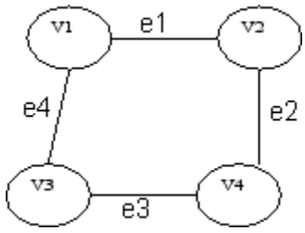
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk

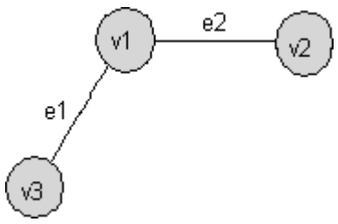
A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

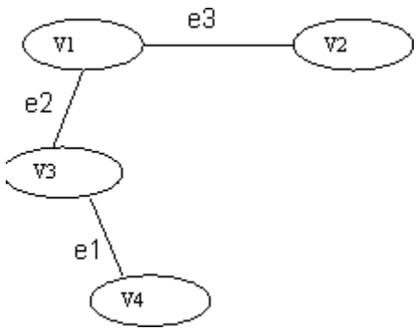
A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

The number of edges in a path is called the length of that path. In the following, the length of the path is 3.

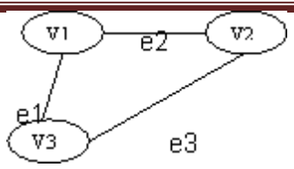


An open walk Graph

Circuit

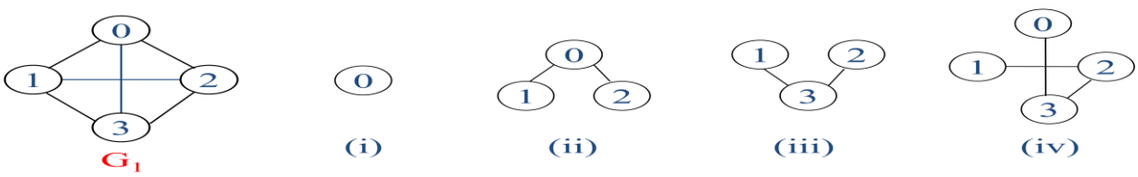
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

A circuit having three vertices and three edges.



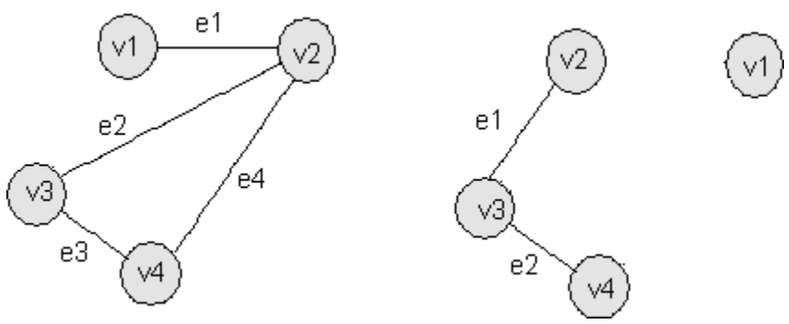
Sub Graph

A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G)



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.



A connected graph G A disconnected graph G

This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

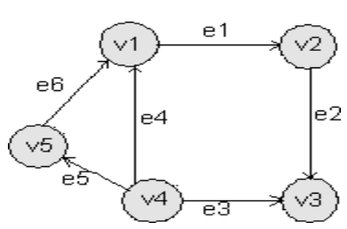
Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Indegree

The indegree of a node is the number of edges connecting to that node or in other words edges incident to it.



In the above graph,the indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the

ADT of Graph:

Structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

3. What is Graph traversal?Explain Depth first traversal with algorithm.

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Depth-First Search

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack
- Time complexity
  - Adjacency list: O(|E|)
  - Adjacency matrix: $O(|V|^2)$

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

This function is best described recursively as in Program.

```
#define FALSE 0
#define TRUE 1
 int visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depth-first search is initiated from vertex 0 then the vertices of G are visited in the following order: **0 1 3 7 4 5 2 6.** Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

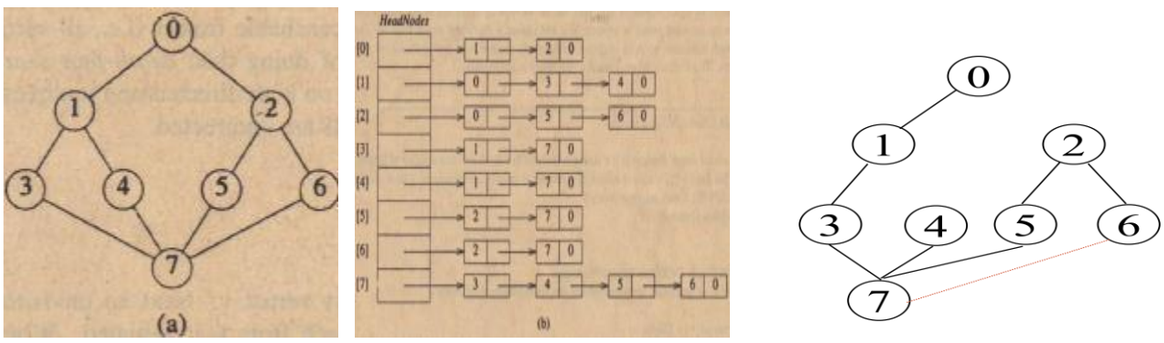


Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:

When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is $O(n^2)$.

Breadth-First Search

In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS (Program 6.2) gives the details.

```
typedef struct queue *queue_pointer;
typedef struct queue {
   int vertex;
```

```
    queue_pointer link;
};
void addq(queue_pointer *,
        queue_pointer *, int);
int deleteq(queue_pointer *);
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%d", v);
  visited[v] = TRUE;
  addq(&front, &rear, v);
while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
      printf("%d", w->vertex);
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

Steps:

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Analysis Of BFS:

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, $O(n^2)$. If adjacency lists are used the loop has a total cost of $d_0 + \ldots + d_{n-1} = O(e)$, where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

3.Connected Components

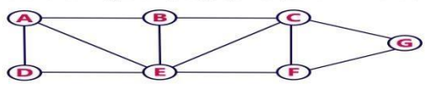
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet been visited. This leads to function Connected(Program 6.3), which determines the connected components of G. The algorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void connected(void){
   for (i=0; i<n; i++) {
       if (!visited[i]) {
           dfs(i);
  printf("\n");        }     } }
```

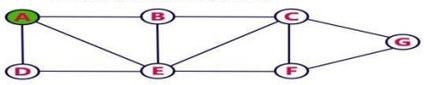
Analysis of Components:

If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used,then the time required is $O(n^2)$

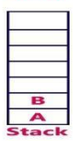
Consider the following example graph to perform DFS traversal



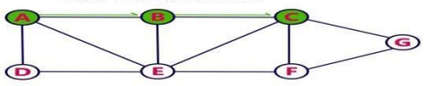
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

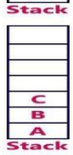


Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

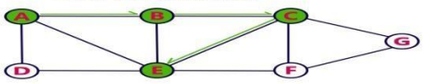


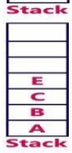
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



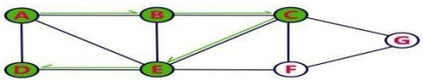


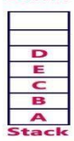
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



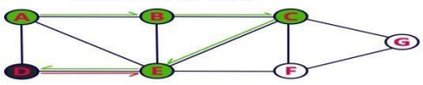


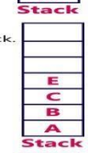
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



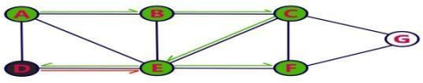


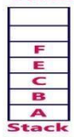
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



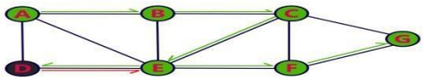


Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

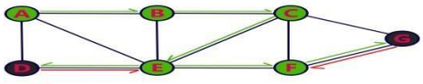


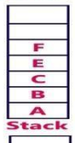


Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

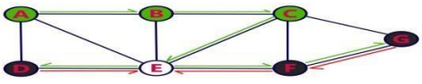


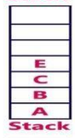
Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



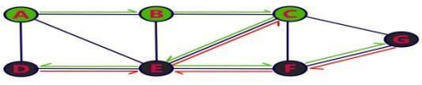


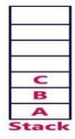
Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



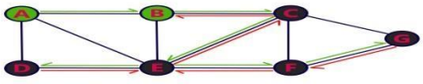


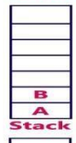
Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



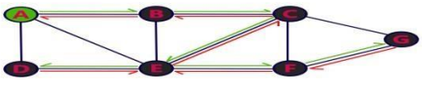


Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

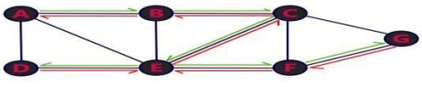


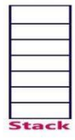


Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

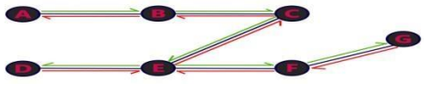


Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

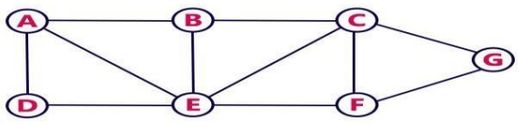




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

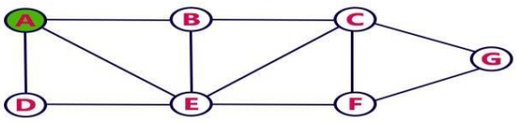


Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

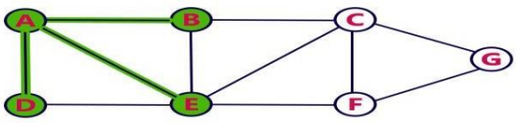


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

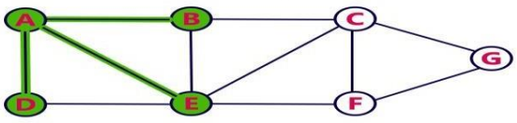


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

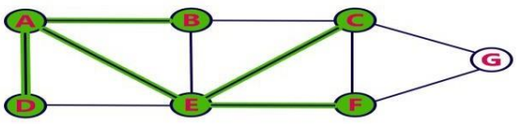


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

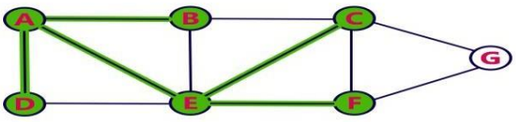


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

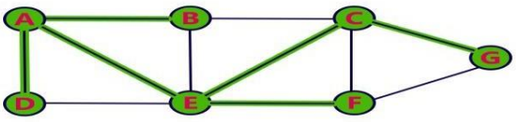


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

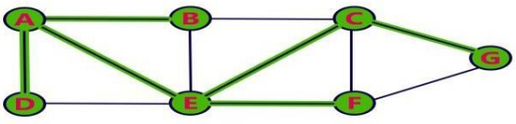


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

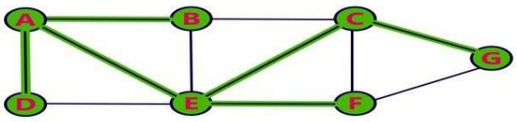


Queue

						G

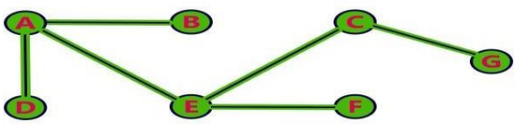
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



4. What is topological sort? Explain topological sort with suitable example?

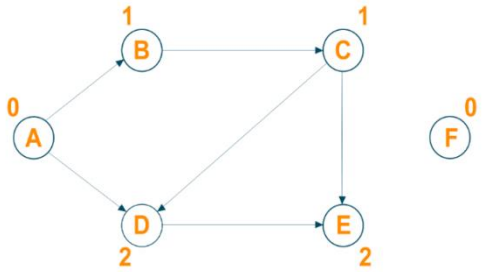
Topological Ordering

Topological sort is an algorithm that takes a directed acyclic graph and returns the sequence of nodes where every node will appear before other nodes that it points to. Just to remind, a directed acyclic graph (DAG) is the graph having directed edges from one node to another but does not contain any directed cycle. Remember topological sorting for graphs is not applicable if the graph is not a Directed Acyclic Graph (DAG). The ordering of the nodes in the array is called topological ordering. Therefore we can say that a topological sort of the nodes of a directed acyclic graph is the operation of arranging the nodes in the order in such a way that if there exists an edge (i,j), i precedes j in the lists. A topological sort basically gives a sequence in which we should perform the job and helps us to check whether the graph consists of the cycle or not.

Every graph can have more than one topological sorting possible. It depends on the in-degree of the node in the graph. Also, the topological sorting of the graph starts with the node that has in-degree as 0 i.e a node with no incoming edges. Let us learn an example for a clear understanding.

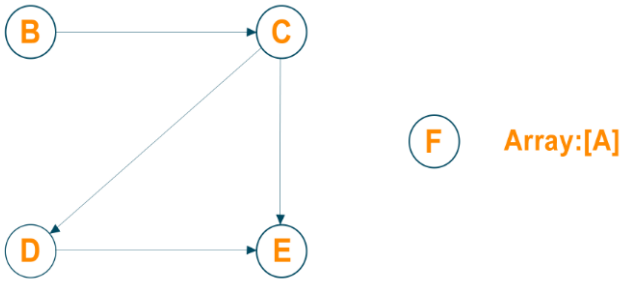
Example

Consider the following directed acyclic graph with their in-degree mentioned.

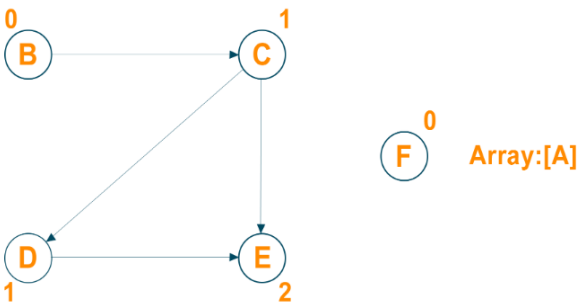


Identifying vertices that have no incoming edge. Here, nodes A and F have no incoming edges.

We will choose node A as the source node and deletes this node with all its outgoing edges and put it in the result array.

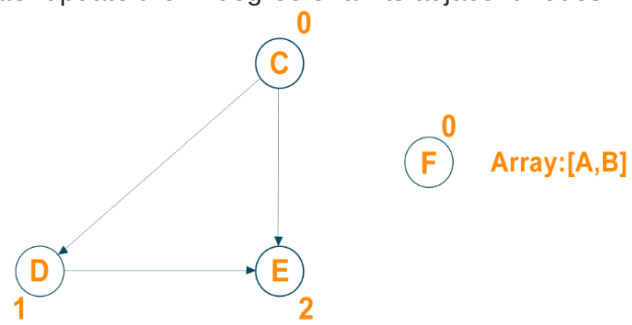


Now, update the in-degree of the adjacent nodes of the source node after deleting the outgoing edges of node A



Now again delete the node with in-degree 0 and its outgoing edges and insert it in the result array.

Later update the in-degree of all its adjacent nodes.



Now repeat the above steps to get output as below:

Array: [A,B,C,D,E,F]

In the second step, if we have chosen the source node as F then the topological sort of the graph will be F, A, B, C, D, E. Therefore, there is more than one topological sort possible for every directed acyclic graph.

Algorithm

The algorithm of the topological sort goes like this:

1. Identify the node that has no in-degree(no incoming edges) and select that node as the source node of the graph
2. Delete the source node with zero in-degree and also delete all its outgoing edges from the graph. Insert the deleted vertex in the result array.
3. Update the in-degree of the adjacent nodes after deleting the outgoing edges
4. Repeat step 1 to step 3 until the graph is empty.

5.Expalin About Kruskal's Algorithm in detail.

Kruskal's algorithm is another widely used algorithm for finding a minimum spanning tree. It follows these steps:

Initialization:

Create a forest (a set of trees), where each vertex is a separate tree.

Process:

While there are still vertices left and the forest is not yet connected:

Choose the smallest edge that connects two disjoint trees.

If adding this edge doesn't create a cycle, add it to the forest.

Termination:

When the forest becomes a single tree (i.e., all vertices are connected), you have a minimum spanning tree.

Kruskal's Algorithm is a popular and widely used greedy algorithm for finding the Minimum Spanning Tree (MST) in a connected, weighted, and undirected graph. It works by iteratively selecting edges with the smallest weights while ensuring that no cycles are formed in the process. Here's a detailed explanation of Kruskal's Algorithm:

Algorithm Steps:

Initialization:

Create a forest (a collection of trees), where each vertex in the graph is initially a separate

tree.

Sort all the edges of the graph in non-decreasing order of their weights.

Edge Selection:

Iterate through the sorted edges from the smallest to the largest weight.

For each edge, check if adding it to the MST would create a cycle. You can use a data structure like a disjoint-set (also known as a union-find data structure) to efficiently check for cycles.

Adding Edges to MST:

If adding the current edge does not create a cycle (i.e., the vertices connected by the edge belong to different trees in the forest), add it to the MST.

Combine the two trees into one by merging their sets of vertices.

Termination:

Continue this process until you have added (V - 1) edges to the MST, where V is the number of vertices in the graph. At this point, the MST is complete.