

Government of Karnataka

Department of Collegiate and Technical Education

GOVERNMENT POLYTECHNIC KOPPAL (137)

Department of Computer Science & Engg.

DATA STRUCTURES WITH PYTHON

(20CS41P) -IV Semester

Study Material

Prepared by

RAGHAVENDRA SETTY B

Lecturer, CSE

Government Polytechnic Koppal

Week 1

Introduction to Data Structures, operations, classification, Characteristics. Primitive types – primitive data structures, python examples. Non primitive types - Non primitive data structures, python examples. Linear and nonlinear data structures – with python examples. Introduction, Abstractions, Abstract Data Types, An Example of Abstract Data Type (Student, Date, Employee), Defining the ADT, Using the ADT, Implementing the ADT

Introduction to Data Structures:

- The data structure deals with the study of how the data is organized in the memory, how the data can be retrieved & how the data are manipulated.
- The way of representing data internally in the memory is called data structure.
- A **data structure** is a particular way of organizing data in a computer so that it can be used effectively.

Operations:

There are different types of operations that can be performed for the manipulation of data in every data structure.

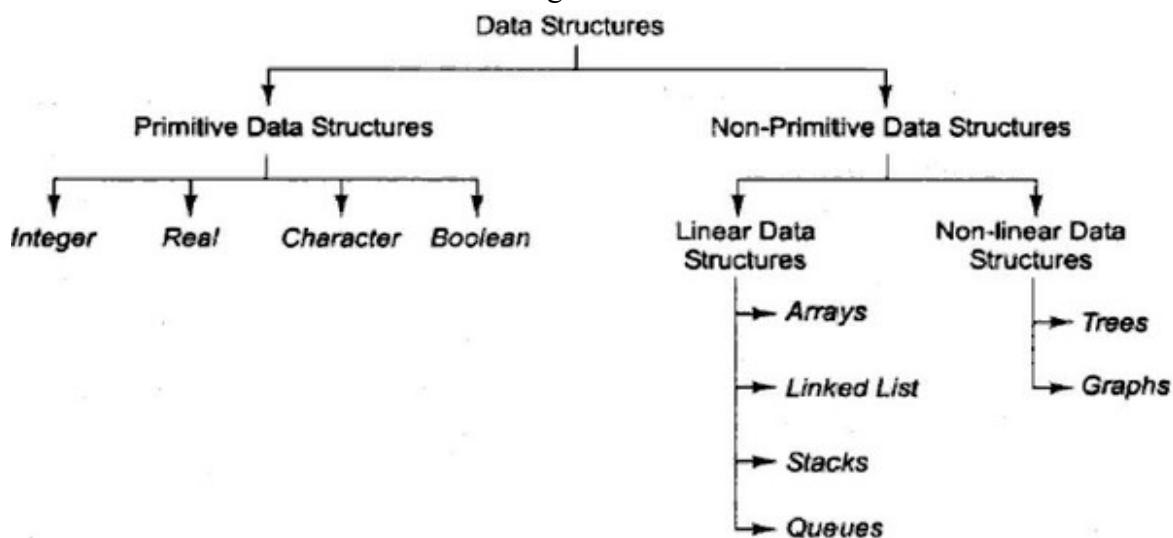
- **Traversing:** Traversing a Data Structure means to visit the element stored in it. This can be done with any type of DS.
- **Searching:** Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found
- **Insertion:** Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure.
- **Deletion:** Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure.

Characteristics:

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Classification:

- The classification of data structures is given below.



Primitive Data Structures:-

- The primitive data structures are the kind of data structures that are derived from the primitive data types.
- These are directly manipulated by the machine instructions
- Ex: - integers, floating point numbers, String, Boolean.
- ✓ **Integers** – This data type is used to represent numerical data, that is, positive or negative whole numbers without a decimal point. Ex: **-1, 3, or 6**.
- ✓ **Float** – Float signifies '**floating-point real number**'. It is used to represent rational numbers, usually containing a decimal point like **2.0 or 5.77**.
- ✓ **String** – This data type denotes a collection of alphabets, words or alphanumeric characters. It is created by including a series of characters within a pair of double or single quotes. **Ex: 'blue', 'red', etc.,**
- ✓ **Boolean** – This data type is useful in comparison and conditional expressions and can take up the values **TRUE or FALSE**.

Non Primitive data structures:-

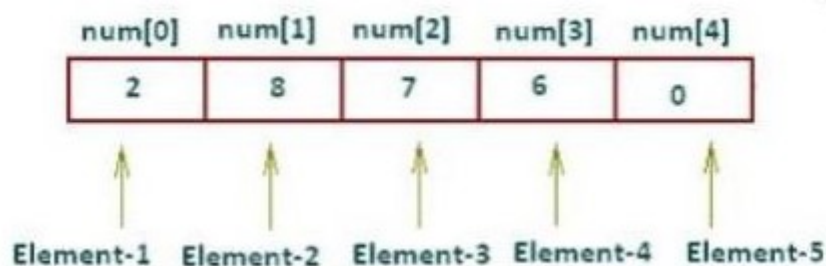
- ✓ The non primitive data structures are the kind of data structures that are derived from the primitive data structures.
- ✓ These cannot be directly manipulated by the machine instructions
- ✓ Ex:- Arrays, Stack, Queues, Linked List, Graphs, Trees, etc
- ✓ Non Primitive data structures are further classified as
 - Linear Data Structures
 - Non Linear Data Structures

Linear Data Structures:

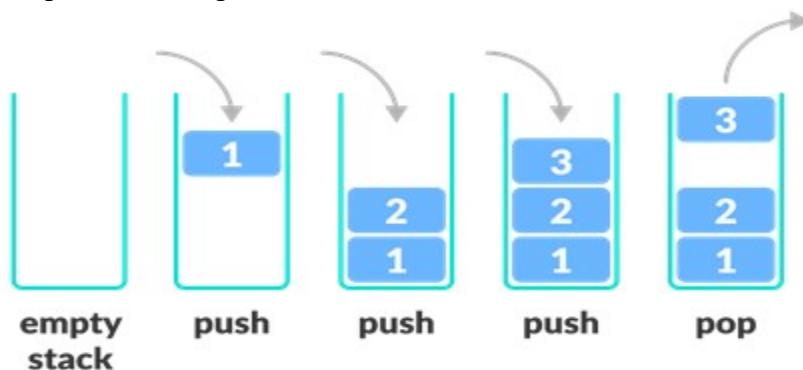
- In linear data structure, data elements are organized **sequentially** and therefore it is easy to implement in the computer memory.

These are the data structures which store the data elements in a sequential manner.

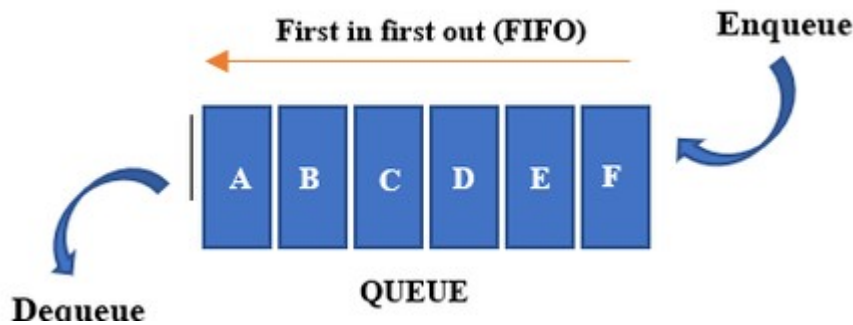
- ✓ **Array** – Array is a collection of elements with similar data type. It holds all elements in a sequential order.



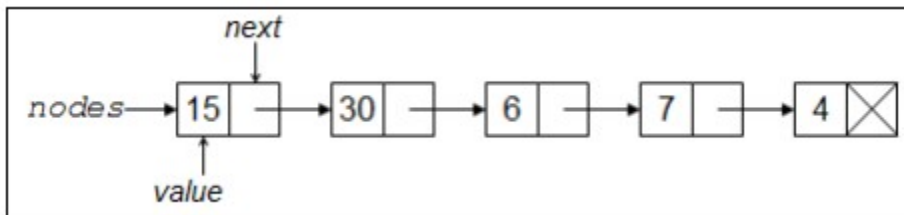
- ✓ **Stack** – Stack is a linear data structure which follows a particular order **LIFO**(Last In First Out) in which the operations are performed.



- ✓ **Queue** – Queue is also a linear data structure, in which the element is inserted from one end called **REAR**, and the deletion of existing element takes place from the other end called as **FRONT** and the order of queue is **FIFO** (First In First Out)



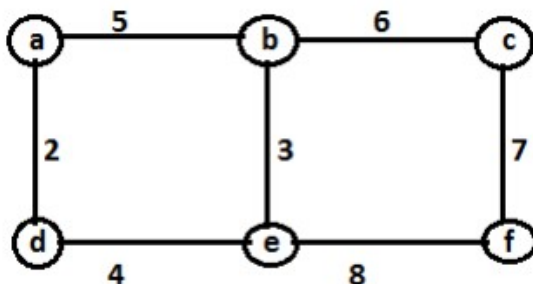
- ✓ **Linked List** – Linked List is a linear data structure and each data element contains a link to another element along with the data present in it.



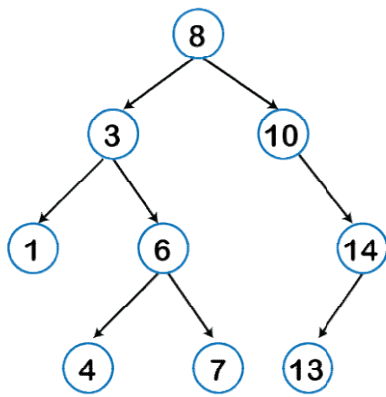
Non Linear Data Structures:

- In non-linear data structure, data elements are connected to several other data elements so that given data element has reach to one or more data elements.
- In non-linear data structure there is no any sequences.
- Every data item is attached to several other data items so that relationship is established between the items.
- Ex: Graph, Tree
- **Graph:**
 - ✓ Graph is collection of vertices and edges.
 - ✓ Graph is used to represents networks; the network may include paths and nodes
 - ✓ It is represented as $G=\{V,E\}$

Where V-> vertices and E -> Edges



- **Tree:**
 - ✓ Tree is a connected acyclic graph.
 - ✓ Tree is a non-linear structure.
 - ✓ Tree is a collection of nodes linked together to simulate hierarchy



Difference between Linear and Non Linear Data Structures:

Linear Data Structures	Non Linear Structures
Data is arranged in linear form.	Data is arranged in non linear form.
Every item is related to its previous and next item.	Every item is attached with many other items
Implementation is easy	Implementation is difficult
Data items can be traversed in a single run.	Data items cannot be traversed in a single run.
Ex:- array, linked list, stack, queue	Ex:- tree, graph

Abstract Data type (ADT):

- Abstract Data type (ADT) is a special kind of data type (or class) whose behavior is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “**abstract**” because it gives an implementation-independent view.
- The process of providing only the essentials and hiding the details is known as **abstraction**.

Stack Abstract Data Type:

- A stack is structured as an ordered collection of items where items are added to and removed from the end called the “top”. Stacks are ordered LIFO.
- **Stack():** creates a new stack that is empty. It needs no parameters and returns an empty stack.
- **push(item):** adds a new item to the top of the stack. It needs the item and returns nothing.
- **pop():** removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- **peek():** returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- **isEmpty():** tests to see whether the stack is empty. It needs no parameters and returns a boolean value.

- **size()**: returns the number of items on the stack. It needs no parameters and returns an integer.

STACK ADT**class stack:**

```
def __init__(self):
    self.items = []

def isEmpty(self):
    return self.items == []

def push(self, item):
    self.items.append(item)

def pop(self):
    return self.items.pop()

def peek(self):
    return self.items[len(self.items) - 1]

def size(self):
    return len(self.items)

def display(self):
    return (self.items)
```

```
s=stack()
print(s.isEmpty())
print("push operations")
s.push(11)
s.push(12)
s.push(13)
print("size:",s.size())
print(s.display())
print("peek",s.peak())
print("pop operations")
print(s.pop())
print(s.pop())
print(s.display())
print("size:",s.size())
```

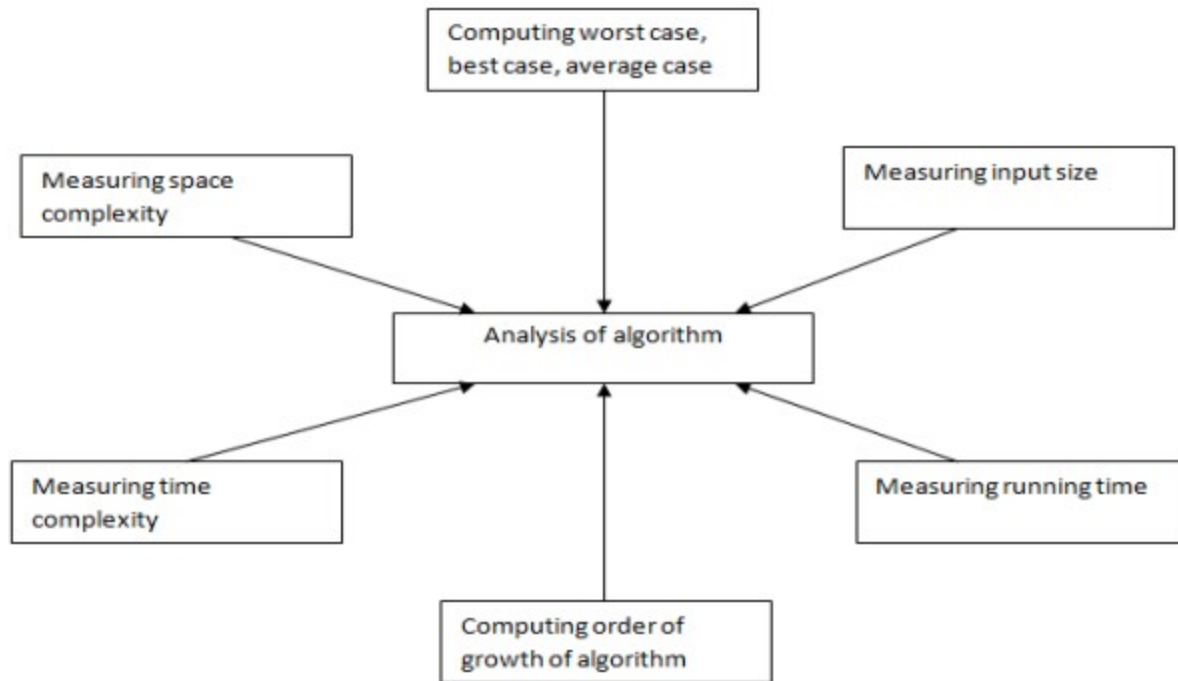
Date Abstract Data Type:**class date:****def __init__(self,a,b,c):****self.d=a****self.m=b****self.y=c****def day(self):****print("Day = ", self.d)****def month(self):****print("Month = ", self.m)****def year(self):****print("year = ", self.y)****def monthName(self):****months = ["Unknown","January","Febuary","March","April","May","June","July",
"August","September","October","November","December"]****print("Month Name:",months[self.m])****def isLeapYear(self):****if (self.y % 400 == 0) and (self.y % 100 == 0):****print("It is a Leap year")****elif (self.y % 4 == 0) and (self.y % 100 != 0):****print("It is a Leap year")****else:****print("It is not a Leap year")****d1 = date(3,8,2000)****d1.day()****d1.month()****d1.year()****d1.monthName()****d1.isLeapYear()**

WEEK-2

Algorithm Analysis – Space Complexity, Time Complexity. Run time analysis. Asymptomatic notations, Big-O Notation, Omega Notation, Theta Notation.

Algorithm Analysis:

The efficiency of algorithm can be decided by measuring the performance of an algorithm.



The performance of the algorithm can be computed based on the following factors

1. Space efficiency OR Space Complexity
2. Time efficiency OR Time Complexity

Space complexity:

- Space complexity can be defined as amount of memory required to run the algorithm.
- The following components are important in calculating the space.
 1. Instruction space: - Instruction Space is used to store the machine code generated by the compiler.
 2. Data Space: - Data Space is used to store constants, static variables, intermediate variables and variables
 3. Stack space:- Stack Space is used to store the return address and return values by the function.
- The space complexity (Sp) can be given as

$$\text{Total space} = \text{Sp} + \text{Cp}$$

Where

Sp -> Space required for the activation record (Instances and variables) that is dynamic part

Cp -> Space required for the constants that is static part

Ex:

$$a = 10$$

$$b = 20$$

$$c = 30$$

$$\text{avg} = (a + b + c) / 3$$

print(avg)

- ✓ Here, a,b,c and avg are the integer variables so that Space is
- ✓ $Sp=4*2=8$ Bytes \Rightarrow 4 is the number of variables and 2 is the size
- ✓ $Cp=1*2=2$ Bytes \Rightarrow 1 is the number of constants and 2 is the size
- ✓ Total Space $=8+2=10$ Bytes

Time complexity:

- Time complexity of an algorithm is the computer time required by the algorithm or program to run till its completion.
- Time complexity of an algorithm is depends on the
 - ✓ System configuration
 - ✓ Input Data
 - ✓ Number of other programs running
- Time complexity is calculated in terms of frequency count.
- Frequency count is a count denoting number of times required for execution of statements.

Ex1:

```
a = 10
b = 20
c = a + b
```

Here 1 unit required for $a=10$, 1 unit is required for $b=20$ and 1 unit required for $c=a+b$ so total 3 units time is required.

Worst-case, Best-case and Average-case Efficiencies

Worst-case:- Worst case efficiency of algorithm for input size N is its takes longest time to run the algorithm for all possible input values.

Best-case:- Best-case efficiency of algorithm for input size N is its takes fastest time to run the algorithm for all possible input values.

Average-case:- Average-case efficiency of algorithm for input size N is its takes Normal time to run algorithm for all possible input values.

Ex:

```
ALGORITHM Sequential_Search(A[0,1.....n-1],key)
//Input : An Array A[0,1...n-1] and search value is key
//Output : Returns index matched item with key
           otherwise returns -1.
{
    for i=0 to n-1 do
        if(A[i] == key) Then
            returns i;
    return -1;
}
```

1. **Best-Case:** In Sequential Search, the element key is searched from the array of N elements. If the key element is present at **first** location in array then algorithm runs for a very short time so the Best-Case efficiency is

$$C_{\text{Best-case}}(n) = 1.$$

2. **Worst-Case:** The key element is searched from the array of N elements. If the key element presents at nth location of array, then algorithm will run for longest time.

$$C_{\text{worst-Case}} = n$$

3. **Average-Case:**

Let $C[j]$ = Number of comparisons up to jth position in the array A.

Then total number of comparisons $C(n) = C[1] + C[2] + \dots + C[n]$

Average-case efficiency is

$$C_{\text{Average-case}}(n) = \frac{n+1}{2}$$

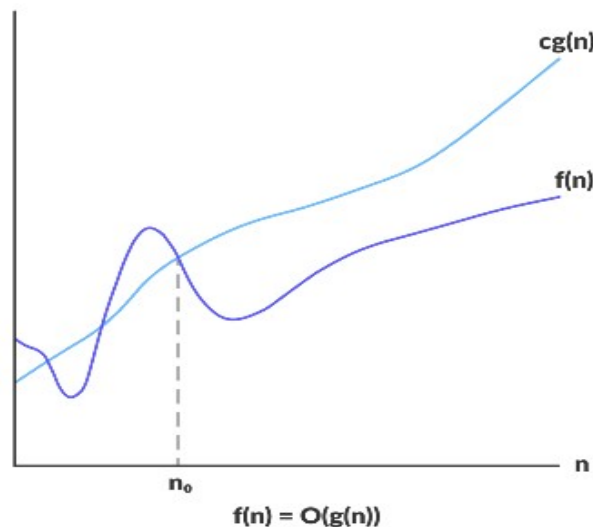
Asymptotic Notations:

- Asymptotic notations are mathematical tools to represents time complexity of the algorithm.
- The efficiency of the algorithm can be measured by computing time-complexity. This time complexity can represent by using asymptotic notations,
- There are 3 asymptotic notations.

1. Big oh notation (O)
2. Omega Notation (Ω)
3. Theta Notation (Θ)

1. Big oh notation

- It represents the upper bound of algorithm running time.
- This notation measures the worst case complexity of the algorithm.
- It indicates the longest amount of time to complete its operations.



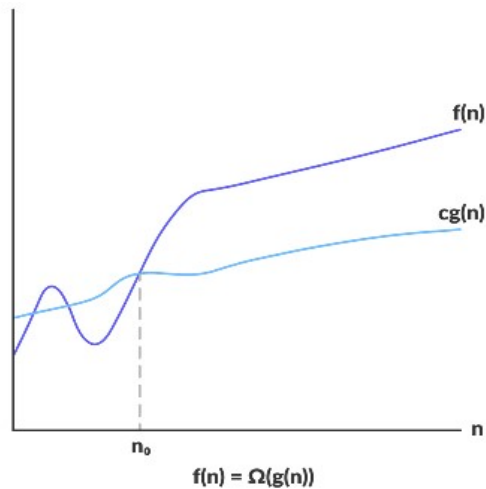
- For a function $g(n)$, $O(g(n))$ is given by the relation:

$$O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

- The above expression can be described as a function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies between 0 and $cg(n)$, for sufficiently large n .

Omega Notation:

- It represents the lower bound of the algorithm running time.
- It measures the best case time complexity of the algorithm.
- It indicates the shortest amount of time taken by the algorithm to complete its operation.



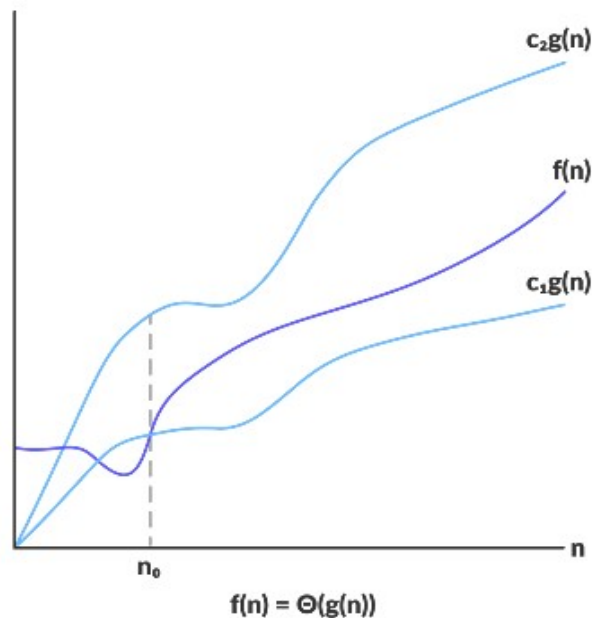
- For a function $g(n)$, $\Omega(g(n))$ is given by the relation:

$$\Omega(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$
- The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$, for sufficiently large n .

Theta Notation:

- Its represents the both lower bound and upper bound of algorithm running time.
- Its measure the average case time complexity of the algorithm.
- For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

$$\Theta(g(n)) = \{ f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$$
- The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n .



Basic asymptotic efficiency classes:

Name of the efficiency class	Order of growth	Description	Example
Constant	1	As input size grows then we get larger running time	Scanning array elements
Logarithmic	$\log n$	When we get logarithmic running time then it is sure that the algorithm doesn't consider all its input rather the problem is divided into smaller parts on each iteration.	Performing binary search operation.
Linear	n	The running time of algorithm depends on the input size n .	Performing sequential search operation.
$n \log n$	$n \log n$	Some instance of input is considered for the list of size n	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	$n!$	When an algorithm is computing all the permutations then this efficiency occurs.	Generating all permutations.

Week-3

Algorithm design strategies: Brute force – Bubble sort, Selection Sort, Linear Search. Decrease and conquer - Insertion Sort

Algorithm design strategies:**Brute Force:**

- Brute force is a straight-forward approach for solving the problem. It is also called as “Just do it” approach.
- It is a simplest way to explore the space of solutions. This will go through all possible solutions extensively.
- Brute force method solves the problem with acceptable speed and large class of input.
- Examples (Applications)
 - ✓ Bubble Sort
 - ✓ Selection Sort
 - ✓ Computing $n!$: The $n!$ Can be computed as $1 \times 2 \times 3 \times \dots \times n$
 - ✓ Multiplication of two matrices

Bubble Sort:

- Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- It works as follows
 - ✓ Compare each pair of adjacent elements from the beginning of an array.
 - ✓ Swap those elements if the elements are in reverse order
 - ✓ Repeat the steps 1 and 2 until the array is sorted
- The algorithm for bubble sort is given below

```
Algorithm Bubble_Sort(A[0,1,...n-1],n)  
//Input : An Array elements A[0,1,...n-1]  
//Output : Sorted Array  
{  
    for i ← 0 to n-2 do  
        for j ← 0 to n-2-i do  
            if(A[j] > A[j+1])  
            {  
                temp ← A[j]  
                A[j] ← A[j+1]  
                A[j+1] ← temp  
            }  
        }  
    }  
}
```

- **How Bubble Sort works :** Sort the following elements using Bubble Sort 7,5,9,2,3,1
- **Pass:1**

7 5 9 2 3 1 ↑ Swap A[j] and A[j+1]	Iteration 1
5 7 9 2 3 1 ↑ No Swap	2
5 7 9 2 3 1 ↑ Swap	3
5 7 2 9 3 1 Swap ↑	4
5 7 2 3 9 1 Swap ↑	5 (n-1) Iterations
5 7 2 3 1 9	After 1 st Pass

➤ Pass:2

5 7 2 3 1 9 ↑ No Swap	Iteration 1
5 7 2 3 1 9 ↑ Swap	2
5 2 7 3 1 9 ↑ Swap	3
5 2 3 7 1 9 Swap ↑	4 (n-2) Iterations
5 2 3 1 7 9	After 2 nd Pass

➤ Pass:3

5 2 3 1 7 9 ↑ Swap	Iteration 1
2 5 3 1 7 9 ↑ Swap	2
2 3 5 1 7 9 ↑ Swap	3 (n-3) Iterations
2 3 1 5 7 9	After 3 rd Pass

➤ Pass:4

2 3 1 5 7 9 ↑ No Swap	Iteration 1
2 3 1 5 7 9 ↑ Swap	2 (n-4) Iterations
2 1 3 5 7 9	After 4 th Pass

➤ Pass:5

2 1 3 5 7 9 ↑ Swap	Iteration 1 (n-5) Iterations
1 2 3 5 7 9	After 5 th Pass

➤ **Time Complexities:**

- ✓ The time complexity of the bubble sort is $O(n^2)$
- ✓ **Worst Case Complexity:** $O(n^2)$: If we want to sort in ascending order and the array is in descending order then the worst case occurs.
- ✓ **Best Case Complexity:** $O(n)$: If the array is already sorted, then there is no need for sorting.
- ✓ **Average Case Complexity:** $O(n^2)$: It occurs when the elements of the array are in random order.

➤ **Space Complexity:** Space complexity is $O(1)$ because an extra variable (temp) is used for swapping.

➤ The python code implementation of bubble sort is given below

```
def bubblesort(a):
```

```
    n = len(a)
```

```
    for i in range(n-2):
```

```
        for j in range(n-2-i):
```

```
            if a[j]>a[j+1]:
```

```
                temp = a[j]
```

```
                a[j] = a[j+1]
```

```
                a[j+1] = temp
```

```
x = [34,46,43,27,57,41,45,21,70]
```

```
print("Before sorting:",x)
```

```
bubblesort(x)
```

```
print("After sorting:",x)
```

Selection Sort:

- Selection sort algorithm sorts array elements by repeatedly finding the smallest element from the unsorted array and putting at the beginning. This process will continue until the entire array is sorted.
- Selection sort means selecting the smallest element.
- Algorithm for selection sort is given below

```
Algorithm Selection_Sort(A[0,1,...n-1],n)
```

```
//Input : An Array elements A[0,1,...n-1]
```

```
//Output : Sorted Array
```

```
{
    for i ← 0 to n-2 do
    {
        min ← i
        for j ← i+1 to n-1 do
        {
            if(A[j] < A[min])
                min ← j
        }
        temp ← A[i]
        A[i] ← A[min]
        A[min] ← temp
    }
}
```


- **How Selection Sort works:** Sort the following elements using Selection Sort **70,30,40,20,50,60,10,65**

Original Array	After 1 st pass	After 2 nd Pass	After 3 rd Pass	After 4 th Pass	After 5 th Pass
70	10	10	10	10	10
30	30	20	20	20	20
40	40	40	30	30	30
20	20	30	40	40	40
60	60	60	60	50	50
50	50	50	50	60	60
10	70	70	70	70	65
65	65	65	65	65	70

- **Time Complexities:**

- ✓ The time complexity of the selection sort is $O(n^2)$
- ✓ **Worst Case Complexity: $O(n^2)$:** If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
- ✓ **Best Case Complexity: $O(n^2)$:** It occurs when the array is already sorted
- ✓ **Average Case Complexity: $O(n^2)$:** It occurs when the elements of the array are in random order.

- The time complexity of the selection sort is the same in all cases.

- **Space Complexity:** Space complexity is $O(1)$ because an extra variable temp is used for swapping.

- The python code implementation of selection sort is given below

```
def selectionsort(a):
    n = len(a)
    for i in range(n-2):
        min = i
        for j in range(i+1,n-1):
            if a[j]<a[min]:
                min=j
        temp = a[i]
        a[i] = a[min]
        a[min] = temp
x = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",x)
selectionsort(x)
print("After sorting:",x)
```

Sequential Search or Linear Search:

- Sequential search is the most natural searching method and it is simplest method.
- In this method, the searching process starts from the beginning of an array and continues until the given element is found or until the end of the array.

- The algorithm for linear search is given below

```

Algorithm Sequential_Search(A[0,1,...n-1],n,key)
//Input : An Array elements A[0,1,...n-1] and search key
//Output : Returns of the index of element if found otherwise
           returns -1
{
    for i ← 0 to n-1 do
    {
        if(A[i]=key)
            return i
    }
    return -1
}

```

- The **time complexity** of linear search is $O(n)$ and **space complexity** is $O(1)$
- The python code implementation of linear search is given below

```

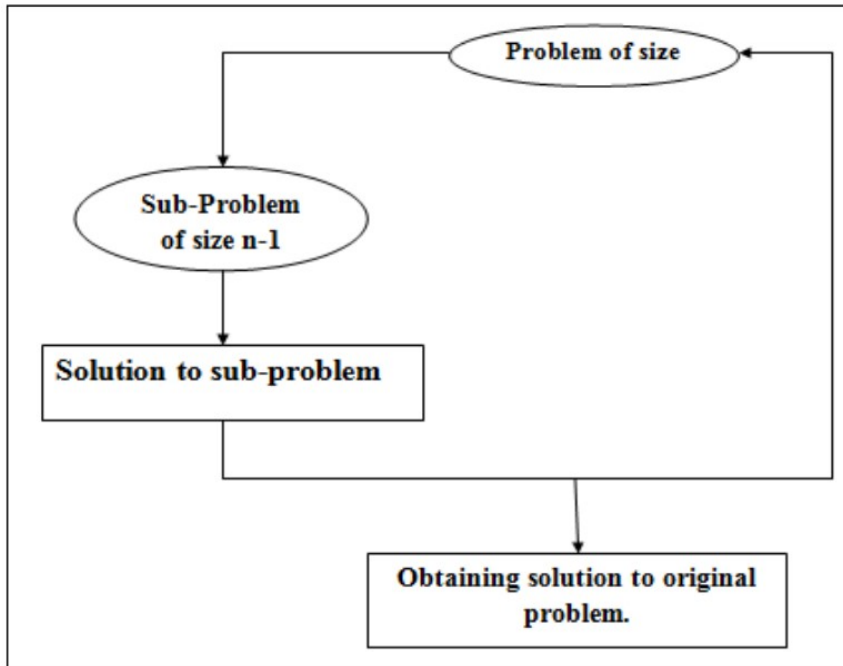
def linearsearch(a, key):
    n = len(a)
    for i in range(n):
        if a[i] == key:
            return i;
    return -1
a = [13,24,35,46,57,68,79]
print("the array elements are:",a)
k = int(input("enter the key element to search:"))
i = linearsearch(a,k)
if i == -1:
    print("Search UnSuccessful")
else:
    print("Search Successful key found at location:",i+1)

```

Decrease-and-Conquer:

- The decrease and conquer is a method of solving a problem by changing the problem size from n to smaller size of $n-1$, $n/2$ etc.
- In other words, change the problem from larger instance into smaller instance.
- Conquer (or solve) the problem of smaller size.
- Convert the solution of smaller size problem into a solution for larger size problem.
- Using decrease and conquer technique, we can solve a given problem using top-down technique (using recursion) or bottom up technique (using iterative procedure).
- There are three major variations of decrease-and-conquer:
 - ✓ decrease by a constant
 - ✓ decrease by a constant factor

- ✓ variable size decrease



Insertion Sort:

- It is a simple Sorting algorithm which sorts the array by shifting elements one by one.
- Insertion sort works as follows
 1. Set the marker for the sorted section after the first element in an array.
 2. Select the first unsorted element
 3. Compare the sorted section element with unsorted section element and swap those elements if the condition is true
 4. Move the marker to the right position of sorted section.
 5. Repeat the steps 3 and 4 until the unsorted section is empty
- The algorithm of insertion sort is given below

Algorithm Insertion_sort (A[0...n-1], n)

//Input : Array A[0..n-1] with n elements

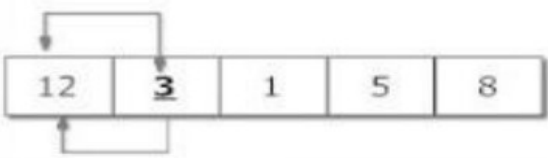
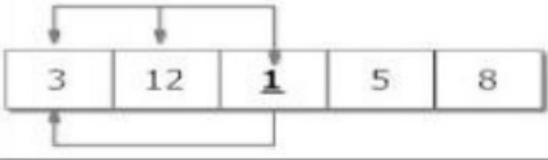
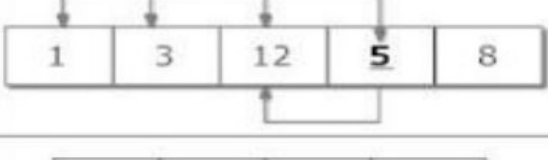
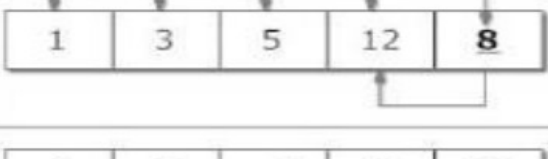

//Output : Sorted Array

```

{
  for i ← 1 to n-1 do
  {
    k ← A[i]
    j ← i-1
    while j ≥ 0 AND A[j] > k do
    {
      A[j+1] ← A[j]
      j ← j-1
    }
    A[j+1] ← k
  }
}

```

- The working of insertion sort is given below

Step 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2		Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

➤ **Time Complexities:**

- The time complexity of the insertion sort is $O(n^2)$

✓ **Worst Case Complexity:** $O(n^2)$: Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

✓ **Best Case Complexity:** $O(n)$: When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n numbers of comparisons. Thus, complexity is linear.

✓ **Average Case Complexity:** $O(n^2)$: It occurs when the elements of an array are in random order.

➤ **Space Complexity:** Space complexity is $O(1)$ because an extra variable k is used.

- The python code implementation of insertion sort is given below

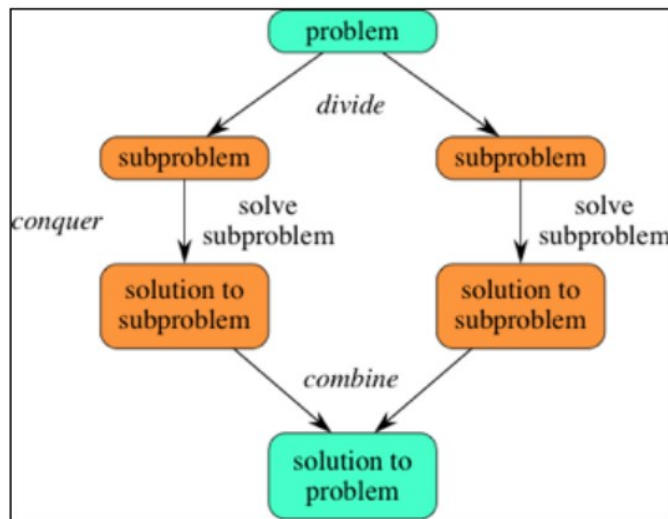
```
def insertionsort(a):
    n = len(a)
    for i in range(1,n-1):
        k = a[i]
        j = i-1
        while j>=0 and a[j]>k:
            a[j+1] = a[j]
            j=j-1
        a[j+1] = k
x = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",x)
insertionsort(x)
print("After sorting:",x)
```

Week-4

Divide and conquer - Merge Sort, Quick Sort, Binary search. Dynamic programming - Fibonacci sequence
 Backtracking – Concepts only (Implementation examples with recursion in week 9). Greedy – Concepts only.

Divide and conquer:

- Divide and conquer is a method to solve the given problem. It solves the problem using 3 steps,
 - ✓ Divide – Break the given problem into sub-problems of same type.
 - ✓ Conquer – Recursively solve these problems.
 - ✓ Combine – Appropriately combine the each sub-solution and make it as single solution.

**Merge Sort:**

- Merge sort is a sorting algorithm, that uses divide and conquer is a method to sort the sequence of values..
- The sequence of values is recursively divided into smaller and smaller subsequences until each value is contained within its own subsequences. The subsequences are then merged back together to create a sorted sequence.
- It works in 3 steps:
 - ✓ Divide: Partition the array into 2 sub array.
 - ✓ Conquer: Recursively sort the sub array.
 - ✓ Combine: Merge the sub array into a unique sorted array / group
- The algorithm for merge sort is given below

Algorithm mergesort (A, l, h)

```

{
  if (l < h) then
  {
    m ← (l+h) / 2 ;
    mergesort (A,l,h) ; //First array
    mergesort (A,m+1,h) ; //Second array
    combine (A,l,m,h) ; // merge the sub array into single array.
  }
}
  
```

Algorithm combine (A,l,m,h)

```

{
  k ← l; // Index of the temp array.
  i ← l; // Index of the left sub - array.
  j ← m+1; // Index of the right sub - array.
  while (i<=m and j<=h) do
  {
    if(A[i]<=A[j]) then //copy the smallest element present in left sub array into temp arra
    {
      temp[k] ← A[i] ;
      i ← i+1 ;
      k ← k+1 ;
    }
    else //copy the smallest element present in right sub array into temp array
    {
      temp[k] ← A[j] ;
      j ← j+1 ;
      k ← k+1 ;
    }
  }
  while (i<=m) do //copy the remaining elements present in left sub array into temp array
  {
    temp [k] ← A[i];
    i ← i+1;
    k ← k+1;
  }
  while (j<=h) do //copy the remaining elements present in right sub array into temp array
  {
    temp[k] ← A[j] ;
    j ← j+1 ;
    k ← k+1 ;
  }
}

```

- The python code implementation of merge sort is given below

```

def mergesort(a, l, h):
    if l < h:
        m = l+(h-l)//2
        mergesort(a, l, m)
        mergesort(a, m+1, h)
        combine(a, l, m, h)
def combine(a, l, m, h):
    n1 = m - l + 1
    n2 = h - m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0, n1):
        L[i] = a[l + i]
    for j in range(0, n2):
        R[j] = a[m + 1 + j]

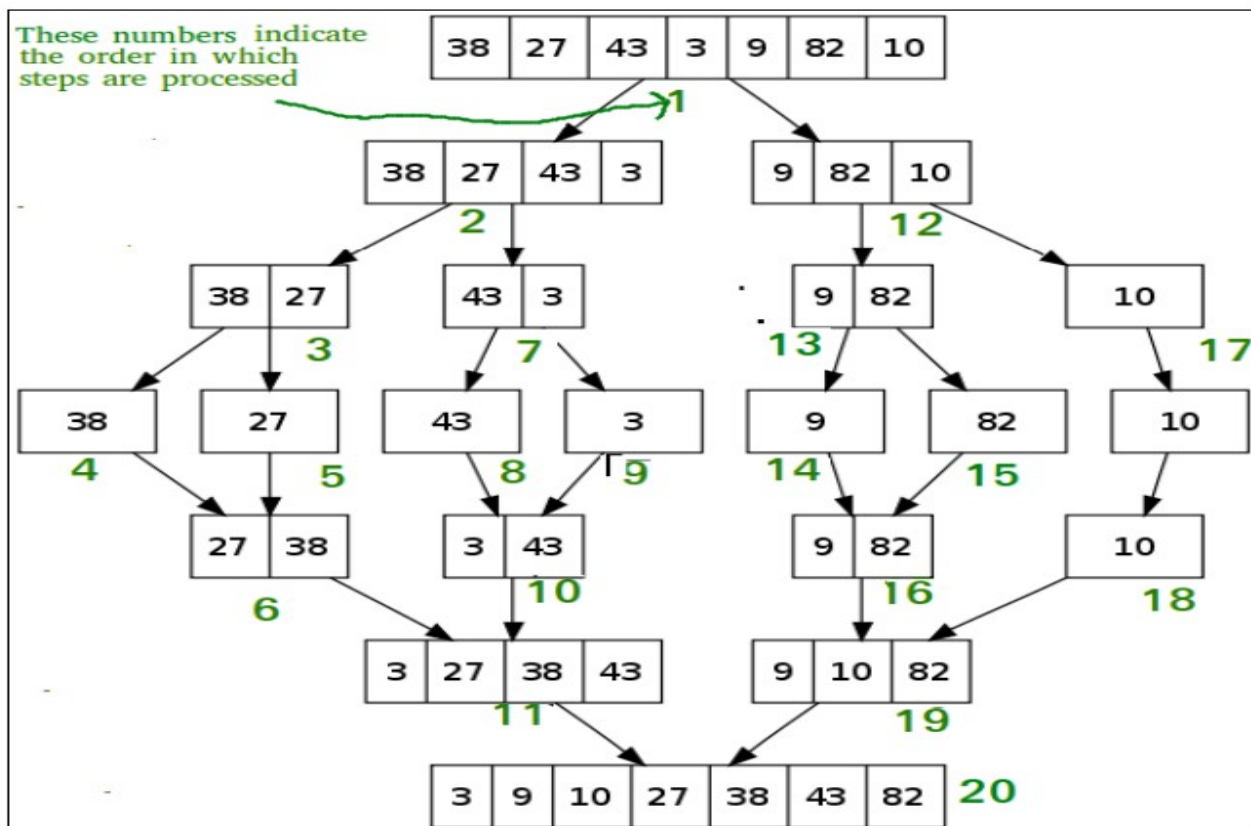
```

```

i = 0
j = 0
k = 1
while i < n1 and j < n2:
    if L[i] <= R[j]:
        a[k] = L[i]
        i += 1
    else:
        a[k] = R[j]
        j += 1
    k += 1
while i < n1:
    a[k] = L[i]
    i += 1
    k += 1
while j < n2:
    a[k] = R[j]
    j += 1
    k += 1
x = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",x)
mergesort(x,0,len(x)-1)
print("After sorting:",x)

```

➤ The working of merge sort is illustrated below



➤ **Time Complexity:**

- ✓ Best Case Complexity: $O(n \log n)$
- ✓ Worst Case Complexity: $O(n \log n)$
- ✓ Average Case Complexity: $O(n \log n)$

➤ **Space Complexity:** The space complexity of merge sort is $O(n)$.

Quick Sort:

- Quick sort is a sorting method that uses the divide and conquer strategy. In this method division is dynamically carried out.
- It works in 3 steps
 - ✓ **Divide:** Split the array into two sub arrays that each element in the left sub array is less than or equal to the middle element and each element in the right sub array is greater than the middle element.
 - ✓ **Conquer:** Recursively sort the two sub arrays
 - ✓ **Combine:** Combine all the sub arrays into single array
- The difference between the merge sort and quick sort is the merge sort splits the sequence of keys at the midpoint, the quick sort partitions the sequence by dividing it into two segments based on a selected pivot key.
- The algorithm for quick sort is given below

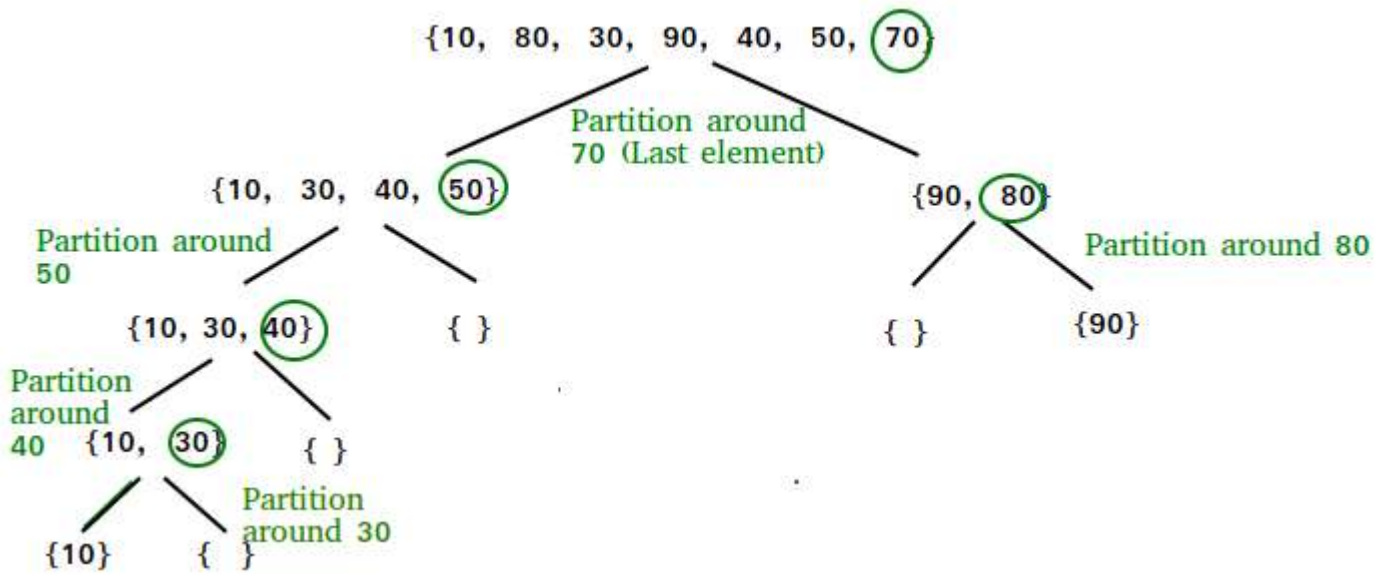
Algorithm Quick($A[0,1,...n-1]$,low,mid)

```
{
    if(low<high) then //Split the array into two sub arrays
        mid ← Partition( $A[low,...high]$ ) // mid is the middle element of the array
        Quick( $A[low ...mid-1]$ ) // Sort the left sub array
        Quick( $A[mid+1...high]$ ) //Sort the right sub array
}
```

Algorithm Partition($A[low ..high]$)

```
{
    P ←  $A[low]$  //Take the first element as pivot
    i ← low
    j ← high+1
    while(i ≤ j) do
    {
        while( $A[i] ≤ p$ ) do //Check the first element is less than p
            i ← i+1 //If yes, increment i
        while( $A[j] ≥ p$ ) do //Check the last element is greater than p
            j ← j-1 // If yes, decrement j
        if(i < j) // If i less than and equal to j
            swap( $A[i], A[j]$ ) //Exchange  $A[i]$  and  $A[j]$ 
    }
    swap( $A[low], A[j]$ ) // When i crosses j, then swap  $A[low]$  and  $A[j]$ 
}
```

- The working of merge sort is illustrated using the below example



- The python code implementation of merge sort is given below

```
def partition(arr, low, high):
    i = (low-1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

def quicksort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quicksort(arr, low, pi-1)
        quicksort(arr, pi+1, high)

data = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",data)
quicksort(data,0,len(data)-1)
print("After sorting:",data)
```

- **Time Complexities:**

- ✓ **Worst Case Complexity** [Big-O]: $O(n^2)$:It occurs when the pivot element picked is either the greatest or the smallest element.
- ✓ **Best Case Complexity** [Big-omega]: $O(n \cdot \log n)$: It occurs when the pivot element is always the middle element or near to the middle element.

- ✓ **Average Case Complexity** [Big-theta]: $O(n \cdot \log n)$:It occurs when the above conditions do not occur.
- **Space Complexity**: The space complexity for quick sort is $O(\log n)$.

Binary Search:

- Binary search works only on sorted array.
- This technique searches the given element by repeatedly dividing the array into half.
- The idea of binary search is to reduce the time complexity
- The algorithm of Binary Search is given below

Algorithm Binary_Search (A[0,1,...n-1], key)

```

{
    low ← 0
    high ← n-1
    while( low <= high ) do
    {
        mid ← (high + low) // 2
        if (key == a[mid]) then
            return mid
        else if (key < a[mid]) then
            high ← mid - 1
        else
            low ← mid + 1
    }
    return -1
}

```

- The working of Binary Search is given below

Search Element : 25

5	10	15	20	25	30	35
---	----	----	----	----	----	----

Starts with middle element

5	10	15	20	25	30	35
---	----	----	----	----	----	----

25 > 20

5	10	15	20	25	30	35
---	----	----	----	----	----	----

5	10	15	20	25	30	35
---	----	----	----	----	----	----

25 < 30

5	10	15	20	25	30	35
---	----	----	----	----	----	----

Element Found

5	10	15	20	25	30	35
---	----	----	----	----	----	----

- The python code implementation of Binary Search is given below

```
def binarysearch(a, key):  
    low = 0  
    high = len(a) - 1  
    while low <= high:  
        mid = (high + low) // 2  
        if key == a[mid]:  
            return mid  
        elif key < a[mid]:  
            high = mid - 1  
        else:  
            low = mid + 1  
    return -1  
a = [13,24,35,46,57,68,79]  
print("the array elements are:",a)  
k = int(input("enter the key element to search:"))  
r = binarysearch(a,k)  
if r == -1:  
    print("Search UnSuccessful")  
else:  
    print("Search Successful key found at location:",r+1)
```

- **Time Complexity:**

- ✓ **Best Case Complexity:** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is $O(1)$.
- ✓ **Average Case Complexity:** The average case time complexity of Binary search is $O(\log n)$.
- ✓ **Worst Case Complexity:** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is $O(\log n)$.

- **Space Complexity:** The space complexity of binary search is $O(1)$.

Dynamic programming:

- Dynamic programming is a technique that breaks the problems into sub problems and stores the results for future purposes so that we do not need to compute the result once again.
- This technique of storing the values of sub-problems is called **Memorization**.
- It is used to solve optimization problems.
- Dynamic programming is a technique to solve the recursive problems in more efficient manner. Many times in recursion we solve the sub-problems repeatedly.
- Example: Fibonacci sequence

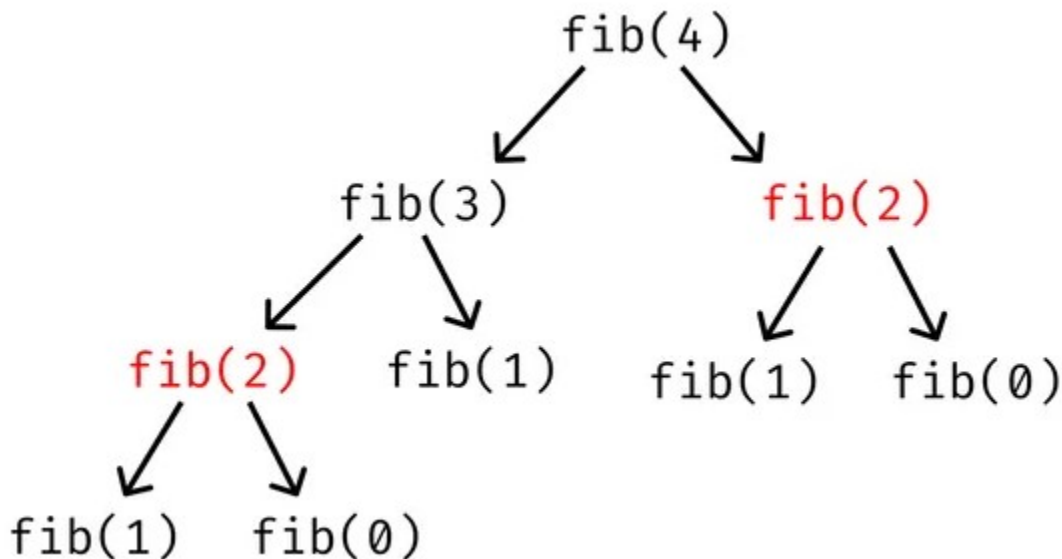
Fibonacci sequence: A Fibonacci series is the sequence of numbers in which each number is the sum of the 2 preceding numbers.

sequence: 0, 1, 1, 2, 3, 5, 8, 13
 term: 0 1 2 3 4 5 6 7

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

- The python code to implement Fibonacci sequence using recursion is given below

```
def fib(n):
    if n<=1:
        return n
    return fib(n-1) + fib(n-2)
n=int(input("Enter the term:"))
print("The Fibonacci value is:",fib(n))
```



- The larger Fibonacci sequence has overlapping function call. We can reduce the call with dynamic programming by storing the results of the sub problems.
- The python code to implement Fibonacci sequence using dynamic programming is given below

```
def fib(n):
    if n<=1:
        return n
    f = [0, 1]
    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    print("The Fibonacci sequence is:",f)
    return f[n]
n=int(input("Enter the term:"))
print("The Fibonacci value is:",fib(n))
```

Greedy Concepts:

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong.
- It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Week-5

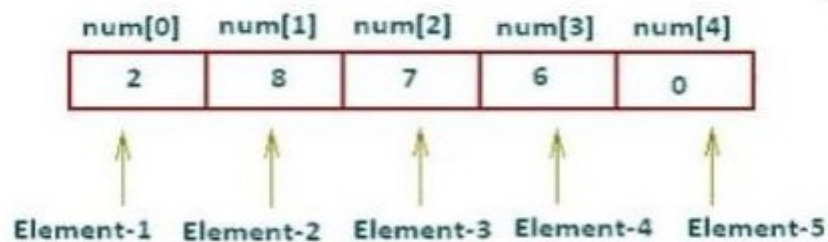
Linear (arrays) vs nonlinear (pointer) structures – Run time and space requirements, when to use what? Introduction to linked list, Examples: Image viewer, music player list etc. (to be used to explain concept of list), applications.

Week-6

The Singly Linked List- Creating Nodes, Traversing the Nodes, searching for a Node, Prepending Nodes, Removing Nodes. Linked List Iterators.

Linear Structures (Arrays):

- Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
- All arrays consist of contiguous memory locations.
- The lowest address corresponds to the first element and the highest address to the last element.

**Non-Linear structures (Pointers):**

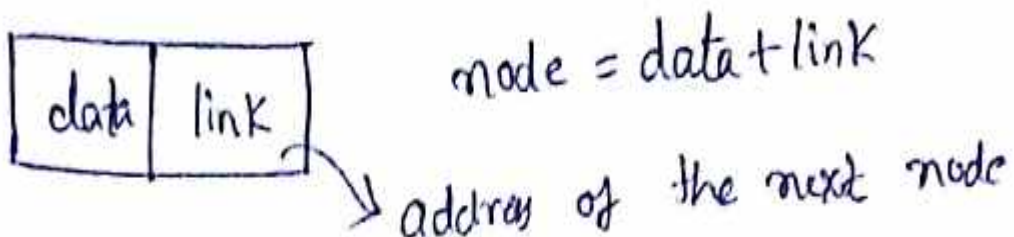
- A pointer is a variable which holds the address of another variable, i.e., address of the memory location.

Advantages of pointers:-

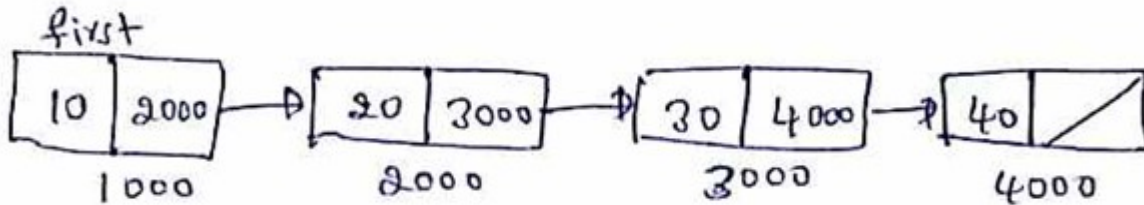
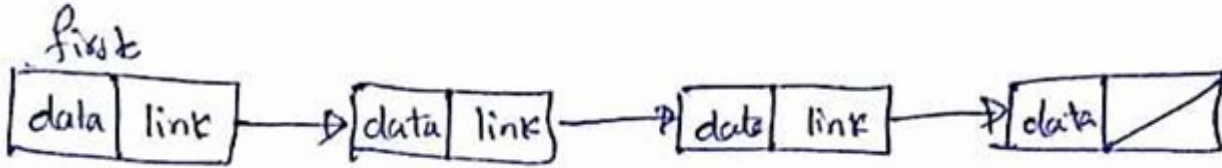
- Pointers increase the speed of execution because manipulation with address is faster than the variables.
- Pointers allow dynamic memory allocation and de allocation.
- Using pointers arrays, strings, structures can be handled in more efficient way.
- Using pointers it is possible to create more complex data structures like linked lists, trees etc

Introduction to Linked List:

- A linked list is a sequence of data elements, which are connected together via links.
- A Linked list is a collection of zero or more nodes where each node contains 2 fields, data and link.
- Data field stores the actual information and link (address) field contains the address of the next node.
- The memory representation of a node is given below



- The memory representation of a linked list is given below



- Each node contains two fields i.e. data and link field
- The data field contains data or information.
- Link field is a pointer which contains the address of the next node.
- Last node's link field will be NULL.
- They are less rigid; elements can be stored in non-contiguous locations.
- They require additional values to reference the next element.
- Every node in the linked list points to the next element in the linked list.
- Since they are non-contiguous, the size of the linked list can be altered at run-time.
- Memory is allocated to linked list at run time.
- Linked list requires more memory since it includes reference to next node.

Array V/s Linked List:

Arrays	Linked Lists
Size of an array is fixed	Size of a linked list is not fixed.
Memory is allocated from stack area.	Memory is allocated from heap area.
Memory is allocated during compile time.	Memory is allocated during run time.
It occupies less memory	It occupies more memory
Inserting a new elements is difficult	Inserting new elements is easy
Deleting an element from an array is difficult	Deleting an element is easy

Advantages of Linked List:-

- ✓ Linked lists are dynamic data structure they can grow or shrink during the execution of a program.
- ✓ Insertion and deletion are easier & efficient.
- ✓ Efficient memory utilization i.e. memory is allocated whenever it is required and de allocated whenever it is no longer needed.
- ✓ Many complex applications can be easily carried out with Linked Lists.

Disadvantages of Linked List:-

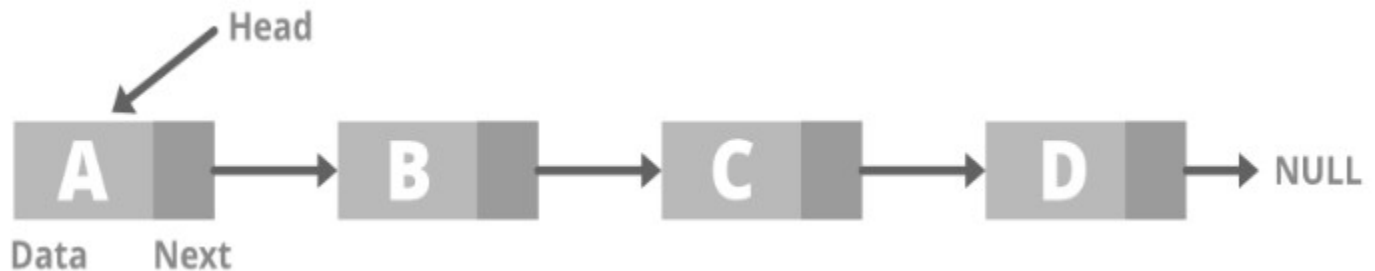
- ✗ It consumes more space because every node requires a additional space to store the address of the next node.
- ✗ Searching is difficult & also time consuming (only linear search)
- ✗ No random access (only sequential access)
- ✗ Reverse traversing is difficult

Applications of linked list in real world:

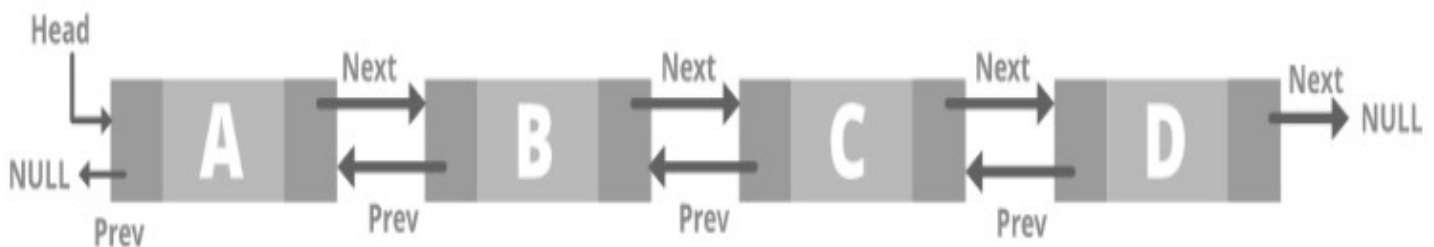
1. **Image viewer** – Previous and next images are linked, hence can be accessed by next and previous button.
2. **Previous and next page in web browser** – We can access previous and next URL searched in web browser by pressing back and next button since, they are linked as linked list.
3. **Music Player** – Songs in music player are linked to previous and next song. We can play songs either from starting or ending of the list.

Types of Linked List:**Singly Linked List:**

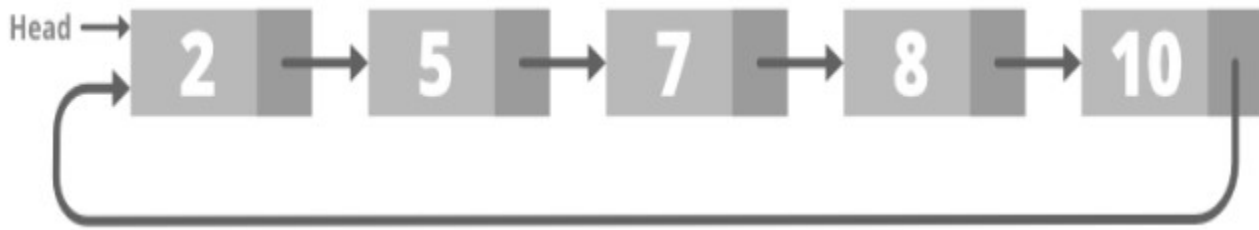
- A singly linked list is one in which each node has only one link field which contains the address of the next node of a list.
- A single linked list allows traversal of data only in one way.

**Doubly Linked List:**

- A doubly linked list or a two-way linked list which contains a pointer to the next node as well as the previous node in the list.
- Doubly Linked list is a list in which each node contains 2 link fields is called doubly Linked List.
- It contains three fields, data, a pointer to the next node, and a pointer to the previous node.
- It allows us to traverse the list in the backward direction as well.

**Circular Linked List:**

- A Circular Linked List is a list in which the link field of the last node contains the address of the first node of a list.
- While traversing a circular linked list, we can begin at any node and traverse the list until we reach the same node we started.



Doubly Circular Linked List:

- In Circular Doubly Linked list the previous pointer of the first node contains address of the last node & next pointer of last node contains the address of the first node
- It also contains three fields, data, a pointer to the next node, and a pointer to the previous node.
- While traversing a circular doubly linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started.



Creation of Singly Linked List:

We can create a singly linked list in python by following the mentioned steps.

Step 1: Create a class “SinglyLinkedList”. Create empty head or first reference and initialize it with None.

Step 2: Create a class “Node”. The objects of this class hold one variable to store the values of the nodes and another variable to store the reference addresses.

Step 3: Create a new node and assign it a value. Set the reference part of this node to None.

Step 4: Since we have only one element in the linked list so far, we will link first to this node by putting in its reference address

The python code to create a new node of the singly linked list is given below.

class Node:

```
def __init__(self, data = None):
    self.data = data
    self.next = None
```

temp = Node(50) will create node with value 50.

The python code to create a singly linked list and initialize first reference.

class SinglyLinkedList:

```
def __init__(self):
    self.first = None
```

sll = SinglyLinkedList()

Prepending Nodes:

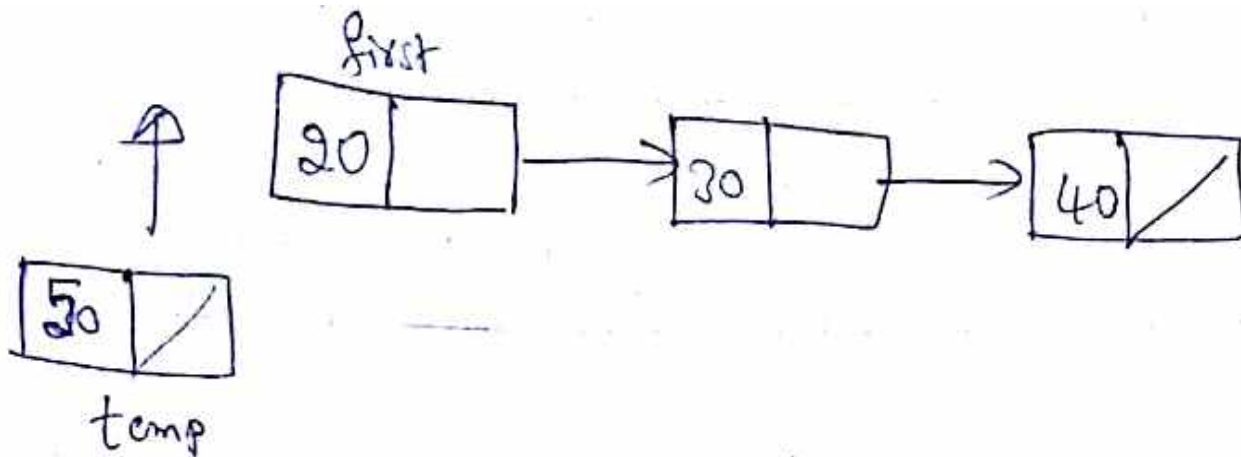
The following are the steps to be followed to insert a new node at beginning of the list.

Step 1:- Create a new node and insert the item into the new node.

Step 2:- If the list is empty then make new node as first. Go to step 4

Step 3:- Assign the “next” reference of the new node as first. Set the created node as the new first.

Step 4: Terminate.



The python code to add new node to the linked list is given below.

```
def insertFirst(self, data):
```

```
    temp = Node(data)
```

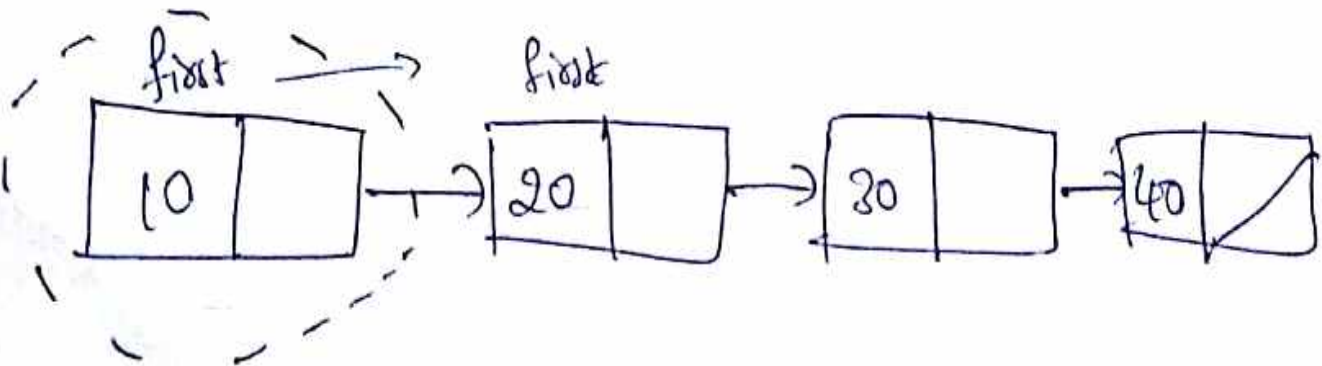
```
    temp.next=self.first
```

```
    self.first=temp
```

Removing Nodes:

The following are the steps to be followed to remove node from the beginning of the list.

- Step 1: If the linked list is empty, return and go to step 5.
- Step 2: If there's only one element, delete that node and set first to none. Go to step 5.
- Step 3: Set a “cur” node pointing at first.
- Step 4: Assign first as the next node. Delete the “cur” node.
- Step 5: Terminate



The python code to remove node from the linked list is given below.

```
def removeFirst(self):
    if(self.first== None):
        print("list is empty")
    else:
        cur=self.first;
        self.first=self.first.next
        print("the deleted item is",cur.data)
```

Traversing the Nodes:

A singly Linked list can only be traversed in forward direction from the first element to the last. We get the value of the next data element by simply iterating with the help of the reference address.

- Step 1: If the linked list is empty, display the message “List is empty” and move to step 5.
- Step 2: Iterate over the linked list using the reference address for each node.
- Step 3: Print every node’s data.
- Step 4: Terminate.

The python code to traversing the nodes of linked list is given below.

```
def display(self):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while(cur):
        print(cur.data, end = " ")
        cur = cur.next
```

Searching for a Node:

- To find a node in a given singly linked list, we use the technique of traversal. In this case as soon as we find the node, we will terminate the loop.
- Algorithm to search a given node from the linked list is given below
 - ✓ Step 1: If the linked list is empty, display the message “List is empty” and move to step 5.
 - ✓ Step 2: Iterate over the linked list using the reference address for each node.
 - ✓ Step 3: Search every node for the given value.
 - ✓ Step 4: If the element is found, print the message “Element found” and return. If not, print the message “Element not found”.
 - ✓ Step 5: Terminate.
- The python code implementation of searching the nodes of linked list is given below.

```
def search(self,item):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
```

```
while cur != None:
    if cur.data == item:
        print("Item is present in the Linked list")
        return
    else:
        cur = cur.next
print("Item is not present in the Linked list")
```

Implementation of Singly Linked List (SLL) (Traversing the Nodes, searching for a Node, Prepending Nodes, and Removing Nodes):

```
class Node:
    def __init__(self, data = None):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.first = None
    def insertFirst(self, data):
        temp = Node(data)
        temp.next=self.first
        self.first=temp
    def removeFirst(self):
        if(self.first== None):
            print("list is empty")
        else:
            cur=self.first
            self.first=self.first.next
            print("the deleted item is",cur.data)
    def display(self):
        if(self.first== None):
            print("list is empty")
            return
        cur = self.first
        while(cur):
            print(cur.data, end = " ")
            cur = cur.next
    def search(self,item):
        if(self.first== None):
            print("list is empty")
            return
        cur = self.first
        while cur != None:
```

```

    if cur.data == item:
        print("Item is Present in the Linked list")
        return
    else:
        cur = cur.next
    print("Item is not present in the Linked list")
#Singly Linked List
sll = SinglyLinkedList()
while(True):
    ch = int(input("\nEnter your choice 1-insert 2-delete 3-search 4-display 5-exit :"))
    if(ch == 1):
        item = input("Enter the element to insert:")
        sll.insertFirst(item)
        sll.display()
    elif(ch == 2):
        sll.removeFirst()
        sll.display()
    elif(ch == 3):
        item = input("Enter the element to search:")
        sll.search(item)
    elif(ch == 4):
        sll.display()
    else:
        break

```

Linked List Iterators:

- Traversals are very common operations, especially on containers.
- A python for loop is used to traverse the items in strings, lists, tuples and dictionaries as follows.

```

print("List iteration")
l1 = [1,2,3,4]
for x in l1:
    print(x)
print("tuple iteration")
t1 = (10,20,30,40)
for x in t1:
    print(x)
print("String iteration")
t1 = "Welcome to gpt koppal"
for x in t1:
    print(x)

```

- An iterators guarantee that each element is visited exactly once.
- Custom created linked list is not iterable; there is a need to add a `__iter__` function to traverse through the list.
- Iterator function is defined as follows

```
def __iter__(self):
    cur = self.first
    while cur:
        yield cur.data
        cur = cur.next
```

Implementation of linked list Iterators:

```
class Node:
    def __init__(self, data = None):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.first = None
    def insert(self, data):
        temp = Node(data)
        if(self.first):
            cur = self.first
            while(cur.next):
                cur = cur.next
            cur.next = temp
        else:
            self.first = temp
    def __iter__(self):
        cur = self.first
        while cur:
            yield cur.data
            cur = cur.next
# Linked List Iterators
ll = LinkedList()
ll.insert(9)
ll.insert(98)
ll.insert("welcome")
ll.insert("govt polytechnic koppal")
ll.insert(456.35)
ll.insert(545)
ll.insert(5)
for x in ll:
    print(x)
```

Time Complexity of Linked List Vs Array:

Operations	Array	Linked List
Creation	$O(1)$	$O(1)$
Insertion at beginning	$O(1)$	$O(1)$
Insertion in between	$O(1)$	$O(1)$
Insertion at end	$O(1)$	$O(n)$
Searching in Unsorted Data	$O(n)$	$O(n)$
Searching in Sorted Data	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$
Deletion at beginning	$O(1)$	$O(1)$
Deletion in between	$O(1)$	$O(n)/O(1)$
Deletion at end	$O(1)$	$O(n)$
Deletion of entire linked list/array	$O(1)$	$O(n)/O(1)$
Accessing elements	$O(1)$	$O(n)$

Week-7

The Doubly Linked List, Examples: Image viewer, music player list etc. (to be used to explain concept of list).
DLL node, List Operations – Create, appending nodes, delete, search.

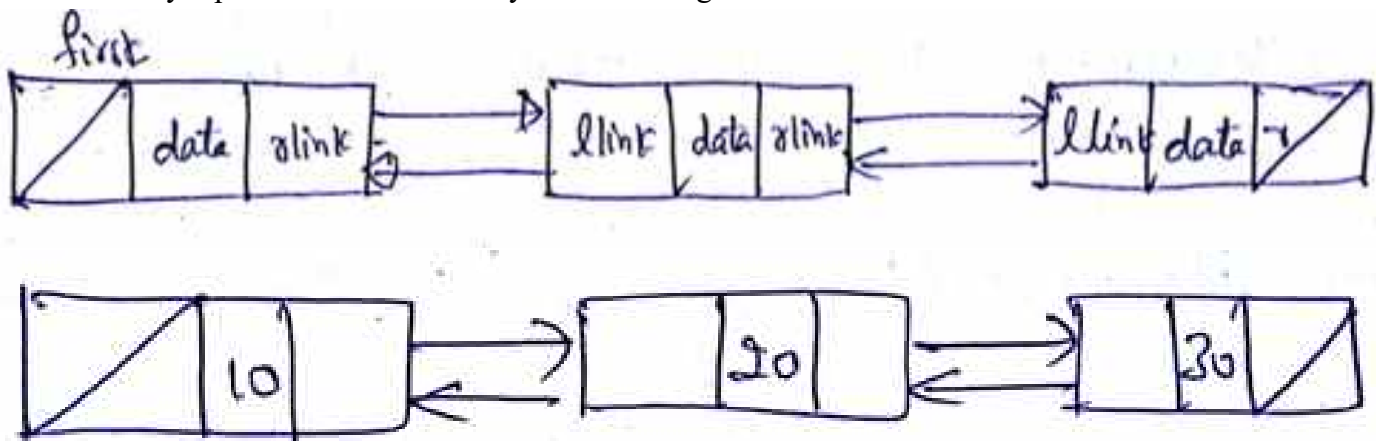
The Circular Linked List-Organization, List Operations – Appending nodes, delete, iterating circular list

The Doubly Linked List:

- A doubly linked list or a two-way linked list which contains a pointer to the next node as well as the previous node in the list.
- A Doubly Linked list is a list in which each node contains 2 link fields is called doubly Linked List.
- A Doubly Linked list is a sequence of data elements, which are connected together via 2 links.
- A Doubly Linked list is a collection of zero or more nodes where each node contains 3 fields, data, llink and rlink. (data, next, prev)
- Data field stores the actual information and llink field contains the address of the previous node and rlink contains the address of the next node.
- It allows us to traverse the list in the backward direction as well.
- The memory representation of a node is given below



- The memory representation of a doubly linked list is given below



- Each node contains 3 fields i.e. data left link and right link.
- The data field contains data or information.
- Right Link field is a pointer which contains the address of the next node.
- Left Link field is a pointer which contains the address of the previous node.
- Last node's **rlink** field will be None and First node's **llink** field will be None.
- In this we can traverse in both directions.
- Every node in the doubly linked list points to the next element and previous elements in the linked list.

Applications of Doubly Linked list in real world:

4. **Image viewer** – Previous and next images are linked, hence can be accessed by next and previous button.
5. **Previous and next page in web browser** – We can access previous and next URL searched in web browser by pressing back and next button since, they are linked as linked list.
6. **Music Player** – Songs in music player are linked to previous and next song. We can play songs either from starting or ending of the list.
7. **Used to implement undo/redo operations.**
8. **In Navigation Systems for front and back navigation.**

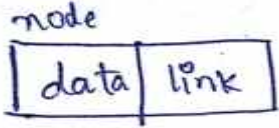
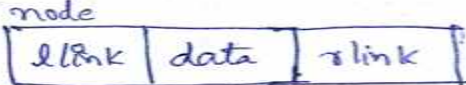
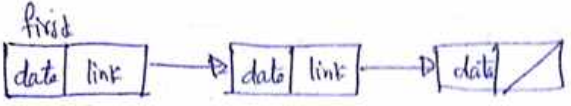

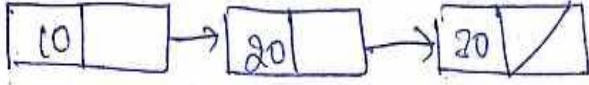
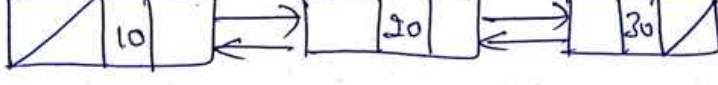
Advantages of DLL:

1. **Reversing a Doubly Linked List is very easy.**
2. **Bi-directional traversal is possible**
3. **Implementation of DLL is easy.**

Disadvantages of DLL:

1. **Uses extra memory for one extra link per node.**
2. **No random access to elements in DLL.**

Compare Singly Linked List with Doubly Linked List:

Singly Linked List	Doubly Linked List
Collection of nodes and each node contains of 2 fields i.e. data & link.	Collection of nodes & each node contains 3 fields i.e. data, left link and right link.
Node is represented as follows 	Node is represented as follows 
The elements can be accessed using link field only	The elements can be accessed using both left link and right link fields.
It contains only the address of the next node.	It contains address of both right node and left node.
It takes less memory	It takes more memory.
Less efficient access to elements	More efficient access to elements
Singly Linked List is represented as follows 	Doubly Linked List is represented as follows 
Ex:- 	Ex:- 

DLL Nodes:

- In Doubly Linked list is each node contains 3 fields, data, next and prev.
- The **data** field stores the actual information.
- The **next** field contains the address of the previous node.
- The **prev** field contains the address of the next node.
- The python code to create a new node of the doubly linked list is given below.

class Node:

def __init__(self, data = None):

self.data = data

self.next = None

self.prev = None

- **temp = Node(10)** will create a node with value 10.

List Operations - Creation of Doubly Linked List:

We can create a singly linked list in python by following the mentioned steps.

Step 1: Create a node class with 3 required variables.

Step 2: Create the Doubly Linked List class and declare the first node.

Step 3: Create a new node and assign it a value.

Step 4: Set the **next** reference of the newly created node to **None**.

Step 5: Set the **prev** reference of the newly created node to **None**.

Step 6: we will link first to this node by putting in its reference address

Step 7: Terminate.

The python code to create a new node of the doubly linked list is given below.

class Node:

def __init__(self, data = None):

self.data = data

self.next = None

self.prev = None

The python code to create a doubly linked list and initialize first reference.

class DoublyLinkedList:

def __init__(self):

self.first = None

dll = DoublyLinkedList()

List Operations - Appending Nodes:

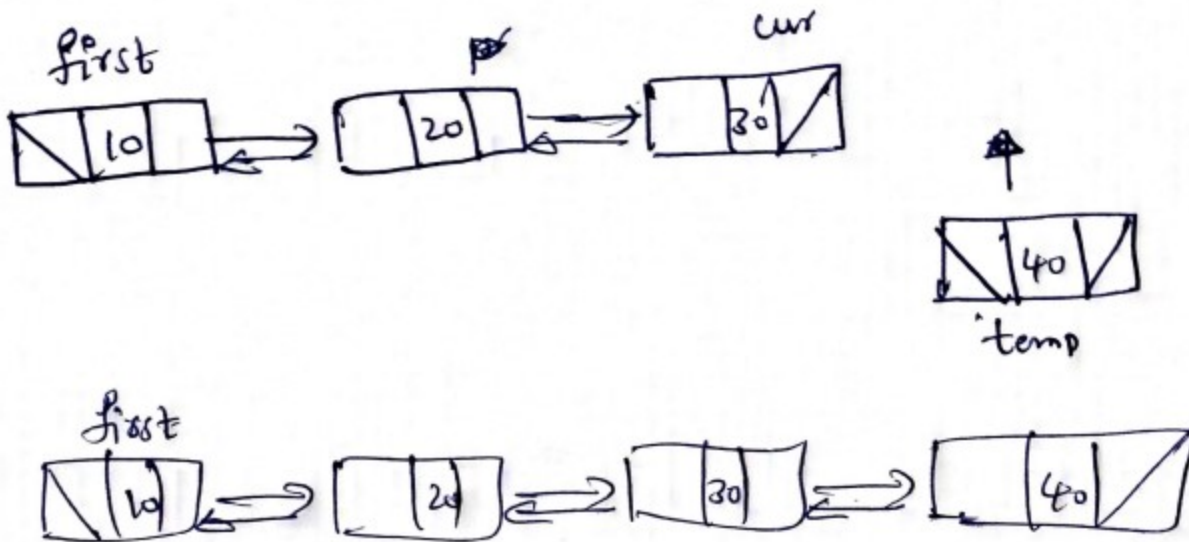
The following are the steps to be followed to append a new node to the doubly linked list.

Step 1:- Create a new node and assign a value to it.

Step 2:- If the list is empty then make new node as first. Go to step 4

Step 3:- Search for the last node. Once found set its **next** reference pointing to the new node and set the **prev** reference of the new node to the last node.

Step 4: Terminate.



The python code to append new node to the doubly linked list is given below.

```
def insertAtEnd(self, data):
    temp = Node(data)
    if(self.first == None):
        self.first=temp
    else:
        cur = self.first
        while(cur.next != None):
            cur = cur.next
        cur.next = temp
        temp.prev = cur
```

List Operations - Delete:

The following are the steps to be followed to remove node from the beginning of the list.

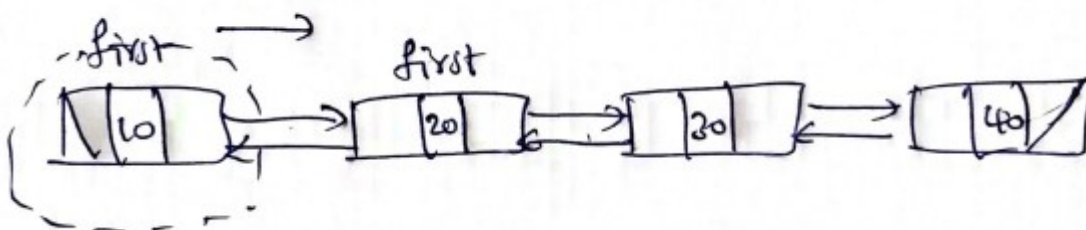
Step 1: If the linked list is empty, return and go to step 5.

Step 2: If there's only one element, delete that node and set first to **None**. Go to step 5.

Step 3: Set a "cur" node pointing at first node.

Step 4: Assign first as the next node. Set the **prev** reference of first node to **None**. Delete the cur node.

Step 5: Terminate



The python code to delete node from the linked list is given below.

```
def deleteFirst(self):  
    if(self.first== None):  
        print("list is empty")  
    elif(self.first.next == None):  
        print("the deleted item is",self.first.data)  
        self.first = None  
    else:  
        cur=self.first  
        self.first=self.first.next  
        self.first.prev = None  
        print("the deleted item is",cur.data)
```

List Operations - Traverse:

- A Doubly Linked list can only be traversed in forward direction from the first node to the last and in backward direction from last node to first.
- We get the value of the next data element by simply iterating with the help of the reference addresses **next** and **prev**.
- The following are the steps to be followed to traverse the doubly linked list.

Step 1: If the linked list is empty, display the message “List is empty” and move to step 4.

Step 2: Iterate over the linked list using the reference address **next** for each node until the last node.

Step 3: Print every node’s data.

Step 4: Terminate.

- The python code to traverse the nodes of doubly linked list is given below.

```
def display(self):  
    if(self.first== None):  
        print("list is empty")  
        return  
    current = self.first  
    while(current):  
        print(current.data, end = " ")  
        current = current.next
```

List Operations - Searching for a Node:

- To find a node in a given doubly linked list, we use the technique of traversal. In this case as soon as we find the node, we will terminate the loop.
- Algorithm to search a given node from the linked list is given below
Step 1: If the linked list is empty, display the message “List is empty” and move to step 5.
Step 2: Iterate over the linked list using the reference address **next** for each node.
Step 3: Search every node for the given value.
Step 4: If the element is found, print the message “Element found” and return. If not, print the message “Element not found”.
Step 5: Terminate.
- The python code implementation of searching the nodes of linked list is given below.

```
def search(self,item):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while cur != None:
        if cur.data == item:
            print("Item is present in the Linked list")
            return
        else:
            cur = cur.next
    print("Item is not present in the Linked list")
```

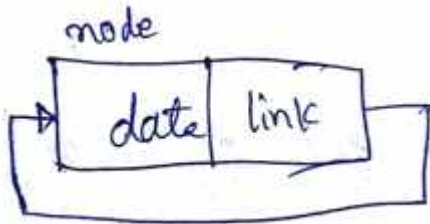
Implementation of Doubly linked list (Traversing the Nodes, searching for a Node, Appending Nodes, Deleting Nodes):

```
class Node:
    def __init__(self, data = None):
        self.data = data
        self.next = None
        self.prev = None
class DoublyLinkedList:
    def __init__(self):
        self.first = None
    def insertAtEnd(self, data):
        temp = Node(data)
        if(self.first == None):
            self.first=temp
        else:
            cur = self.first
            while(cur.next != None):
                cur = cur.next
            cur.next = temp
            temp.prev = cur
    def deleteFirst(self):
        if(self.first== None):
            print("list is empty")
        elif(self.first.next == None):
            print("the deleted item is",self.first.data)
            self.first = None
        else:
            cur=self.first
            self.first=self.first.next
```

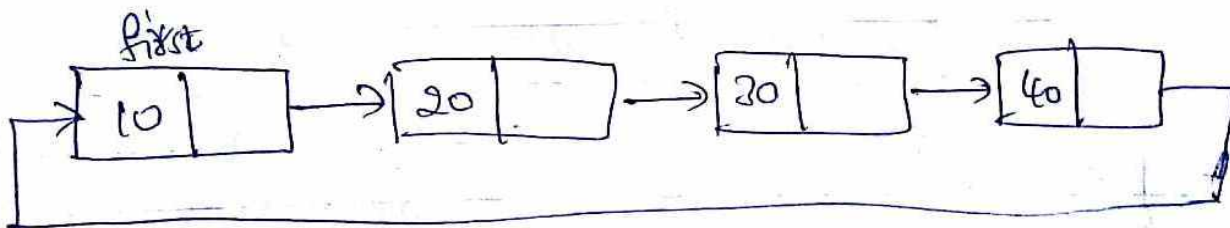
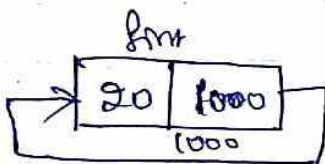
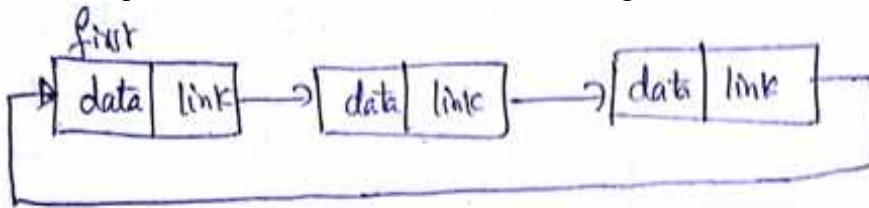
```
        self.first.prev = None
        print("the deleted item is",cur.data)
def display(self):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while(cur):
        print(cur.data, end = " ")
        cur = cur.next
def search(self,item):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while cur != None:
        if cur.data == item:
            print("Item is present in the Linked list")
            return
        else:
            cur = cur.next
    print("Item is not present in the Linked list")
#Doubly Linked List
dll = DoublyLinkedList()
while(True):
    ch = int(input("\nEnter your choice 1-insert 2-delete 3-search 4-display 5-exit :"))
    if(ch == 1):
        item = input("Enter the element to insert:")
        dll.insertAtEnd(item)
        dll.display()
    elif(ch == 2):
        dll.deleteFirst()
        dll.display()
    elif(ch == 3):
        item = input("Enter the element to search:")
        dll.search(item)
    elif(ch == 4):
        dll.display()
    else:
        break
```

Circular Linked List:

- A Circular Linked List is a list in which the link field of the last node contains the address of the first node of a list.
- While traversing a circular linked list, we can begin at any node and traverse the list until we reach the same node we started.
- A Singly Linked List can be converted into Circular Linked List by simply replacing the **None** value of the last node's link field to the address of the very first node of the Singly Linked List.
- In Circular Linked List the first node follows the last node therefore we have no first node or last node explicitly in a Circular Linked List.
- In a Circular Linked List from any given node we can move forward to the starting point.
- The Circular Linked List node is represented as follows.



- Each node contains two fields i.e. data and link.
- Pictorial representation of Circular Linked List is given below



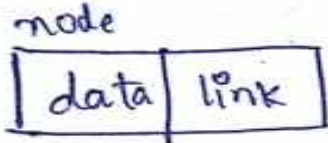

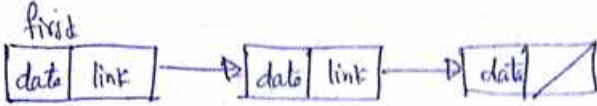
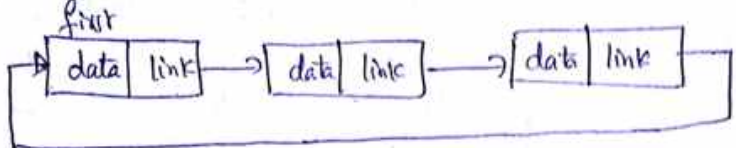
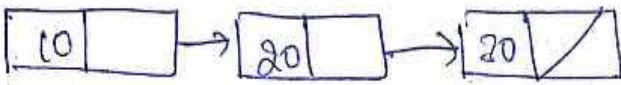
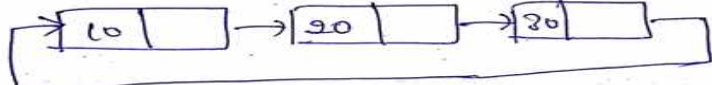
- A Circular Linked List Node is created with the following syntax.

```
class Node:
    def __init__(self, data = None):
        self.data = data
        self.next = None
```

where

- data field contains information
- next field contains address of the next node.

Compare Singly Linked List with Circular Linked List:

Singly Linked List	Circular Linked List
It is a Linear Linked List in which nodes are arranged linearly adjacent to each other.	It is a kind of Linked List in which the nodes are adjacent to each other but the first node appears next to the last node.
Link field of the last node is NULL	Link field of last node contains the address of the first node.
During traversal once we reach the last node it is not possible to visit the first node.	During traversal it is very easy to visit first node from the last node.
In Singly Linked List node is represented as follows 	In Circular Linked List node is represented as follows 
Singly Linked List is represented as follows 	Circular Linked List is represented as follows. 
Ex: 	Ex:- 

List Operations - Creation of Circular Linked List:

We can create a Circular linked list in python by following the mentioned steps.

Step 1: Create a node class with 2 required variables.

Step 2: Create the Circular Linked List class and declare the first node.

Step 3: Create a new node and assign it a value.

Step 4: Set the **next** reference of the newly created node to **itself**.

Step 5: we will link first to this node by putting in its reference address

Step 6: Terminate.

The python code to create a new node of the doubly linked list is given below.

class Node:

```
def __init__(self, data = None):
    self.data = data
```



```
self.next = None
```

The python code to create a circular linked list and initialize first reference.

```
class CircularLinkedList:
```

```
    def __init__(self):
```

```
        self.first = None
```

```
cil = CircularLinkedList ()
```

List Operations - Appending Nodes:

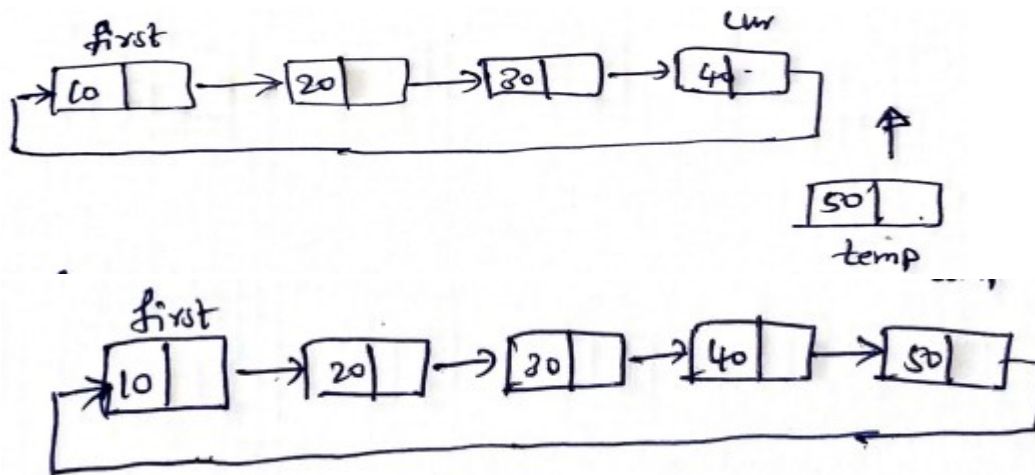
The following are the steps to be followed to append a new node to the circular linked list.

Step 1:- Create a new node and assign a value to it.

Step 2:- If the list is empty then make new node as first. Go to step 4

Step 3:- Search for the last node. Once found set its **next** reference to the new node and set the **next** reference of the new node to the first node.

Step 4: Terminate.



The python code to append new node to the circular linked list is given below.

```
def insertAtEnd(self, data):
```

```
    temp = Node(data)
```

```
    if(self.first == None):
```

```
        self.first = temp
```

```
        self.first.next = temp
```

```
    else:
```

```
        cur = self.first
```

```
        while(cur.next != self.first):
```

```
            cur = cur.next
```

```
        cur.next = temp
```

```
        temp.next = self.first
```

List Operations - Delete:

The following are the steps to be followed to delete a node at the end of the list.

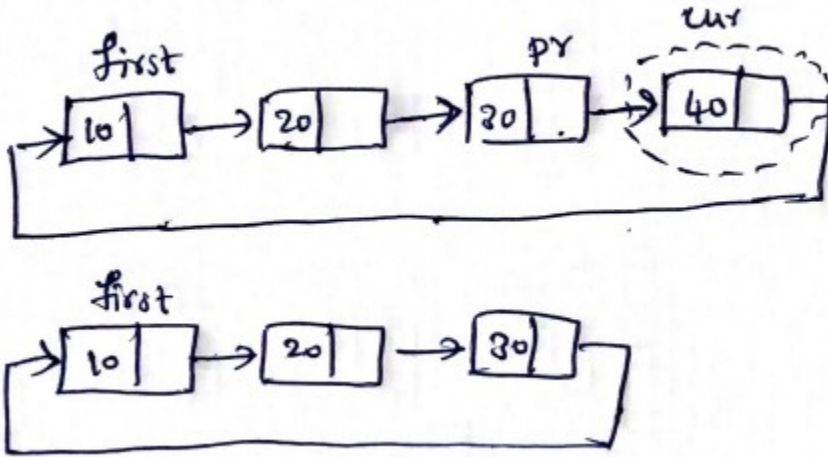
Step 1: If the linked list is empty, return and go to step 5.

Step 2: If there's only one element, delete that node and set first to **None**. Go to step 5.

Step 3: Set a "cur" node pointing at first node.

Step 4: Search for the last node and its previous node. Once found assign its previous node's next reference to first and delete the last node.

Step 5: Terminate



The python code to delete node from the circular linked list is given below.

```
def deleteAtEnd(self):
    if(self.first == None):
        print("list is empty")
    elif(self.first.next == self.first):
        print("the deleted item is", self.first.data)
        self.first = None
    else:
        cur = self.first
        while(cur.next != self.first):
            pr = cur
            cur = cur.next
        pr.next = self.first
        print("the deleted item is", cur.data)
```

List Operations - Traverse:

- A Circular Linked list can only be traversed in forward direction from the first node to the last.
- We get the value of the next data element by simply iterating with the help of the reference address.
- The following are the steps to be followed to traverse the circular linked list.

Step 1: If the linked list is empty, display the message "List is empty" and move to step 4.

Step 2: Iterate over the linked list using the reference address **next** for each node until the last node.

Step 3: Print every node's data.

Step 4: Terminate.

- The python code to traversing the nodes of linked list is given below.

```
def display(self):
    if(self.first == None):
```

```

    print("list is empty")
    return
cur = self.first
while(True):
    print(cur.data, end = " ")
    cur = cur.next
    if(cur == self.first):
        break

```

List Operations - Searching for a Node:

- To find a node in a given circular linked list, we use the technique of traversal. In this case as soon as we find the node, we will terminate the loop.
- Algorithm to search a given node from the linked list is given below
 - Step 1: If the linked list is empty, display the message “List is empty” and move to step 5.
 - Step 2: Iterate over the linked list using the reference address **next** for each node.
 - Step 3: Search every node for the given value.
 - Step 4: If the element is found, print the message “Element found” and return. If not, print the message “Element not found”.
 - Step 5: Terminate.
- The python code implementation of searching the nodes of linked list is given below.

```

def search(self,item):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while cur.next != self.first:
        if cur.data == item:
            print("Item is present in the linked list")
            return
        else:
            cur = cur.next
    print("Item is not present in the linked list")

```

Implementation of Circular linked list (Traversing the Nodes, searching for a Node, Appending Nodes, and Deleting Nodes):

```

class Node:
    def __init__(self, data = None):
        self.data = data
        self.next = None
class CircularLinkedList:
    def __init__(self):
        self.first = None
    def insertAtEnd(self, data):

```

```
temp = Node(data)
if(self.first == None):
    self.first = temp
    self.first.next = temp
else:
    cur = self.first
    while(cur.next != self.first):
        cur = cur.next
    cur.next = temp
    temp.next = self.first
def deleteAtEnd(self):
    if(self.first== None):
        print("list is empty")
    elif(self.first.next == self.first):
        print("the deleted item is",self.first.data)
        self.first = None
    else:
        cur=self.first
        while(cur.next != self.first):
            pr = cur
            cur = cur.next
        pr.next = self.first
        print("the deleted item is",cur.data)
def display(self):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while(True):
        print(cur.data, end = " ")
        cur = cur.next
        if(cur == self.first):
            break
def search(self,item):
    if(self.first== None):
        print("list is empty")
        return
    cur = self.first
    while cur.next != self.first:
        if cur.data == item:
            print("Item is present in the linked list")
            return
    else:
```

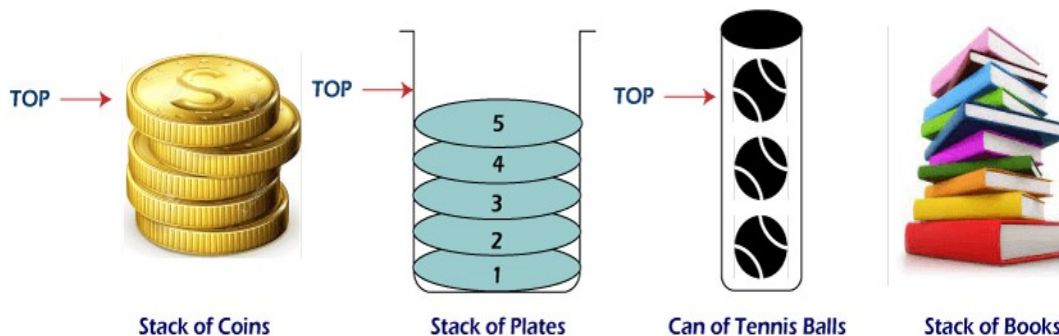
```
        cur = cur.next
    print("Item is not present in the linked list")
#Circular Linked List
ccl = CircularLinkedList()
while(True):
    ch = int(input("\nEnter your choice 1-insert 2-delete 3-search 4-display 5-exit :"))
    if(ch == 1):
        item = input("Enter the element to insert:")
        ccl.insertAtEnd(item)
        ccl.display()
    elif(ch == 2):
        ccl.deleteAtEnd()
        ccl.display()
    elif(ch == 3):
        item = input("Enter the element to search:")
        ccl.search(item)
    elif(ch == 4):
        ccl.display()
    else:
        break
```

Week-8

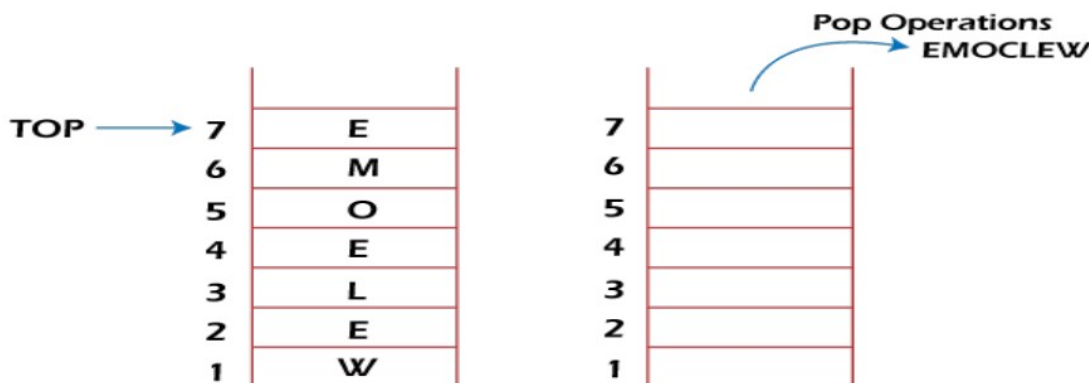
Last In First Out (Stack) Data structures – Example: Reversing a word, evaluating an expression, message box etc. (to be used to explain concept of LIFO). The Stack implementation – push, pop, display. Stack Applications- Balanced Delimiters, Evaluating Postfix Expressions.

The Last In First Out (Stack) Data structure:

- Stack: A Stack is special type of linear data structure where elements are inserted and deleted from same end, commonly referred to as the **top** of the stack.
- Stack is a **LIFO** (Last in First out) data structure.
- A stack is used to store data such that the last item inserted is the first item to be removed.
- Some examples are Stack of books, Stack of plates in a cafeteria, tokens in bank, and clothes in a shelf, etc.

**Applications of Stack:**

1. **Reverse a String:** A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the last in first out (LIFO) property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



2. **Evaluating an Expression:** A stack is a very effective data structure for evaluating arithmetic expressions.
3. **Conversion of expressions:** A stack is used to convert one form of expression to another form.
4. **Message Box**

Stack ADT:

A stack is a data structure that stores a linear collection of items with access limited to a last-in first-out order. Adding and removing items is restricted to one end known as the top of the stack. An empty stack is one containing no items.

- ✓ `Stack()`: Creates a new empty stack.
- ✓ `push(item)`: Adds the given item to the top of the stack.
- ✓ `pop()`: Removes and returns the top item of the stack, if the stack is not empty. Items cannot be popped from an empty stack. The next item on the stack becomes the new top item.
- ✓ `peek()`: Returns a reference to the item on top of a non-empty stack without removing it. Peeking, which cannot be done on an empty stack, does not modify the stack contents.
- ✓ `isEmpty()`: Returns a boolean value indicating if the stack is empty.
- ✓ `length ()`: Returns the number of items in the stack.

The Stack Implementation:

Let us implement a very simple stack using Python's list class. The operations that must be performed on the stack are encapsulated in the Stack class:

class Stack:

```
def __init__(self):  
    self.items = []
```

In the initialization method `__init__`, the `items` instance variable is set to `[]`, which means the stack is empty when created.

Push operation:

- The push operation is used to insert items to the stack.
- In stack items are always inserted from the top.
- The push operation or method uses the `append` method of the list class to insert items (or data) to the stack.
- The python code of push operation is given below.

```
def push(self,item):  
    self.items.append(item)
```

Pop operation:

- The pop operation is used to remove items from the stack.
- In stack items are always deleted from top end.
- The python code of pop operation is given below.

```
def pop(self):  
    if self.isEmpty():  
        print("Stack is Empty")  
    else:  
        item = self.items[-1]  
        del(self.items[-1])  
        print("The popped element is:",item)
```


Peek Operation:

- Peek method will return the top of the stack without removing it from the stack.
- If there is a top element, return its data, otherwise display message.
- The python code of peek operation is given below.

```
def peek(self):  
    if self.isEmpty():  
        print("Stack is Empty")  
    else:  
        print("Top item is ", self.items[-1])
```

Display Operation:

- Display method will display the all the elements of stack.
- The python code of display operation is given below.

```
def display(self):  
    if self.isEmpty():  
        print("Stack is Empty")  
    else:  
        for i in reversed(self.items):  
            print(i)
```

Complete Implementation of Stack data structure:

```
class Stack:  
    def __init__(self):  
        self.items = []  
    def isEmpty(self):  
        return len(self.items) == 0  
    def push(self,item):  
        self.items.append(item)  
    def pop(self):  
        if self.isEmpty():  
            print("Stack is Empty")  
        else:  
            item = self.items[-1]  
            del(self.items[-1])  
            print("The popped element is:",item)  
    def display(self):  
        if self.isEmpty():  
            print("Stack is Empty")  
        else:  
            for i in reversed(self.items):  
                print(i)  
    def peek(self):  
        if self.isEmpty():
```









```

    print("Stack is Empty")
else:
    print("Top item is ", self.items[-1])
s = Stack()
while(True):
    print("1:push 2:pop 3:display 4:peek 5:exit")
    choice = int(input("Enter your choice:"))
    if choice == 1:
        item = input("Enter the item to push:")
        s.push(item)
    elif choice == 2:
        s.pop()
    elif choice == 3:
        s.display()
    elif choice == 4:
        s.peek()
    else:
        break

```

Stack Applications: Balanced Delimiters:

- A number of applications use delimiters to group strings of text or simple data into sub-parts by marking the beginning and end of the group.
- The delimiters must be used in pairs of corresponding types: {}, [], and ().
- They must also be positioned such that an opening delimiter within an outer pair must be closed within the same outer pair.
- The Stack is a perfect data structure for implementing an algorithm which will be used to check balanced parameters.
- Each time, when an opening delimiter is encountered push it in the stack. When a closing delimiter is encountered, we pop the opening delimiter from the stack and compare it to the closing delimiter. If stack is empty at the end, return balanced otherwise, Unbalanced.

Brackets	Matched	Balanced
{ () } []		
{ (}) []		
{ () [] }		
{ }) ([]		

Ex1: $\{A + (B * C) - (D / [E + F])\}$

Operation	Stack	Current Scan Line
Push {	{	{
Push ({({A+(
Pop (& Match)	{	{A+(B*C)-
Push ({({A+(B*C)- (
Push [{([{A+(B*C)- (D/[
Pop [& Match]	{({A+(B*C)- D/[E+F]
Pop (& Match)	{	{A+(B*C)- D/[E+F])
Pop { & Match	Empty	

The expression $\{A + (B * C) - (D / [E + F])\}$ is Balanced

Ex2: $(A + [B * C]) - \{D / E\}$

Operation	Stack	Current Scan Line
Push (((
Push [([{A+[
Pop (& Match)	([- Error	{A+[B*C)

The expression $(A + [B * C]) - \{D / E\}$ is Unbalanced

Implement Bracket matching using Stack.

class Stack:

def __init__(self):

self.items = []

def push(self,item):

self.items.append(item)

def pop(self):

if len(self.items) is 0:

print("Stack is Empty")

else:

item = self.items[-1]

del(self.items[-1])

return item

def check_brackets(expr):

s = Stack()

for token in expr:

if token in "{[(":

s.push(token)

elif token in "}]":

if len(s.items) == 0:

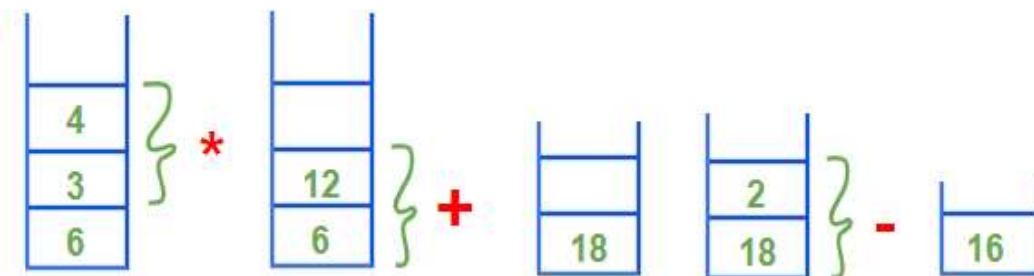
```

    return False
else:
    left = s.pop()
    if (token == "}" and left != "{" ) or \
        (token == "]" and left != "[" ) or \
        (token == ")" and left != "(" ) :
        return False
if len(s.items) == 0:
    return True
expr =input("Enter the Expertion:")
result = check_brackets(expr)
if result:
    print("The Given Expression is Valid")
else:
    print("The Given Expression is Invalid")

```

Evaluating Postfix Expressions:

- There are 3 notations to represent a mathematical expression.
- The most common is the infix notation where the operator is placed between the operands **A+B**.
- The prefix notation places the operator before the operands **+AB**.
- The postfix notation places the operator after the operands **AB+**.
- The stack is used to evaluate the postfix expressions.
- The algorithm of evaluating postfix expression is given below
 1. For each token from the postfix expression, we perform the following steps:
 2. If the current item is an operand, push its value onto the stack.
 3. If the current item is an operator:
 - a) Pop the top two operands from the stack.
 - b) Perform the operation.
 - c) Push the result of this operation back onto the stack.
 4. The final result of the expression will be the last value on the stack.
- Ex1: Evaluate the postfix expression 634*+2-



Week-9

Recursion: Properties of Recursion. Recursive functions: Factorials, Recursive Call stack, The Fibonacci Sequence. How Recursion Works? The Run Time Stack. Recursive Applications- Recursive Binary Search, Towers of Hanoi.

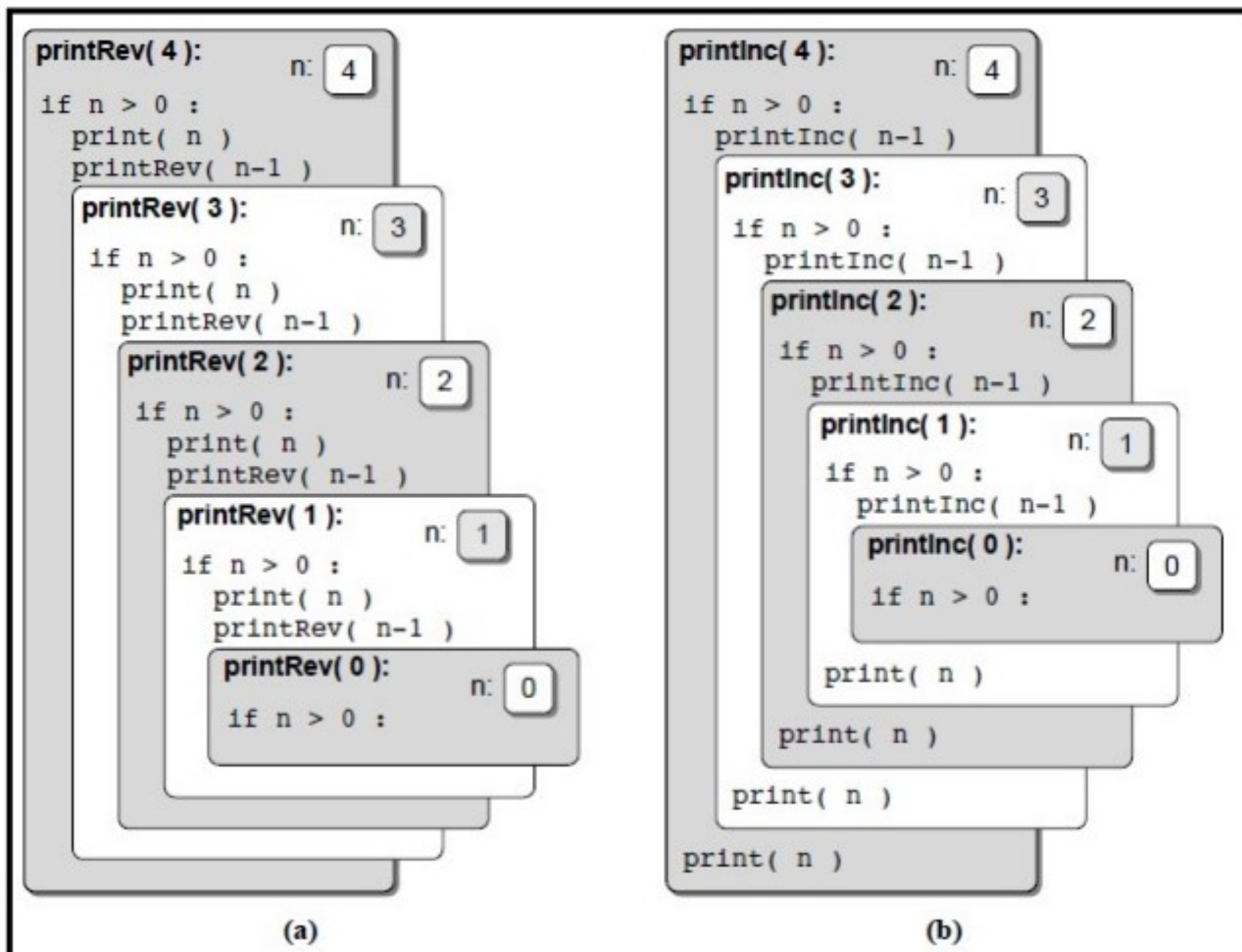
Recursion:

- A function (or method) can call any other function (or method), including itself.
- A function that calls itself is known as a recursive function.
- Consider the simple problem of printing the integer values from 1 to n in reverse order:

```
def printRev( n ):
    if n > 0 :
        print( n )
        printReverse( n-1 )
```

We call the function with an argument of 4:

printRev(4)



- The body of `printRev()` begins execution at the first statement. Since 4 is greater than 0, the body of the if statement is executed.

- When the flow of execution reaches the `printRev(3)` function call, the sequential flow is once again interrupted as control is transferred to another instance of the `printRev()` function.
- The body of this instance begins execution at the first statement, but this time with $n = 3$.
- Above Figure illustrates the execution of the recursive function as a group of boxes with each box representing a single invocation of the `printRev()` function.
- The boxes contain the contents of local variables and only the statements of the function actually executed.

Properties of Recursion:

All recursive solutions must satisfy three rules or properties:

1. A recursive solution must contain a **base case**: The base case is the terminating case. It indicates the end of the recursive calls. In `printRev()`, the base case occurred when $n = 0$ and the function simply returned without performing any additional operations.
2. A recursive solution must contain a **recursive case**: In the `printRev()` function, the recursive case is performed for all values of $n > 0$.
3. A recursive solution must make **progress toward the base case**: A recursive solution must make progress toward the base case, otherwise recursion will never stop.

Recursive functions:

Factorials

- The factorial of a positive integer n can be used to calculate the number of permutations of n elements. The function is defined as:

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

with the special case of $0! = 1$.

- Consider the factorial function on different integer values:

$$0! = 1$$

$$1! = 2 * 1$$

$$2! = 3 * 2 * 1$$

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

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

- After careful inspection of these equations, it becomes obvious each of the successive equations, for $n > 1$, can be rewritten in terms of the previous equation

$$0! = 1$$

$$1! = 1 * (1 - 1)!$$

$$2! = 2 * (2 - 1)!$$

$$3! = 3 * (3 - 1)!$$

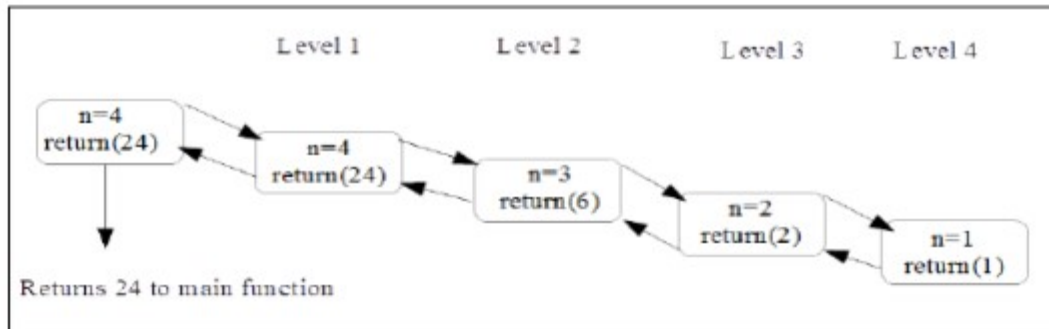
$$4! = 4 * (4 - 1)!$$

$$5! = 5 * (5 - 1)!$$

- Since the function is defined in terms of itself and contains a base case, a recursive definition can be produced for the factorial function as shown here.

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

- We can visualize the process of finding factorial of 4 as shown below



- The python code to implement factorial of a number using recursion is given below

```

def fact(n):
    if n == 1:
        return 1
    else:
        return (n * fact(n-1))
n=int(input("Enter the number:"))
print("The factorial of a number is:",fact(n))
  
```

The Fibonacci sequence:

- The Fibonacci sequence is a sequence of integer values in which the first two values are both 1 and each subsequent value is the sum of the two previous values.
- The first 11 terms of the sequence are:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 . . .
- The n^{th} Fibonacci number can be computed by the recursive relation (for $n > 0$)

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

- The python code to implement n^{th} Fibonacci sequence of a number using recursion is given below

```

def fib(n):
    if n<=1:
        return n
    return fib(n-1) + fib(n-2)
n=int(input("Enter the range:"))
print("The fibonacci value is:",fib(n))
  
```

How Recursion Works

- When a function is called, the sequential flow of execution is interrupted and execution jumps to the body of that function.
- When the function terminates, execution returns to the point where it was interrupted before the function was invoked.

The Run Time Stack:

- Each time a function is called, an activation record is automatically created in order to maintain information related to the function. One piece of information is the return address. This is the location of the next instruction to be executed when the function terminates. Thus, when a function returns, the address is obtained from the activation record and the flow execution can return to where it left off before the function was called.
- The activation records also include storage space for the allocation of local variables. Remember, a variable created within a function is local to that function and is said to have local scope. Local variables are created when a function begins execution and are destroyed when the function terminates. The lifetime of a local variable is the duration of the function in which it was created.
- An activation record is created per function call, not on a per function basis. When a function is called, an activation record is created for that call and when it terminates the activation record is destroyed.
- The system must manage the collection of activation records and remember the order in which they were created. It does this by storing the activation records on a **run time stack**.
- The run time stack is just like the stack structure but it's hidden from the programmer and is automatically maintained.

Recursive Applications:

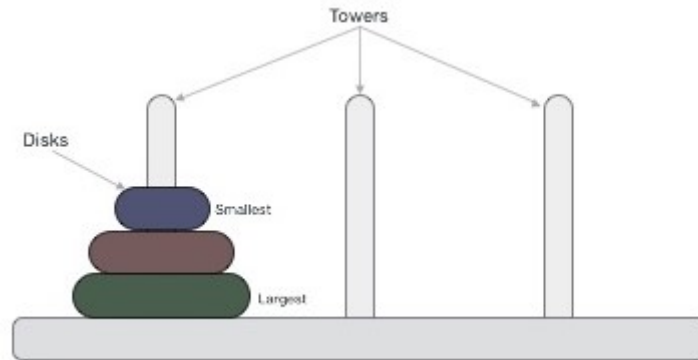
Recursive Binary Search:

- In searching for a target within the sorted sequence, the middle value is examined to determine if it is the target. If not, the sequence is split in half and either the lower half or the upper half of the sequence is examined depending on the logical ordering of the target value in relation to the middle item. In examining the smaller sequence, the same process is repeated until either the target value is found or there are no additional values to be examined.
- A recursive implementation of the binary search algorithm is provided below.

```
def binarysearch(a, low, high, key):
    if low <= high:
        mid = (high + low) // 2
        if a[mid] == key:
            print("Search Successful key found at location:", mid+1)
            return
        elif key < a[mid]:
            binarysearch(a, low, mid-1, k)
        else :
            binarysearch(a, mid + 1, high, k)
    else:
        print("Search UnSuccessful")
a = [13,24,35,46,57,68,79]
print("the array elements are:", a)
k = int(input("enter the key element to search:"))
binarysearch(a, 0, len(a)-1, k)
```

Tower of Hanoi:

- Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is depicted as



- These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules:-

- The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are
 - ✓ Only one disk can be moved among the towers at any given time.
 - ✓ Only the "top" disk can be removed.
 - ✓ No large disk can sit over a small disk.
- We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part. Our ultimate aim is to move nth disk from source to destination and then put all other (n-1) disks onto it.
- The steps to follow are
 - ✓ Step 1 – Move n-1 disks from source to auxiliary
 - ✓ Step 2 – Move nth disk from source to destination
 - ✓ Step 3 – Move n-1 disks from aux to destination
- The python code to implement tower of hanoi using recursion is given below

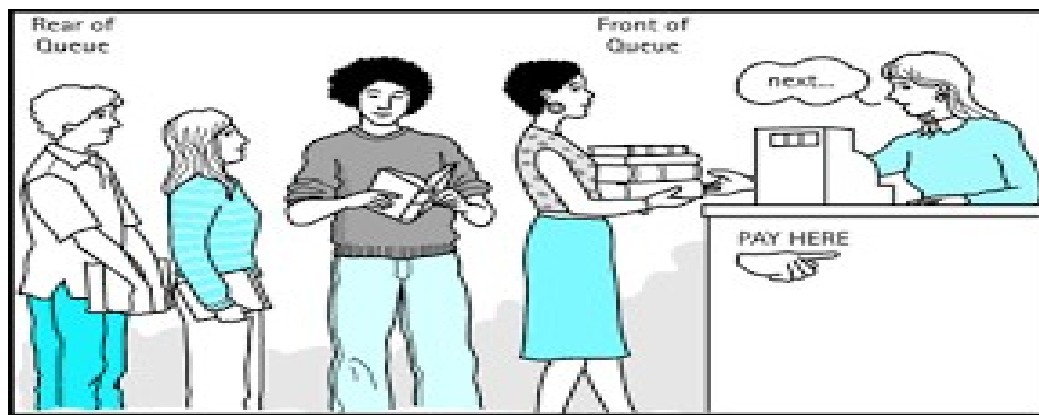
```
def towerofhanoi(n, source, destination, auxiliary):
    if n==1:
        print ("Move disk 1 from source",source,"to destination",destination)
        return
        towerofhanoi(n-1, source, auxiliary, destination)
        print ("Move disk",n,"from source",source,"to destination",destination)
        towerofhanoi(n-1, auxiliary, destination, source)
n = 4
towerofhanoi(n,'A','B','C')
```

Week-10

The First In First Out (Queue) Data structure – Example: Media player list, keyboard buffer queue, printer queue etc. (to be used to explain concept of FIFO). Implementing the Queue and its operations using Python List. Priority Queues, Implementation.

The First In First Out (Queue) Data structure:

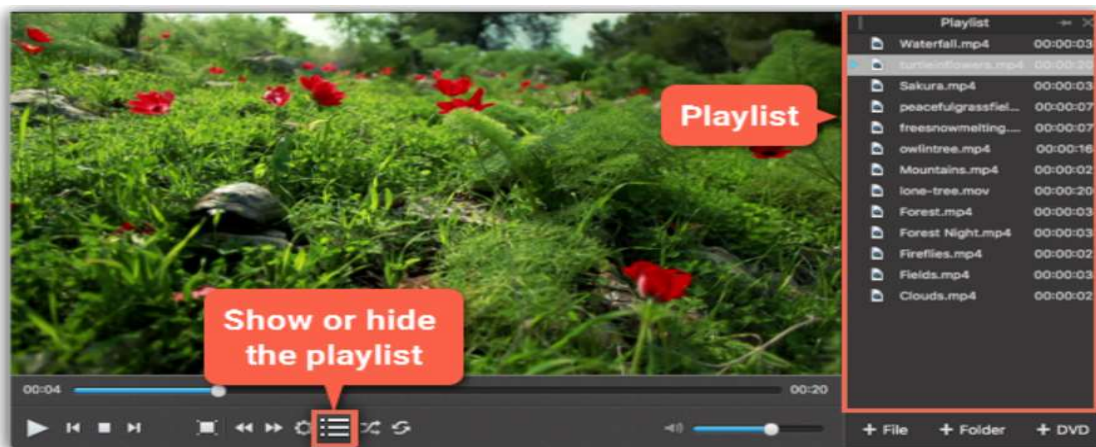
- Queue: A Queue is special type of data structure where elements are inserted from one end called **rear** and elements are deleted from another end called **front**.
- Queue is a FIFO (First in First out) data structure.
- A queue can be illustrated with line of people waiting for the bus at a bus station, a list of calls put on hold to be answered by a telephone operator is a queue, and a list of waiting jobs to be processed by a computer is a queue.
- The pictorial representation of the queue is given below.



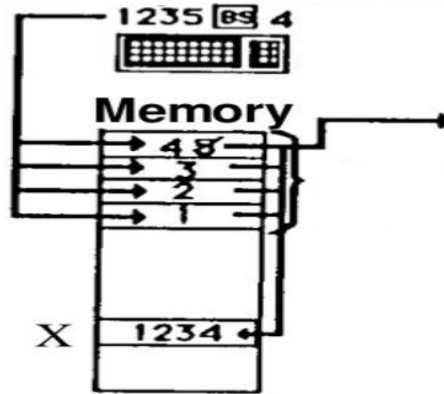
A queue data structure is well suited for problems in computer science that require data to be processed in the order in which it was received.

Examples:

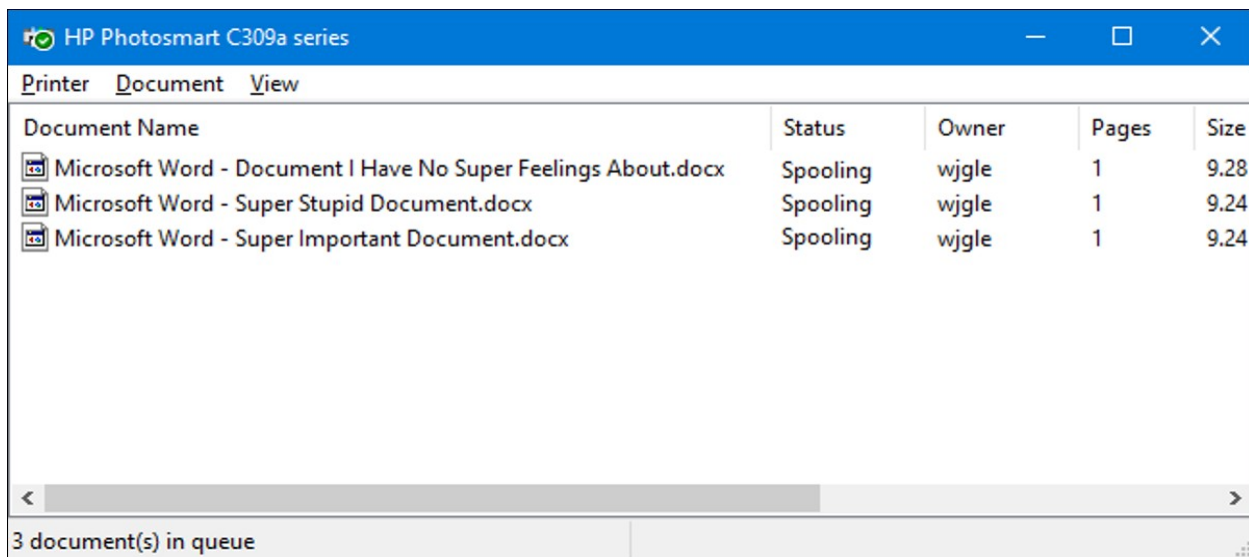
- ✓ **Media player list:** Music player software allows users the chance to add songs to a playlist. Upon hitting the play button, all the songs in the main playlist are played one after the other. The sequential playing of the songs can be implemented with queues because the first song to be queued is the first song that is played.



- ✓ **Keyboard buffer queue:** When users type in the characters using keyboard they stored in a queue in a keyboard buffer and given as input to the application.



- ✓ **Printer queue:** A job to be printed is stored on disk in a queue. The printer prints them in the order it receives - first come first print. This allows users to work on other tasks after sending the printing jobs to the queue.



Queue ADT:

- A queue is a data structure which consists of linear collection of items in which access is restricted to a first-in first-out basis.
- New items are inserted at the back and existing items are removed from the front.
- The items are maintained in the order in which they are added to the structure.
 - ✓ Queue(): Creates a new empty queue, which is a queue containing no items.
 - ✓ enqueue(): Adds the given item to the back of the queue.
 - ✓ dequeue(): Removes the front item from the queue. An item cannot be dequeued from an empty queue.
 - ✓ isEmpty(): Returns a Boolean value indicating whether the queue is empty.
 - ✓ length (): Returns the number of items currently in the queue.

Implementing the Queue and its operations using Python List:

Let us implement a very simple queue using Python's list class. The operations that must be performed on the queue are encapsulated in the Queue class:

```
class Queue:  
    def __init__(self):  
        self.items = []
```

In the initialization method `__init__`, the instance variable “**items**” is set to `[]`, which means the queue is empty when created.

Enqueue operation:

- The enqueue operation is used to add items to the queue.
- In queue, elements are always inserted from **rear** end.
- The enqueue operation or method uses the **append** method of the list class to insert items (or data) at the rear of the list:
- The python code of enqueue operation is given below.

```
def enqueue(self,item):  
    self.items.append(item)
```

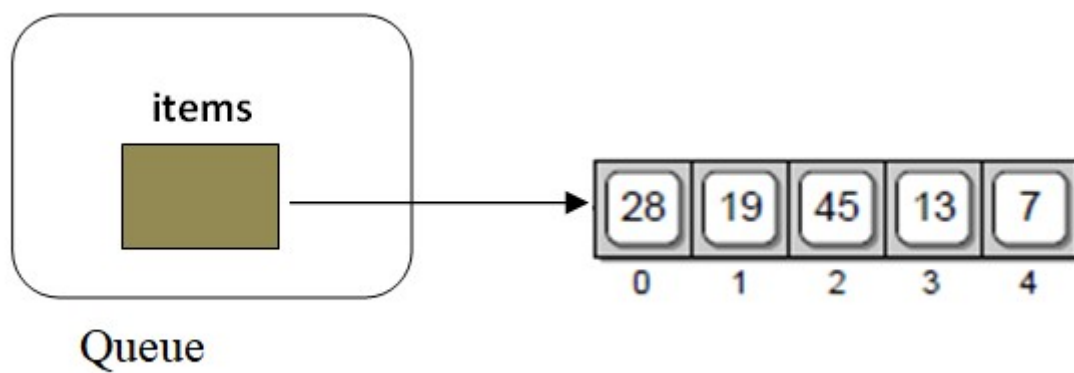
The enqueue operation can be visualized as shown in the diagram below.

```
Q = Queue()  
Q.enqueue(28)  
Q.enqueue(19)  
Q.enqueue(45)  
Q.enqueue(13)  
Q.enqueue(7)
```



Fig. Abstract view of the queue after the above enqueue operations.

We first create the Queue object and enqueue 5 values into the queue in the order as shown in queue.



The above figure shows the abstract view of the object or instance of Queue ADT using python list.

Deque operation:

- The dequeue operation is used to remove items from the queue.
- In queue items are always deleted from the **front** end.
- The Python list class has a method called **pop(index)** which is used to remove item from the queue.
- We will use **pop(0)** to delete the front item from the list.
- The item in the list is popped and saved in the data variable.
- The python code of dequeue operation is given below.

```
def dequeue(self):
    if self.isEmpty():
        print("Queue is Empty cannot delete")
    else:
        item=self.items.pop(0)
        print("Deleted Item is:",item)
```

isEmpty() operation :

- This isEmpty() is a utility function which returns Boolean value **true** if list containing queue elements is empty and **false** if list has some elements.
- This function is used by other functions for example: dequeue operation will first make sure whether the queue is not empty before removing items from the queue.
- The python code of this operation is given below

```
def isEmpty(self):
    return len(self.items) == 0
```

The below diagram shows the enqueue and dequeue operations on the queue.

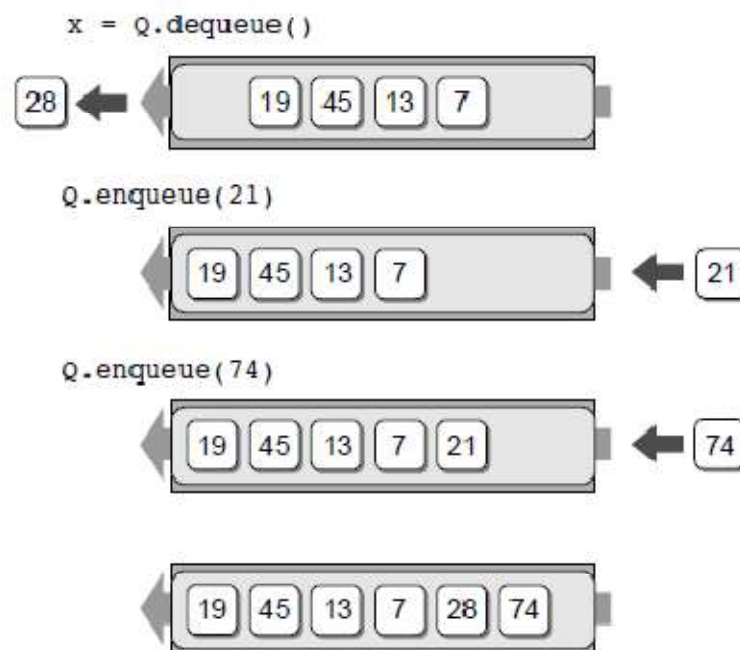


Fig. Abstract view of the queue after performing the additional operations.

Complete Implementation of Queue data structure:

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self,item):
        self.items.append(item)
    def dequeue(self):
        if self.isEmpty():
            print("Queue is Empty cannot delete")
        else:
            item=self.items.pop(0)
            print("Deleted Item is:",item)
    def display(self):
        if self.isEmpty():
            print("Queue is Empty")
        else:
            print(self.items)
    def isEmpty(self):
        return len(self.items) == 0
    def length(self):
        return len(self.items)

q = Queue()
while True:
    print("1:Enqueue 2:Dequeue 3:Display 4:Length 5:Exit")
    choice = int(input("Enter your choice:"))
    if choice==1:
        item=input("Enter the element:")
        q.enqueue(item)
    elif choice==2:
        q.dequeue()
    elif choice==3:
        q.display()
    elif choice==4:
        n = q.length()
        print("Length of the queue is ",n)
    elif choice==5:
        break
```

Priority Queues:

- A priority queue is a queue in which each item is assigned a priority and items with a higher priority are dequeued (removed) before those with a lower priority, irrespective of when they were added. However all the items with the same priority still obey FIFO principle.
- Integer values are used for the priorities with a smaller integer value having a higher priority.

- There are two basic types of priority queues:
 1. Bounded priority queue.
 2. Unbounded priority queue.
- A **bounded priority queue** restricts the priorities to the integer values between zero and a predefined upper limit **N** whereas an unbounded priority queue places no limits on the range of priorities.

The priority Queue ADT:

- ✓ `PriorityQueue()`: Creates a new empty unbounded priority queue.
- ✓ `isEmpty()`: Returns a boolean value indicating whether the queue is empty.
- ✓ `length ()`: Returns the number of items currently in the queue.
- ✓ `enqueue(item, priority)`: Adds the given item to the queue by inserting it in the proper position based on the given priority. The priority value must be within the legal range when using a bounded priority queue.
- ✓ `Dequeue()`: Removes and returns the item from the queue, which is the item with the highest priority. If two items have the same priority, then those items are removed in a FIFO order. An item cannot be deleted from an empty queue.

A note on priority queue item:

- When implementing the priority queue, the items cannot simply be added directly to the list, but instead we must have a way to associate a priority with each item.
- This can be done with a simple storage class containing two fields: one for the priority and one for the queue item as shown below.

class **PriorityQueueEntry**:

```
def __init__(self,value,priority):  
    self.v = value  
    self.p = priority
```

How to organize items in the priority queue?

We can consider two approaches, both of which satisfy the requirements of the priority queue:

1. Append new items to the end of the list.

- When a new item is enqueued, simply append a new instance of the storage class (containing the item and its priority) to the end of the list.
- When an item is dequeued, search the list for the item with the highest priority and remove it from the list.
- If more than one item has the same priority, the first one encountered during the search will be the first to be dequeued.

2. Keep the items sorted within the list based on their priority.

- When a new item is enqueued, find its proper position within the list based on its priority and insert an instance of the storage class at that point.
- If we order the items in the list from lowest priority at the front to highest at the end, then the dequeue operation simply requires the removal of the last item in the list.
- To maintain the proper ordering of items with equal priority, the enqueue operation must ensure newer items are inserted closer to the front of the list than the other items with the same priority.

- We have used first approach in our implementation below.

Enqueue Operation:

- We are using the first approach for enqueue operation.
- In this approach, we simply add the items into the priority queue using the **append** function on the items list.
- We first create a **PriorityQueueEntry** object and assign it with value and the priority fields and then **append** the object to list.
- The python code of enqueue operation is given below

```
def enqueue(self,value,priority):
    item = PriorityQueueEntry(value,priority)
    self.items.append(item)
```

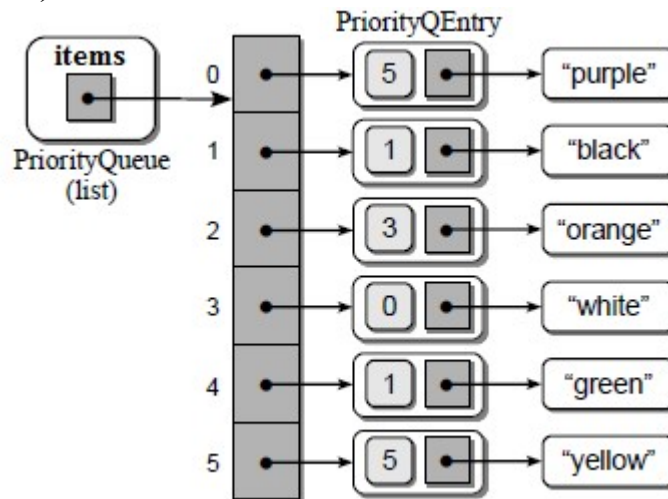


Fig. Abstract view of the priority queue object.

The above diagram shows the visualization of the priority queue after enqueue of 5 queue items. You can see that the items are simply appended as they are inserted.

Dequeue operation:

- The dequeue operation will remove the item with highest priority irrespective of when it was inserted into the queue.
- We will search for the item with highest priority in the queue and find its index and then use pop method provide with index of the highest priority item to delete the item from the queue.
- In our implementation we assume item at first index as highest priority at first and then loop through all items to find index of highest priority.
- **Note:** The highest priority item is one with lowest integer. In the above diagram white is highest priority item since it has 0 as priority value.
- The python code of dequeue operation is given below

```
def dequeue(self):
    if self.isEmpty():
        print("Queue is empty cannot delete")
    else:
```

```
highest = self.items[0].p
index = 0
for i in range(0,self.length()):
    if highest > self.items[i].p:
        highest = self.items[i].p
        index = i
del_item = self.items.pop(index)
print("deleted item is ",del_item.v)
```

isEmpty() operation:

- This isEmpty() is a utility function which returns Boolean value **true** if list containing queue elements is empty and false if list has some elements.
- This function is used by other functions for example: dequeue operation will first make sure whether the queue is not empty before removing items from the queue.
- The python code of this operation is given below

```
def isEmpty(self):
    return len(self.items) == 0
```

Implementation of priority queue data structure:

```
class PriorityQueueEntry:
    def __init__(self,value,priority):
        self.v = value
        self.p = priority
class PriorityQueue:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return len(self.items) == 0
    def length(self):
        return len(self.items)
    def enqueue(self,value,priority):
        item = PriorityQueueEntry(value,priority)
        self.items.append(item)
    def dequeue(self):
        if self.isEmpty():
            print("Queue is empty cannot delete")
        else:
            highest = self.items[0].p
            index = 0
            for i in range(0,self.length()):
                if highest > self.items[i].p:
                    highest = self.items[i].p
                    index = i
```

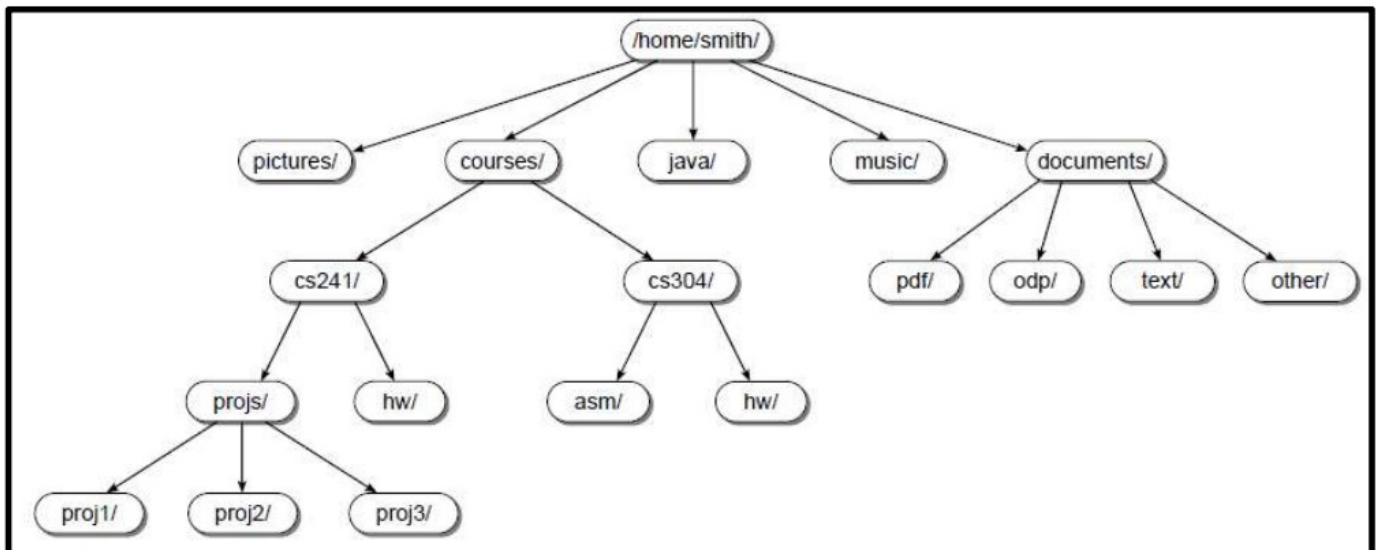
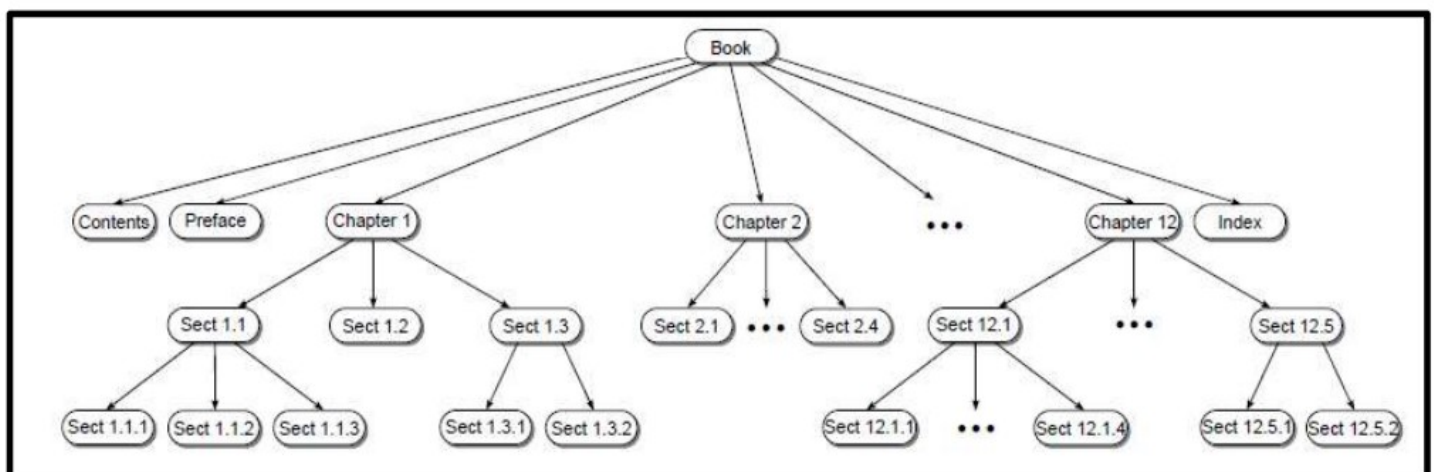
```
        del_item = self.items.pop(index)
        print("deleted item is ",del_item.v)
def display(self):
    if self.isEmpty():
        print("Queue is empty")
    else:
        for x in range(0,self.length()):
            print(self.items[x].v,":",self.items[x].p)
pq = PriorityQueue()
while(True):
    print("1:Enqueue 2:Dequeue 3:Display 4:Length 5:Exit")
    choice = int(input("Enter your choice:"))
    if choice == 1:
        value = input("enter the item to insert:")
        priority = int(input("Enter the priority:"))
        pq.enqueue(value,priority)
    if choice == 2:
        pq.dequeue()
    if choice == 3:
        pq.display()
    if choice == 4:
        n = pq.length()
        print("length of queue is:",n)
    if choice == 5:
        break
```

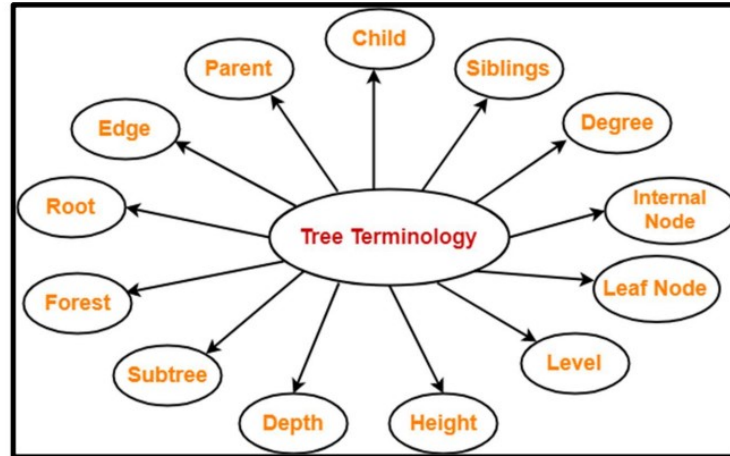
Week-11

The Tree data structure – Example: File explorer/Folder structure, Domain name server. Tree Terminologies, Tree node representation. Binary trees, Binary search trees, Properties, Implementation of tree operations – insertion, deletion, search, Tree traversals (in order, pre order and post order).

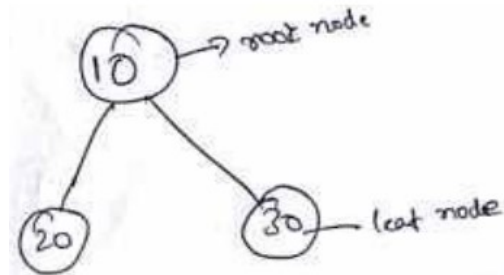
The Tree Data structure:

- A tree is a connected acyclic graph.
- A tree is non linear data structure in which all the values are arranged in a non linear form.
- A tree is a data structure which is a collection of zero or more nodes connected by edges.
- A Tree Data Structure consists of nodes and edges that organize group of data in a hierarchical fashion.
- The data elements are stored in nodes and pairs of nodes are connected by edges.
- A classic example of a tree structure is:
 - ✓ File/Folder Explorer: The representation of files, directories and subdirectories in a file system
 - ✓ Organization web pages in website
 - ✓ Content/Index page of text book

File/Folder Explorer:**Content/Index page of text book:**

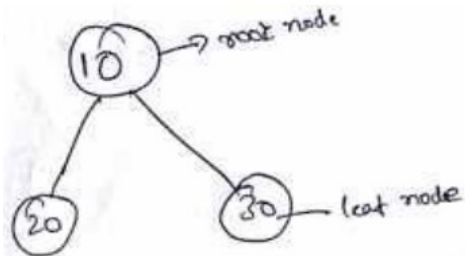
Tree Terminologies:

Root node: The topmost (first) node in a tree is called as root node.



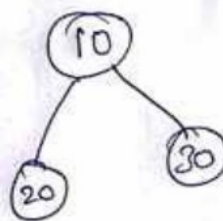
Ex: - 10 is the root node

Leaf node: A node which does not have any child node is called as leaf node.



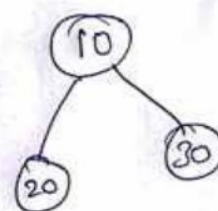
Ex: - 20 and 30 are leaf nodes.

Child node: The node obtained from the parent node is called as child node.



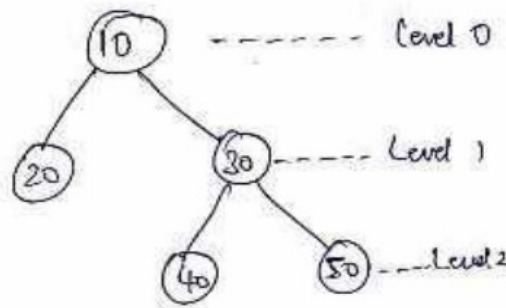
Ex: - 20 and 30 are children of 10

Parent node: The node having further sub branches is called as the parent node.



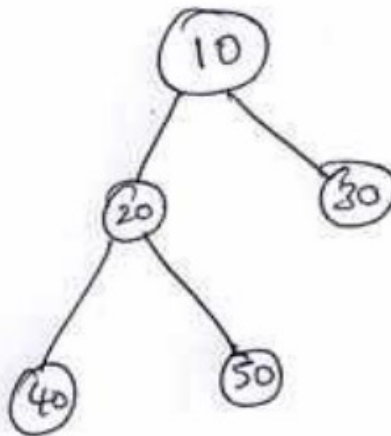
Ex: - 10 is the parent of 20 and 30

Level of tree: The distance of a node from the root node is called as level of a node.



Ex: - In above tree level is 2.

Internal node: The non leaf nodes in a tree are called as internal node.



Ex: - In above tree 10, 20 are the internal nodes

External node: The leaf nodes in a tree are called as external node.

Ex: - In above tree 40, 50 and 30 are the external nodes

Sibling: Two or more nodes having the same parent are called siblings.

Ex: - 40 and 50 are siblings

Ex: - 20 and 30 are siblings

Degree of a node: The number of branches associated with each node is called degree of a node.

Ex: - degree of 20 is 2

Ex: - degree of 40 is 0

Degree of a tree: The maximum degree in the tree is the degree of a tree.

Ex: - Degree of an above tree is 2.

Depth: The height/depth of a tree is defined as the maximum level of any leaf node in the tree is called as depth of a tree. Ex: - In above tree depth is 3

Path: The sequence of nodes and edges from one node to another node is called as path.

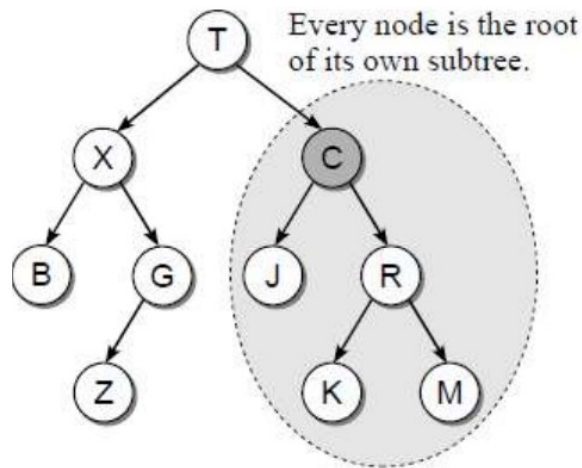
Ex: - The path from 10 to 40 is 10->20->40

Ancestors: - The nodes in the path above the child are called ancestors.

Descendents: - The nodes in the path below parent are called descendents.

Subtree: Every node can be the root of its own subtree, which consists of a subset of nodes and edges of the larger tree.

Ex:



A subtree with root node C.

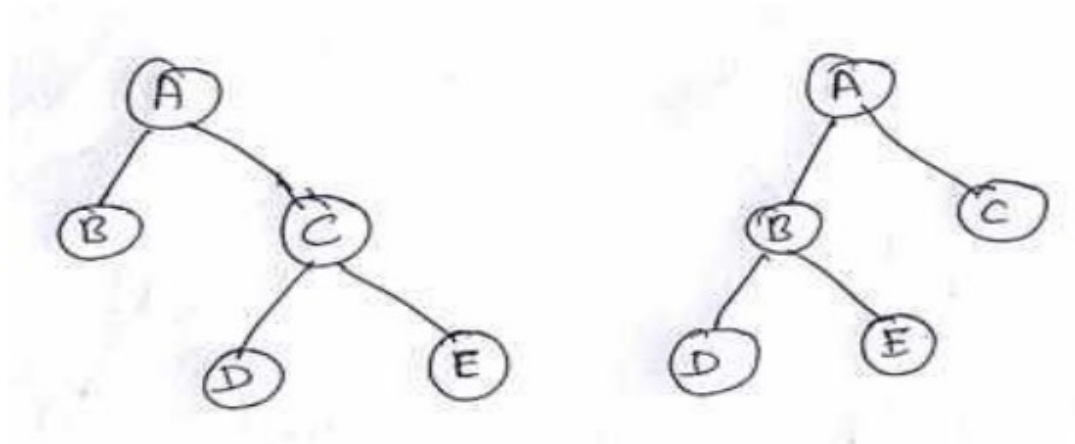
Applications of trees:

- ✓ Searching and Sorting (Binary Search Tree)
- ✓ Manipulation of arithmetic expressions.
- ✓ Symbol table construction
- ✓ Trees are used in syntax analysis of compiler design.
- ✓ Heap trees
- ✓ Router algorithms
- ✓ Huffman coding
- ✓ To solve database problems such as indexing

Binary tree: Binary tree is a tree in which every node can have maximum of two children i.e. each node can have zero children, one child or two children, but not more than two children.

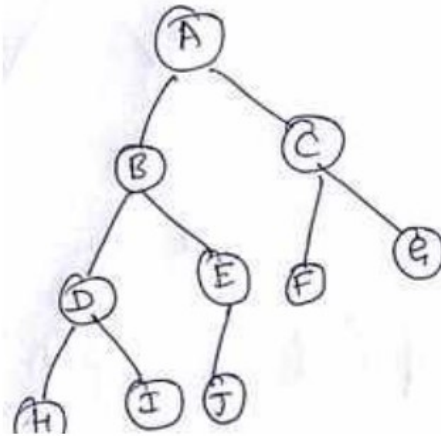
Strictly binary tree:

- A binary tree in which every node can have either 2 children or 0 children is called strictly binary tree.
- It is also called as full binary tree or proper binary tree.
- It is used to represent mathematical expressions.



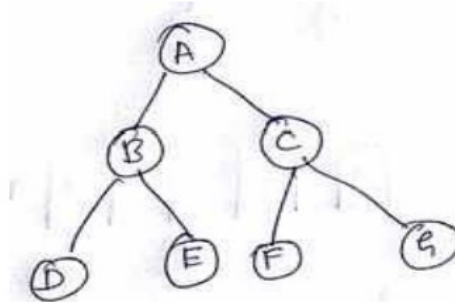
Complete binary tree: A complete binary tree is a tree in which every level except the last level is completely filled and in last level all nodes are left aligned.

Ex:



Perfect binary tree: A binary tree in which every internal node must have exactly two children and all leaf nodes at the same level is called perfect binary tree.

Ex:

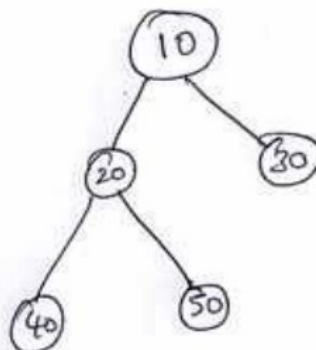


Tree node representation:

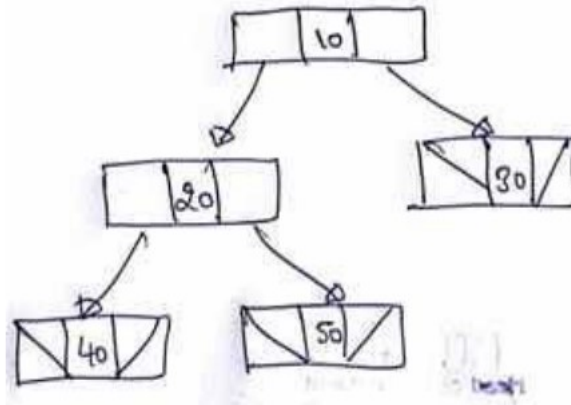
- A linked representation of each node in a binary tree has three fields left, data, right
- The node of a binary tree is represented as follows:



- ✓ The left contains the address of left sub tree.
- ✓ The right contains the address of right sub tree.
- ✓ If any sub tree is empty corresponding pointer stores a **None** value.
- Consider the following binary tree:



- The linked representation of above binary tree is given below:

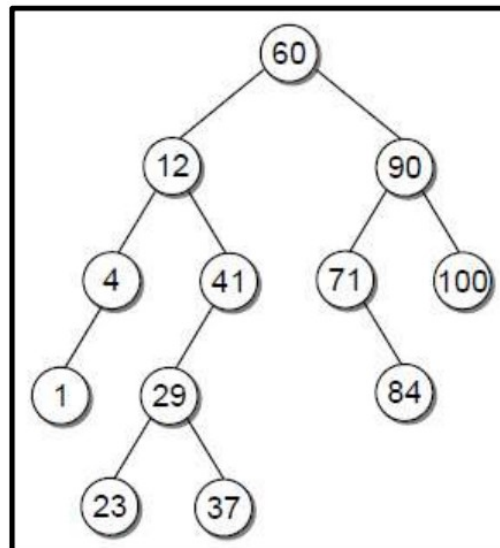


- The python code to create a node is given below.

```
class Node:
    def __init__(self,value):
        self.left = None
        self.data = value
        self.right =None
```

Binary search tree:-

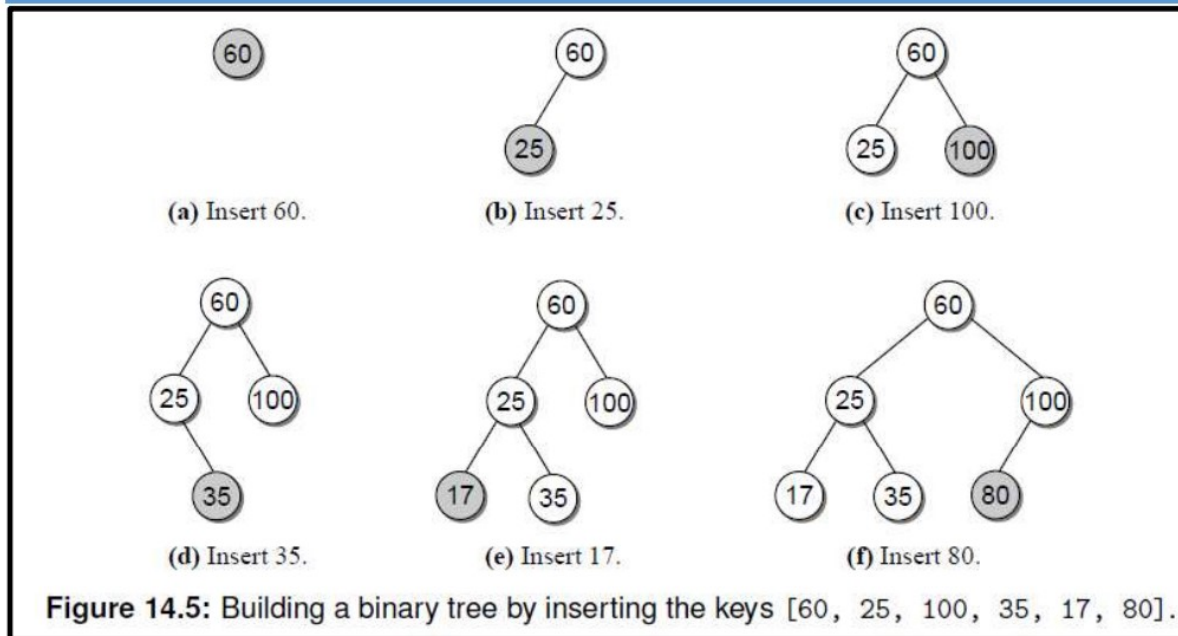
- A binary search tree (BST) is a binary tree in which each node contains a value and it should have the following properties:
 - ✓ Values of Nodes of left subtree must be smaller
 - ✓ Values of Nodes of right subtree must be greater than or equal.
- A binary search tree is a binary tree in which for each node say x in the tree, elements in the left subtree are less than x and elements in the right subtree are greater than or equal to x.
- Consider the binary search tree in Figure, which contains integer values. The root node contains value 60 and all values in the root's left subtree are less than 60 and all of the values in the right subtree are greater than 60.



Insert Operation:

- When we create Binary Search Tree, nodes are inserted one at a time by creating new node. Suppose we want to build a binary search tree from the value list [60, 25, 100, 35, 17, 80] by inserting the keys in the order they are listed.

- We start by inserting value 60. A node is created and its data field set to that value. Since the tree is initially empty, this first node becomes the root of the tree (part a).
- Next, we insert value 25. Since it is smaller than 60, it has to be inserted to the left of the root, which means it becomes the left child of the root (part b).
- Value 100 is then inserted in a node linked as the right child of the root since it is larger than 60 (part c).
- Similarly, we insert 35, 17 and 80 (part d, e f).



Tree traversals (in order, pre order and post order):

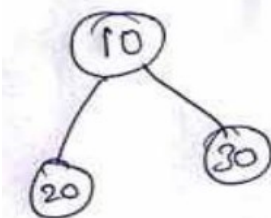
- Tree Traversal is defined as “**Visiting each and every node of a tree in some predefined order**”.
- Traversing a binary tree can be defined as a procedure with which each node in a given tree is visited exactly once in some predetermined order.
- There are 3 ways of a traversing a binary tree
 1. Preorder (NLR)
 2. Postorder (LRN)
 3. Inorder (LNR)

Preorder (NLR):

- In this node is visited before the subtrees

Algorithm: - It can be recursively defined as follows

- If the tree is not empty
 - Step 1: visit the root node
 - Step 2: Traverse the left sub tree in preorder
 - Step 3: Traverse the right sub tree in preorder.
- Consider the following example:

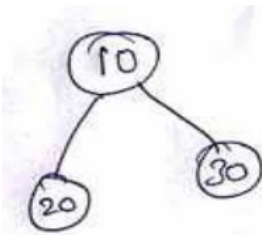


- The preorder traversal of the above tree is 10, 20, 30
- The python code of preorder traversal is given below:

```
def preorder(self, rt):  
    print(rt.data, end=" ")  
    if rt.left is not None:  
        self.preorder(rt.left)  
    if rt.right is not None:  
        self.preorder(rt.right)
```

Postorder (LRN):

- In this node is visited after the subtrees
- Algorithm:** - It can be recursively defined as follows
- If the tree is not empty
Step 1: Traverse the left sub tree in postorder
Step 2: Traverse the right sub tree in postorder.
Step 3: visit the root node.
 - Consider the following example:

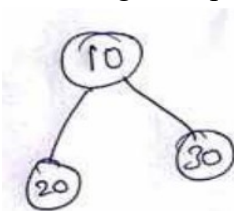


- The postorder traversal of the above tree is 20, 30, 10
- The python code of postorder traversal is given below:

```
def postorder(self, rt):  
    if rt.left is not None:  
        self.postorder(rt.left)  
    if rt.right is not None:  
        self.postorder(rt.right)  
    print(rt.data, end=" ")
```

Inorder (LRN):

- In this node is visited between the subtrees
- Algorithm:** - It can be recursively defined as follows
- If the tree is not empty
Step 1: Traverse the left sub tree in inorder
Step 2: visit the root node.
Step 3: Traverse the right sub tree in inorder.
 - Consider the following example:



- The inorder traversal of the above tree is 20, 10, 30
- The python code of inorder traversal is given below:

```
def inorder(self, rt):  
    if rt.left is not None:  
        self.inorder(rt.left)  
    print(rt.data, end=" ")  
    if rt.right is not None:  
        self.inorder(rt.right)
```

Implement Binary Search Tree:

class Node:

```
    def __init__(self,value):  
        self.data = value  
        self.left = None  
        self.right =None
```

class BinarySearchTree:

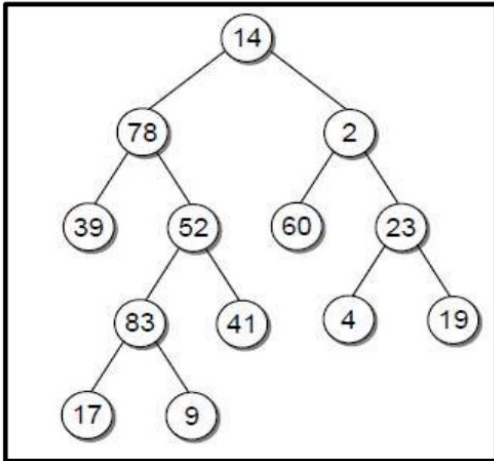
```
    def __init__(self):  
        self.root=None  
    def insert(self,value):  
        newNode=Node(value)  
        if self.root is None:  
            self.root = newNode  
        else:  
            curNode = self.root  
            while curNode is not None:  
                if value < curNode.data:  
                    if curNode.left is None:  
                        curNode.left=newNode  
                        break  
                    else:  
                        curNode = curNode.left  
                else:  
                    if curNode.right is None:  
                        curNode.right=newNode  
                        break  
                    else:  
                        curNode=curNode.right
```

```
    def preorder(self, rt):  
        print(rt.data, end=" ")  
        if rt.left is not None:  
            self.preorder(rt.left)  
        if rt.right is not None:  
            self.preorder(rt.right)  
    def postorder(self, rt):
```

```
if rt.left is not None:
    self.postorder(rt.left)
if rt.right is not None:
    self.postorder(rt.right)
print(rt.data, end=" ")
def inorder(self, rt):
    if rt.left is not None:
        self.inorder(rt.left)
    print(rt.data, end=" ")
    if rt.right is not None:
        self.inorder(rt.right)
bst = BinarySearchTree()
ls = [25,10,35,20,65,45,24]
for i in ls:
    bst.insert(i)
print("\nPre-order")
bst.preorder(bst.root)
print("\nPost-order")
bst.postorder(bst.root)
print("\nIn-order")
bst.inorder(bst.root)
```

Assignment Questions:

1. Construct the binary search tree for the following values and traverse the tree in preorder, postorder, inorder? 46, 76, 36, 26, 16, 56, 96.
2. Construct the binary search tree for the following values and traverse the tree in preorder, postorder, inorder? 30 63 2 89 16 24 19 52 27 9 4 45
3. Construct the binary search tree for the following values and traverse the tree in preorder, postorder, inorder? 25,10,35,20,65,45,24
4. Sort the following list using Binary Search Tree:
 - a) 10, 8, 11, 4, 9, 15, 7, 14
 - b) 10, 8, 9, 15, 7, 14
5. Consider the following binary tree:



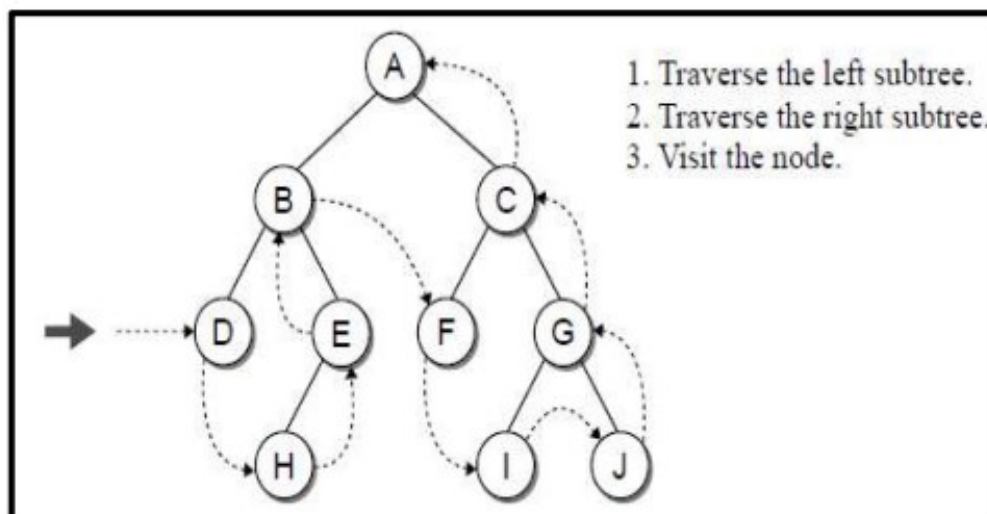
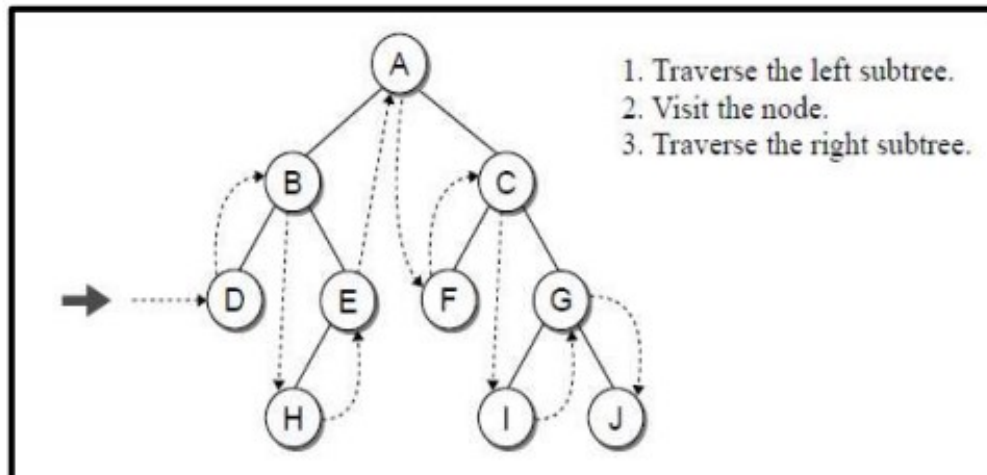
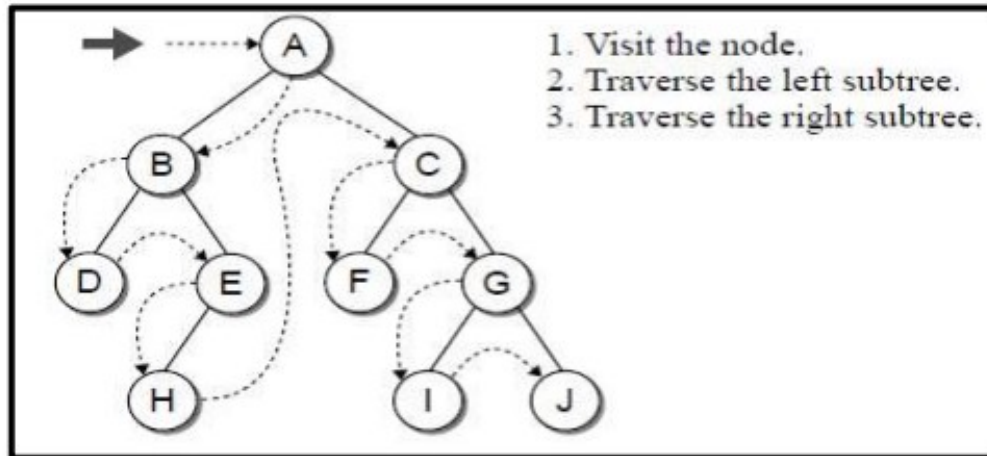
- (a) Show the order the nodes will be visited in a: i. preorder traversal ii. inorder traversal iii. postorder traversal
- (b) Identify all of the leaf nodes.
- (c) Identify all of the interior nodes.
- (d) List all of the nodes on level 4.
- (e) List all of the nodes in the path to each of the following nodes: i) 83 ii) 39 iii) 4 iv) 9
- (f) Consider node 52 and list the node's: i) Descendants ii) Ancestors iii) Siblings
- (g) Identify the depth of each of the following nodes: i) 78 ii) 41 iii) 60 iv) 19

Week-12

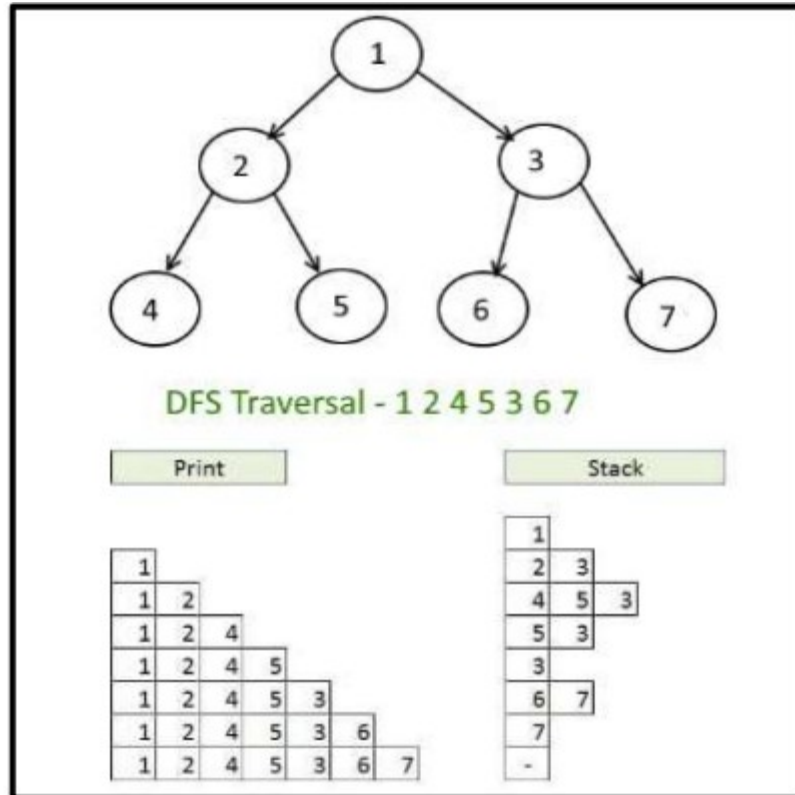
Depth-first traversal, Breadth-first traversal, Tree applications: Expression evaluation.

Depth-First Traversal:

- The pre-order, in-order, and post-order traversals are all examples of a Depth First Traversal. That is, the nodes are traversed deeper in the tree before returning to higher-level nodes.
- Figure shows the ordering of the nodes in a Depth First Traversal of a following Binary Tree.



- Stack data structure is used to implement the Depth First Traversal.
- First add the root to the Stack.
- Pop out an element from Stack, visit it and add its right and left children to stack.
- Pop out an element, visit it and add its children.
- Repeat the above two steps until the Stack is empty.
- Example:

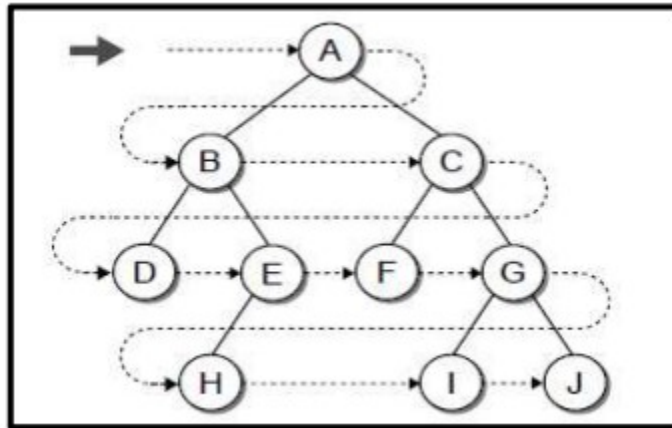


- The python code implementation of DFS is given below

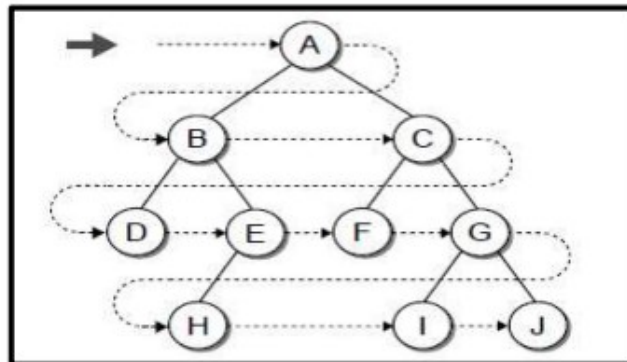
```
def DFS(root):
    s = Stack()
    s.push(root)
    while s.isEmpty() != True:
        node=s.pop()
        print(node.data,end="\t")
        if node.right is not None:
            s.push(node.right)
        if node.left is not None:
            s.push(node.left)
```

Breadth-First Traversal:

- Another type of traversal that can be performed on a binary tree is the Breadth First Traversal.
- In a Breadth First Traversal, the nodes are visited by level by level, from left to right.
- Figure shows the ordering of the nodes in a Breadth First Traversal of a following Binary Tree.



1. Queue data structure is used to implement the Breadth First Traversal.
2. The process starts by adding root node to the queue.
3. Next, we visit the node at the front end of the queue, remove it and insert all its child nodes.
4. This process is repeated till queue becomes empty.
5. Example:



Queue

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

BFS Traversal: A B C D E F G H I J

➤ The python code implementation of BFS is given below

```

def BFS(root):
    q = Queue()
    q.enqueue(root)
    while q.isEmpty() != True:
        node=q.dequeue()
        print(node.data,end="\t")
  
```

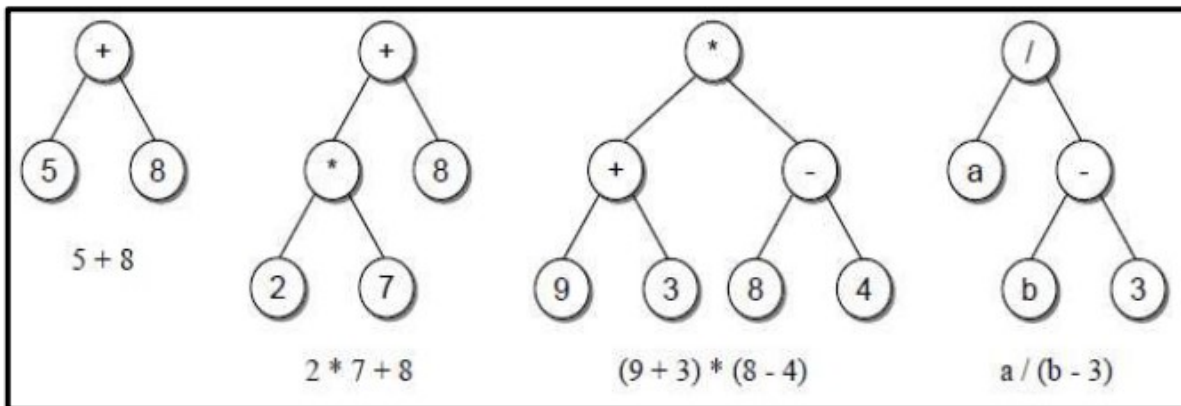
```

if node.left is not None:
    q.enqueue(node.left)
if node.right is not None:
    q.enqueue(node.right)

```

Expression Trees:

- An Arithmetic expressions such as $(9+3)*(8-4)$ can be represented using an expression tree.
- An expression tree is a binary tree in which the operators are stored in the interior nodes and the operands (the variables or constant values) are stored in the leaves.
- Once constructed, an expression tree can be used to evaluate the expression or for converting an infix expression to either prefix or postfix notation.
- Sample Arithmetic Expression Trees are given below:



Construction of Expression Trees:

- To construct Expression Tree, for simplicity, we assume the following:
 1. The expression is stored in string with no white space;
 2. The supplied expression is valid and fully parenthesized;
 3. Each operand will be a single-digit or single-letter variable;
 4. The operators will consist of +, -, *, /, and %.

Build the tree for the expression $((2*7)+8)$. The steps illustrated in the figure are described below:

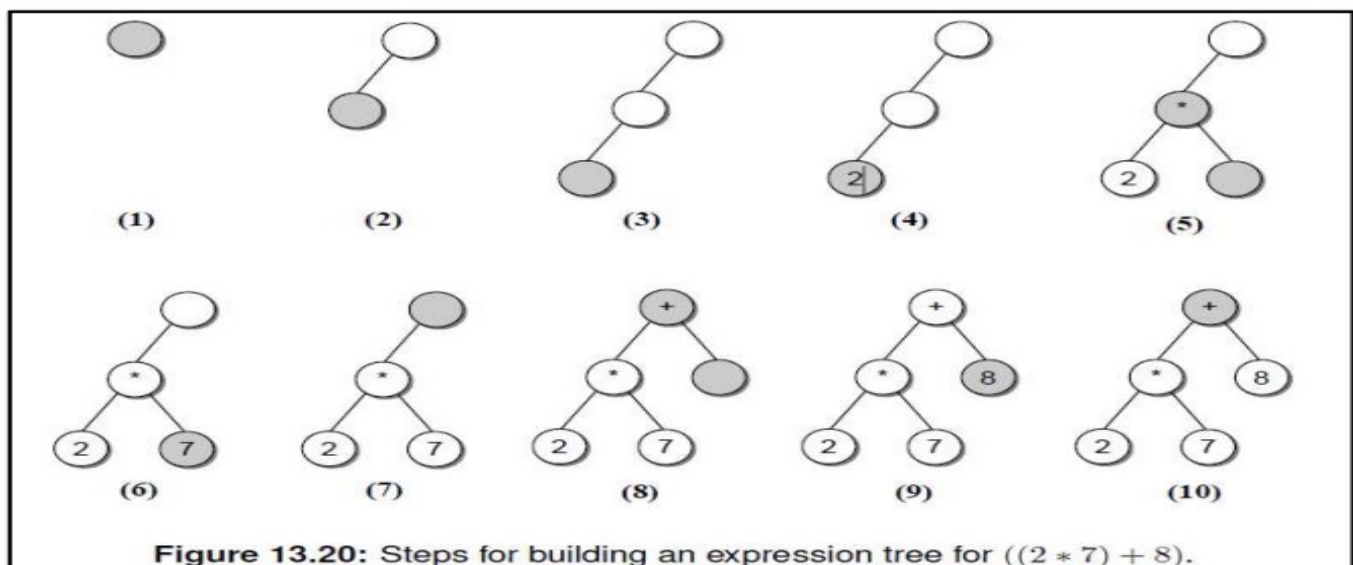
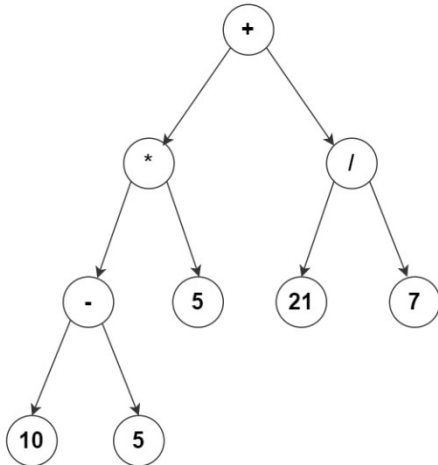
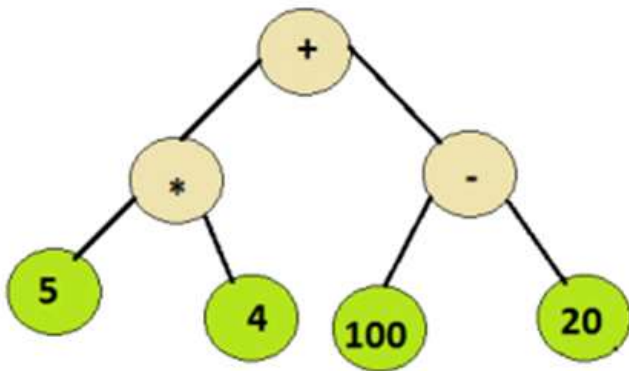


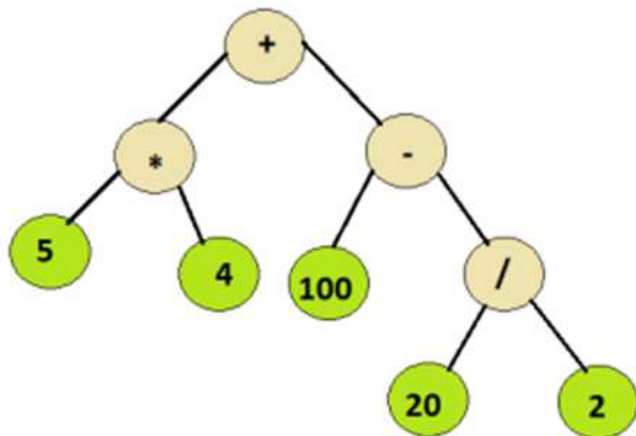
Figure 13.20: Steps for building an expression tree for $((2 * 7) + 8)$.

Expression Evaluation:

The value of the above expression tree is 28.0



The value of the above expression tree is 100



The value of the above expression tree is 110

Implementation of Depth First Traversal:

```
class Stack:
```

```
    def __init__(self):
```

```
        self.items=[]
```

```
    def push(self,value):
```

```
        self.items.append(value)
```

```
def pop(self):
    if len(self.items) != 0:
        return self.items.pop()
def isEmpty(self):
    return len(self.items) == 0
class Node:
    def __init__(self,value):
        self.data = value
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None
    def insert(self,value):
        newNode=Node(value)
        if self.root is None:
            self.root = newNode
        else:
            curNode = self.root
            while curNode is not None:
                if value < curNode.data:
                    if curNode.left is None:
                        curNode.left=newNode
                        break
                    else:
                        curNode = curNode.left
                else:
                    if curNode.right is None:
                        curNode.right=newNode
                        break
                    else:
                        curNode=curNode.right
def DFS(root):
    s = Stack()
    s.push(root)
    while s.isEmpty() != True:
        node=s.pop()
        print(node.data,end=" ")
        if node.right is not None:
            s.push(node.right)
        if node.left is not None:
            s.push(node.left)
bst = BinarySearchTree()
```

```
ls = [25,10,35,20,5,30,40]
for i in ls:
    bst.insert(i)
print("Depth First Search Traversal:")
DFS(bst.root)
```

Implementation of Breadth First Traversal:

```
class Queue:
    def __init__(self):
        self.items=[]
    def enqueue(self,value):
        self.items.append(value)
    def dequeue(self):
        if len(self.items) != 0:
            return self.items.pop(0)
    def isEmpty(self):
        return len(self.items) == 0
class Node:
    def __init__(self,value):
        self.data = value
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None
    def insert(self,value):
        newNode=Node(value)
        if self.root is None:
            self.root = newNode
        else:
            curNode = self.root
            while curNode is not None:
                if value < curNode.data:
                    if curNode.left is None:
                        curNode.left=newNode
                        break
                else:
                    curNode = curNode.left
            else:
                if curNode.right is None:
                    curNode.right=newNode
                    break
```

```

        else:
            curNode=curNode.right
def BFS(root):
    q = Queue()
    q.enqueue(root)
    while q.isEmpty() != True:
        node=q.dequeue()
        print(node.data,end=" ")
        if node.left is not None:
            q.enqueue(node.left)
        if node.right is not None:
            q.enqueue(node.right)
bst = BinarySearchTree()
ls = [25,10,35,20,5,30,40]
for i in ls:
    bst.insert(i)
print("Breadth First Search Traversal:")
BFS(bst.root)

```

Assignment Questions:

- 1) For the following trees write the order the nodes will be visited in: i. Depth First Traversal ii. Breadth First Traversal

