# Design database using simple data structure. Algorithm:

Step1: Start (create a class named student)

Step2: the method of the class,getstudentdetails()gets input from the user print result() calculates result and prints it.

Step3: we will also add extra marks(9) to a student Step4: then print the result

Step5: Stop

# Input:

class Student:

def getStudentDetails(self,i):

self.rollno=input("Enter Roll Number: ") Self.name = input("Enter Name: ")

self,physics =int(input("Enter Physics Marks: "))

self.chemistry = int(input("Enter Chemistry Marks: ")) self.maths = int(input("Enter Math Marks: "))

def display(self,i):

print("RollNo: ", self.rollno) print("Name: ",self.name) print("Score1: ", self.physics) print("Score2: ", self.chemistry) print("Score3: ",self.maths) print("n")

def printResult(self):

self.percentage = (int)( (self.physics+ self.chemistry + self.maths)/300 \* 100); print(self.rollno,self.name, self.percentage)

n=intinput("Enter how many students records")) for i in range(n):

S1-Student() 51.getStudentDetails(i)

print("\nList of Students\n") for i in range(n):

$1.displayli) print("Result: ") S1.printResult()

S1.physics +9

print("result after adding grace marks...") S1.printResult()

# Output:

Enter how many students records 3

Enter Roll Number: 1 Enter Name: a

Enter Physics Marks : 45 Enter Chemistry Marks : 67 Enter Math Marks : 78 Enter Roll Number: 2

Enter Name : b

Enter Physics Marks : 56 Enter Chemistry Marks : 78 Enter Math Marks : 89 Enter Roll Number: 3

Enter Name:c

Enter Physics Marks : 67 Enter Chemistry Marks : 78 Enter Math Marks : 89

List of Students RollNo: 3 Name : c

Score1 : 67

Score2 : 78

Score3: 89

RollNo: 3 Name: c Scorel: 67

Score2 : 78

Score3: 89

RollNo: 3 Name: c Score1: 67 Result:

3 c 78

result after adding grace marks... 3 c 81

# Working:

In this problem we need to get input from users for the rollno,name and marks of three subjects.then we need to calculate the percentage and print result, and reprint it after giving extra marks.

# Implementation of student database using Abstract data types (ADT) Algorithm:

Step 1 :- Start

Step2 :- create a class named Student

Step3. - using init fuctions and self argument give the detail name, rollno, m1, m2 marks. create the functions and view them to give values.

Step4 :- print every junction name and give the values.

Step5: perform accept, delete Update, exit operations. Step6:Stop

# Input:

class student:

def init\_(self, name, rollno, m1, m2):

selt.name=name self.rollno=rollno self.m1=m1

self.m2=m2

def accept(self, name, rollno, marks1, marks2): ob=student(name, rolino, marks1, marks2)) Is.append(ob)

def display(self, ob): print("Name", ob.name) print("Rollno ", ob.rollno) print("Marks1:", ob.m1)

print("Marks2:", ob.m2) print("\n")

def search(self, rn):

for i in range(ls.\_len\_0): if(ls[i].rolino==rn):

return i

def delete(self, rn): i=obj.search (rn) del Is[i]

def update(self, rn, no): i=obj.search(rn)

roll=no Is[i].rollno=roll;

Is=1

obj=student(''0,0,0) print("nOperations used, ")

print("n1.accept student details\n2.display student details \n" "3.search details of a student\n4.delete details of student" "\n5.update student details\n6.exit")

print("1->ACCEPT")

obj.accept("A", 1, 100, 100)

obj.accept("B", 2, 90, 90)

obj.accept("C", 3, 80, 80) print("press any key to continue.." input()

1.acc. 2.dis 3.S

print("2->DISPLAY")

print("press any key to continue...") input()

print("\n")

print("\nlist of student\n") for i in range(ls.\_len\_0): obj.display(ls[i])

print("3->SEARCH")

print("press any key to continue... ") input()

print("\n student found,") s=obj.search(2) obj.display(ls[s])

print("4->DELETE")

print("press any key to continue... ") input()

obj.delete(2) print(ls\_len\_())

print("list after deletion") for i in range(ls.len\_(): obj.display(ls[i])

print("5->UPDATE")

print("press any key to continue... ") input()

obj.update(3,2) print(ls\_en\_(0)

print("list after updation") for i in range(ls. len\_(): obj.display(s[il)

print("6->EXIT")

print("press any key to continue...") input()

print("Thank you !")

# Output:

Operations used,

[8:34 PM, 6/12/2023] Smartestbadboy: 1.accept student details 2.display student details

3.search details of a student 4.delete details of student 5.update student details 6.exit

1-->ACCEPT

press any key to continue... 2-->DISPLAY

press any key to continue... list of student

Name : A Rollno : 1

Marks1 : 100

Marks2 : 100 Name : B Rollno :2 Marks1:90 Marks2 :90 Name : C Rollno :3 Marks1:80 Marks2: 80 3-->SEARCH

press any key to continue... press any key to continue... student found,

Name : B Rollno :2

Marks1 : 90

Marks2: 90 4-->DELETE

press any key to continue...

[8:36 PM, 6/12/2023] Smartestbadboy: 2 list after deletion

Name : A Rollno : 1

Marks1 : 100

Marks2 : 100 Name : C Rollno :3 Marks1:80 Marks2 : 80 5-->UPDATE

press any key to continue... 2

list after updation Name : A

Rollno : 1

Marks1 : 100

Marks2 : 100 Name : C Rollno : 2

Marks1 : 80

Marks2 : 80 6-->EXIT

press any key to continue... Thank you !

# Working:

**Operations used,**

1.Accept Student details 2.Display Student Details 3.Search Details of a Student

4.Delete Details of Student 5.Update Student Details 6.Exit

It gives the output of all used functions with students marks and the grace marks added.

# 3.write a program to perform push and pop Operations using list has a stack. Algorithm:

Step 1 :- Start

Step 2 :- Store the element to push into armay.

/Step 3 :- check if top = = (marsize - 1) then atale else goto step 4.

Skep4:- Increment top as top=top +1

Step5 :- Add element to the position stack[top]=num Step 6 :- Stop

# Input:

stack=[] def push():

item=input("enter the item to be pushed") stack.append(item)

print(stack) def pop():

if not stack: print("stack is empty") else:

e=stack.pop()

print("the element poped is =",e) true=1

while true:

print("1.push 2.pop 3.exit") ch=int(input("enter the choice")) if(ch==1):

push()

elif(ch==2): pop() elif(ch==3): break else:

print("enter the correct choice")

# Output:

1.push 2.pop 3.exit enter the choicel

enter the item to be pushed20 ['20']

1.push 2.pop 3.exit enter the choice1

enter the item to be pushed60 ['20', '60']

1.push 2.pop 3.exit enter the choice1

enter the item to be pushed45 ['20', '60', '45']

1.push 2.pop 3.exit enter the choice2

the element poped is = 45 1.push 2.pop 3.exit

enter the choice3

# Working:

In the above code, we have defined an empty list. We inserted the elements one by one using the append() method. That is similar to the push() method. We also removed the elements using the pop() method. The pop() method returns the last element of the list.

# 4.write a program to perform all operations on queue using list has a queue. Algorithm:

Step 1:- Start

Step 2 :- Implementing queue using list : q = [] q. append (10) Step 3 :- implement queue using list funitions) q = []

def enqueue (); if len (q) = -size + check wheather queue in full or not..

Step 4: implement queue using queue module. from queue Import queue of q=Queue (max size 4)......

Steps :- from collection Implement operations like pop() display () and exit ()

Step 6 – Stop

# Input:

queue=[] def push():

print("enter the element to push") item=input()

queue.append(item)

print(item, "placed in queue") def pop():

if not queue: print("queue is empty") else:

item=queue.pop()

print("the element poped is",item)

def display():

print(queue) true=1

while true:

print("the operations on queue are") print("1.push 2.pop 3.display ") ch=int(input("enter the choice")) if(ch==1):

push()

elif(ch==2):

pop()

elif(ch==3):

display() else:

print("enter correct choice")

# Output:

the operations on queue are 1.push 2.pop 3.display

enter the choice1

enter the element to push 20

20 placed in queue

the operations on queue are 1.push 2.pop 3.display

enter the choice1

enter the element to push 45

45 placed in queue

the operations on queue are 1.push 2.pop 3.display

enter the choice3 ['20', '45']

the operations on queue are 1.push 2.pop 3.display

enter the choice2

the element poped is 45

the operations on queue are 1.push 2.pop 3.display

# Working:

In the above, we have imported the queue module that is a LIFOqueue. It works the same as the stack but this module includes some additional functions mentioned above. We defined a stack with the maxsize that means it can hold maximum five values in it.

The initial array size is zero; we pushed three elements in the stack using the put() method.

Now, again we checked whether a stack is empty and size of the stack. We have three

elements in the stack. We popped the element using the get() method. It removes the last added element first. After removing the entire elements, we get an empty queue.

**Write a program to implement push and pop operations using list has a stack:-**

stack=[]

def push():

item=input("enter the item to be pushed:")

stack.append(item)

print(stack)

def pop():

if not stack:

print("stack is empty")

e=stack.pop()

print("the elements poped is:",e)

true -1

while(true):

print("1:push, 2:pop, 3:exit")

ch = int input("enter any choice as above:"))

if(ch==1):

push()

elif(ch==2):

pop()

elif(ch==3):

break

else:

print("enter proper choice:")

**Algorithm:-**

1. Start
2. Store the element to push into array
3. Check if top==(max size-1) then stack is full else go to step4
4. Increment top as top=top+1
5. Add element to the position stack[for]=num
6. Stop

**Output:-**

1:push, 2:pop, 3:exit

Enter any choice as above:1

Enter the item to be pushed:50

[‘50’]

1:push, 2:pop, 3:exit

Enter any choice as above:2

Enter the item to be poped:50

1:push, 2:pop, 3:exit

Enter any choice as above:5

Enter proper choice

**Working:-**

push(n)– This is a user-defined stack method used for inserting an element into the stack. The element to be pushed is passed in its argument. pop()– We need this method to remove the topmost element from the stack. isempty()– We need this method to check whether the stack is empty or not.

**Implement stack operations using ADT:-**

class stack:

def \_\_init\_\_(self):

self.item=[]

def isempty(self):

return self.items==[]

def push(self.item):

self.items.append(item)

def pop(self):

return self.items.pop()

def peep(self):

return self.items[len(self.items)-1]

def size(self):

return[len(self.items)]

def display(self):

print(self.items)

s=stack()

print("stack operations example")

print(s.isempty())

s.push(10)

s.push(20)

s.push(30)

s.display()

print(s.peep())

print(s.size())

print(s.pop())

print(s.pop())

**Algorithm:-**

1. Start
2. Create a class with name stack define a function with maximum arguments of self define functions for call operation with arguments
3. Create a stack
4. Give the values for created functions arguments
5. Stop

**Output:-**

Stack operation example:

True

[10,20,30]

30

[3]

30

20

Working:-

Python’s built-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of the stack while pop() removes the element in LIFO order.   
Unfortunately, the list has a few shortcomings. The biggest issue is that it can run into speed issues as it grows. The items in the list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

**Implement Queue operations using ADT:-**

class queue:

def \_\_init\_\_(self):

self.item=[]

def isempty(self):

return self.items==[]

def push(self.item):

self.items.append(item)

def pop(self):

return self.items.pop()

def peep(self):

return self.items[len(self.items)-1]

def size(self):

return[len(self.items)]

def display(self):

print(self.items)

q=queue()

print("Queue operations example")

print(s.isempty())

q.push(10)

q.push(20)

q.push(30)

q.display()

print(q.peep())

print(q.size())

print(q.pop())

print(q.pop())

**Algorithm:-**

1. Start
2. Create a class with name Queue define a function with maximum arguments of self define functions for call operation with arguments
3. Create a Queue
4. Give the values for created functions arguments
5. Stop

**Output:-**

Queue operation example:

True

[10,20,30]

30

[3]

30

20

**Working:-**

Python’s built-in data structure list can be used as a Queue. Instead of push(), append() is used to add elements to the top of the Queue while pop() removes the element in LIFO order.   
Unfortunately, the list has a few shortcomings. The biggest issue is that it can run into speed issues as it grows. The items in the list are stored next to each other in memory, if the Queue grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

**Implement bubble sort, selection sort and insertion sort and conclude which algorithm is best:-**

//**bubble sort**

import time

def bubblesort(arr):

n=len(arr)

swapped=False

for i in range(n-1):

for j in range(0,n-i-1):

if arr[j]>arr[j+1]:

swapped=True

arr[j], arr[j+1]=arr[j+1], arr[j]

if not swapped:

return

arr=[64,34,25,12,22,11,90]

st=time.time()

bubblesort(arr)

et=time.time()

etime=st-et

print("execution time=",etime)

print("sorted array is:")

for i in range(len(arr)):

print("%d"%arr[i],end=" ")

**Algorithm:-**

* traverse from left and compare adjacent elements and the higher one is placed at right side.
* In this way, the largest element is moved to the rightmost end at first.
* This process is then continued to find the second largest and place it and so on until the data is sorted.
* We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
* To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

**Output:-**

Execution time=0.0

Sorted array is:

11 12 22 25 34 64 90

**Working:-**

The main operation in Bubble sort is to compare two consecutive elements. If the first element is greater than the next element, then swap both, so that the smaller element comes ahead and the greater element goes back.In one iteration of outer loop, the greatest element of the list goes at the last index.

**//selection sort**

import time

import sys

a=[64,25,12,22,11]

st=time.time()

for i in range(0,len(a)):

min\_idx=i

for j in range(i+1,len(a)):

if a[min\_idx]>a[j]:

min\_idx=j

a[i],a[min\_idx]=a[min\_idx],a[i]

print("sorted array")

for i in range(len(a)):

print("%d"%a[i])

et=time.time()

etime=st-et

print("execution time=",etime)

**Algorithm:-**

1. Set the first element as minimum . Select first element as minimum.
2. Compare minimum with the second element.
3. After each iteration, minimum is placed in the front of the unsorted list.
4. For each iteration, indexing starts from the first unsorted element.

[**Output:-**](https://www.programiz.com/dsa/selection-sort)

Sorted array is:

11

12

22

25

64

Execution time:-0.015599966049194336

**Working:-**

Selection sort works by taking the smallest element in an unsorted array and bringing it to the front. You'll go through each item (from left to right) until you find the smallest one. The first item in the array is now sorted, while the rest of the array is unsorted.

**//insertion sort**

import time

def insertionsort(arr):

for i in range(1,len(arr)):

key=arr[i]

j=i-1

while j>=0 and key<arr[j]:

arr[j+1]=arr[j]

j-=1

arr[j+1]=key

arr=[12,11,13,5,6]

st=time.time()

insertionsort(arr)

et=time.time()

etime=st-et

print("execution time:",etime)

print("sorted array is:")

for i in range(len(arr)):

print("%d" %arr[i])

**Algorithm:-**

To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**Output:-**

Execution time:-0.0

Sorted array is:-

5 6 11 12 13

**Working:-**

Selection sort works by taking the smallest element in an unsorted array and bringing it to the front. You'll go through each item (from left to right) until you find the smallest one. The first item in the array is now sorted, while the rest of the array is unsorted.

**Conclusion:-**

Since execution time is less in insertion sort. Insertion sort is optimal.

**Implement binary search using divde and conquer approach:-**

def binarySearch(array,x,low,high)

while low <= high:

mid=low+(high-low)//2

if array[mid]==x:

return mid

elif array[mid]<x:

low=mid+1

else:

high=mid-1

return-1

array=[3,4,5,6,7,8,9]

x=6

result=binarySearch(array,x,0,len(array)-1)

if result !=-1:

print("element",x,"is present at index"+str(result))

else:

print("not found")

**Algorithm:-**

1. Divide the search space into two halves by [**finding the middle index “mid”**](https://www.geeksforgeeks.org/problem-binary-search-implementations/).
2. Compare the middle element of the search space with the key.
3. If the key is found at middle element, the process is terminated.
4. If the key is not found at middle element, choose which half will be used as the next search space.
   1. If the key is smaller than the middle element, then the left side is used for next search.
   2. If the key is larger than the middle element, then the right side is used for next search.
5. This process is continued until the key is found or the total search space is exhausted.

**Output:-**

Element 6 is present at index 3

**Working:-**

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

**Write a program to implement merge sort**

def mergesort(arr):

if len (arr)>1:

mid=len(arr)//2

L=arr[mid]

R=arr[mid:]

mergesort(L)

mergesort(R)

i=j=k=0

while i<len(L) and j<len(R):

if L[i]<R[j]:

arr[K]=L[i]

i+=1

else:

arr[K]=R[i]

j+=1

k+=1

while i<len(2):

arr[K]=R[j]

j+=1

k+=1

def printlist(arr):

for i in range(len(arr)):

print(arr[i],end=" ")

print()

if\_\_name\_\_='\_\_main\_\_';

arr=[12,11,13,5,6,7]

print("given array is",end="\n")

printlist(arr)

mergesort(arr)

print("sorted array",end="\n")

printlist(arr)

**output**:-

given array is:

12 11 13 5 6 7

Sorted array is:

5 6 7 11 12 13

**Time complexity**:- Jun 27 2018, 04:59:51

**Space complexity**:-MSC v.1914 64 bit

**Working**:-

Merge Sort is a Divide and Conquar algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves.

**Algorithm :-**

**Step 1--** start

**Step 2** − if it is only one element in the list it is already sorted, return.

**Step 3** − divide the list recursively into two halves until it can no more be divided.

**Step 4** − merge the smaller lists into new list in sorted order.

**Step 5—**stop

**Write a program to implement bucket sort**

def bucketSort(array):

bucket = []

for i in range(len(array)):

bucket.append([])

for j in array:

index\_b = int(10 \* j)

bucket[index\_b].append(j)

for i in range(len(array)):

bucket[i] = sorted(bucket[i])

k = 0

for i in range(len(array)):

for j in range(len(bucket[i])):

array[k] = bucket[i][j]

k += 1

return array

array = [.42, .32, .33, .52, .37, .47, .51]

print("Sorted Array in descending order is")

print(bucketSort(array))

**output:-**

Sorted Array in descending order is

[0.32, 0.33, 0.37, 0.42, 0.47, 0.51, 0.52]

**Time complexity**:- Jun 27 2018, 04:59:51

**Space complexity**:- MSC v.1914 64 bit

**Working**:-

Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm.

**write a program to implement binary search tree using min & max**

class Node:

def \_\_init\_\_(self,data):

self.key=data

self.lchild=None

self.rchild=None

def insert(self,data):

if self.key is None:

self.key=data

return

if self.key>=data:

if self.lchild:

self.lchild.insert(data)

else:

self.lchild==Node(data)

else:

if self.rchild:

self.rchild.insert(data)

else:

self.rchild=Node(data)

def min\_node(self):

current=slf

while current.lchild:

current=current.lchild

print("min node=",current.key)

def max\_node(self):

current=self

while current.rchild:

current=current.rchild

print("max node=",current.key)

root=Node(Node)

List1=[6,7,9,3,67,2]

for i in List1:

root.insert(i)

root.max\_node()

root.max\_node()

**Output:-**

Node is found!

Time complexity:- Jun 27 2018, 04:59:51

Space complexity:- MSC v.1914 64 bit

**Algorithm:-**

1.start

2. Create a new BST node and assign values to it.

3. insert(node, key)

     i) If root == NULL,

         return the new node to the calling function.

     ii) if root=>data < key

         call the insert function with root=>right and assign the return value in root=>right.

        root->right = insert(root=>right,key)

     iii) if root=>data > key

         call the insert function with root->left and assign the return value in root=>left.

         root=>left = insert(root=>left,key)

4. Finally, return the original root pointer to the calling function.

5.stop

**Working:-**

A binary search tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

**write a program to implement Evalute Expression Tree**

class Node:

def \_\_init\_\_(self,data):

self.key=data

self.lchild=None

self.rchild=None

def evaluateexpressionstree(self,data):

if root is None:

return

if root.left is None and root.right is None:

return (root.data)

left\_sum=evaluateexpressionstree(root.left)

right\_sum=evaluateexpressionstree(root.right)

if root.data=='+':

return left\_sum+right\_sum

elif root.data=='-':

return left\_sum-right\_sum

elif root.data=='\*':

return left\_sum\*right\_sum

else:

return left\_sum//right\_sum

if \_\_name\_\_=="\_\_main\_\_":

root=node('+')

root.left=node('-')

root.left.left=node('5')

root.left.rigth=node('-4')

root.rigth=node('\*')

root.rigth.left=node('100')

root.rigth.rigth=node('20')

print(evaluateexpressionstree(root))

root=None

**Output:-**

60

Time complexity:- Jun 27 2018, 04:59:51

Space complexity:- MSC v.1914 64 bit

**Algorithm:-**

1. LEFT = 0 RIGHT = 1.
2. Define a function evaluate() . This will take root. if value of root is a numeric value, then. return integer representation of value of root. ...
3. Define a function buildTree() . This will take postfix. root := null. stack := a new list. ...
4. return root.

**Working:-**

We can evaluate an expression tree by applying the operator at the root to values obtained by recursively evaluating left and right subtrees. This can be easily done by traversing the expression tree using postorder traversal. The algorithm can be implemented as follows in C++, Java, and Python: C++

**Write a program to implement binary search**

def binarysearch(array,x,low,high):

while low<=high:

mid=low+(high-low)//2

if array[mid]==x:

return mid

elif array[mid]<x:

low=mid+1

else:

high=mid-1

return-1

array=[3,4,5,6,7,8,9]

x=6

result=binarysearch(array,x,0,len(array)-1)

if result != -1:

print("element",x,"is present at index"+str(result))

else:

print("not found")

**Output:-**

element 6 is present at index3

**time complexity**:- Jun 27 2018, 04:59:51

**space complexity**:- MSC v.1914 64 bit

**Algorithm:-**

1.Start

2.compare x with the middle element

3.if x matches with the middle element return the mid index

4.else if x greater than the mid element ,than x can only lie in right half subarray after the mid element,so we run for the right half

5.else(x is smaller) reverse the left half

6.stop

**Algorithm for LinkedList is :**

1. Start by defining the Node class with the data and ref attributes.

2. Define the LinkedList class with the head attribute.

3. Implement the print\_LL method:

4. Implement the add\_begin method:

5. Implement the add\_end method:

6. Implement the add\_after method:

7. Implement the add\_before method:

8. Implement the delete\_begin method:

9. Implement the delete\_end method:

10. Create an instance LL1 of the LinkedList class.

11. Add nodes to the linked list using the add\_begin, add\_end, and add\_after methods with the provided input values.

12. Print the linked list using the print\_LL method.

13. Delete a node at the beginning of the linked list using the delete\_begin method.

14. Print the linked list after deletion.

15. Delete a node at the end of the linked list using the delete\_end method.

16. Print the final linked list after deletion.

**Source code for LinkedList is :**

N1=int(input("Enter the one value = "))

N2=int(input("Enter the two value = "))

N3=int(input("Enter the Three value = "))

N4=int(input("Enter the Four value = "))

N5=int(input("Enter the End value = "))

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.ref=None

class Linkedlist:

def \_\_init\_\_(self):

self.head=None

def print\_LL(self):

if self.head==None:

print("Linkedlist is empty")

else:

n=self.head

while n is not None:

print(n.data,"-->",end="")

n=n.ref

print("NULL")

def add\_begin(self,data):

new\_node=Node(data)

new\_node.ref=self.head

self.head=new\_node

def add\_end(self,data):

new\_node=Node(data)

if self.head is None:

self.head=new\_node

else:

n=self.head

while n.ref is not None:

n=n.ref

n.ref=new\_node

def add\_after(self,data,x):

n=self.head

while n is not None:

if x==n.data:

break

n=n.ref

if n is None:

print("Node is not present in LinkedList")

else:

new\_node=Node(data)

new.node.ref=n.ref

n.ref=new\_node

def add\_before(self,data,x):

if self.head==None:

print("Linkedlist is empty")

return

if self.head.data==x:

new\_node=Node(data)

new\_node.ref=self.head

self.head=new\_node

return

n=self.head

def delete\_begin(self):

if self.head is None:

print("Linkedlist is empty")

else:

self.head=self.head.ref

def delete\_end(self):

if self.head is None:

print ("Linkedlist is empty")

elif self.head.ref is None:

self.head=None

else:

n=self.head

while n.ref:

n=n.ref

n.ref=None

LL1=Linkedlist()

LL1.add\_begin(N1)

LL1.add\_begin(N2)

LL1.add\_begin(N3)

LL1.add\_begin(N4)

LL1.add\_end(N5)

LL1.add\_after(20,50)

LL1.print\_LL()

LL1.delete\_begin()

print("Deleteing a node at the Beginning")

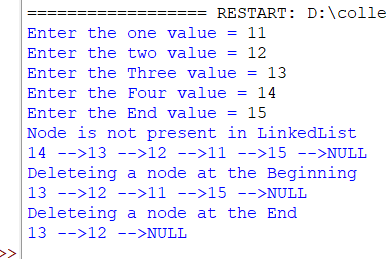
LL1.print\_LL()

LL1.delete\_end()

print("Deleteing a node at the End")

LL1.print\_LL()

**Output for LinkedList is:**



**The working principle for the LinkedList is :**

1. The code begins by creating a linked list with the LinkedList class and initializing it with a head attribute, which initially points to None.
2. The Node class is defined, which represents a single node in the linked list. Each node has a data attribute to store the value and a ref attribute to reference the next node in the list.
3. The print\_LL method is used to print the contents of the linked list. It starts from the head node and traverses the list using a while loop until it reaches the end (i.e., ref becomes None). It prints each node's data value and appends "-->" to indicate the next node. Finally, it prints "NULL" to signify the end of the linked list.
4. The add\_begin method adds a new node at the beginning of the linked list. It creates a new node with the given data, sets the ref attribute of the new node to the current head node, and updates the head attribute to point to the new node.
5. The add\_end method adds a new node at the end of the linked list. It creates a new node with the given data. If the linked list is empty (i.e., head is None), it sets the head attribute to the new node. Otherwise, it traverses the list until the last node is reached and sets the ref attribute of the last node to the new node.
6. The add\_after method adds a new node after a specified value in the linked list. It starts from the head node and traverses the list until it finds the node with the given value or reaches the end. If the value is found, it creates a new node with the given data and inserts it after the current node by updating the ref attributes accordingly. If the value is not found, it prints a message indicating that the node is not present in the linked list.
7. The add\_before method adds a new node before a specified value in the linked list. It checks if the linked list is empty and if the head node itself has the specified value. If true, it inserts the new node before the current head node. Otherwise, it traverses the list to find the node with the specified value. If found, it inserts the new node before the current node by updating the ref attributes. If the value is not found, it prints a message indicating that the node is not present in the linked list.
8. The delete\_begin method deletes the first node in the linked list. It checks if the linked list is empty. If not, it updates the head attribute to the next node, effectively removing the first node from the list.
9. The delete\_end method deletes the last node in the linked list. It checks if the linked list is empty or contains only one node. If true, it updates the head attribute to None, effectively emptying the list. Otherwise, it traverses the list until the second-to-last node is reached and updates its ref attribute to None, effectively removing the last node from the list.
10. Finally, the code creates an instance of the LinkedList class and performs a series of operations on the linked list, such as adding nodes at the beginning and end, adding a node after a specified value, deleting a node at the beginning and end, and printing the final state of the linked list.

**The time and space complexity of LinkedList is :**

**Time Complexity:**

1. Adding a node at the beginning (add\_begin): O(1)
2. Adding a node at the end (add\_end): O(N), where N is the number of nodes in the linked list, as we need to traverse the entire list to reach the last node.
3. Adding a node after a specified value (add\_after): O(N), as we may need to traverse the entire list to find the specified value.
4. Adding a node before a specified value (add\_before): O(N), as we may need to traverse the entire list to find the specified value.
5. Deleting a node at the beginning (delete\_begin): O(1)
6. Deleting a node at the end (delete\_end): O(N), as we may need to traverse the entire list to reach the second-to-last node.
7. Printing the linked list (print\_LL): O(N), as we need to traverse the entire list to print all the nodes.
8. Overall, the time complexity of the code depends on the number of nodes in the linked list and can be expressed as O(N), where N is the number of nodes.

**Space Complexity:**

1. The space complexity of the code is O(N), where N is the number of nodes in the linked list. This is because we need to allocate memory for each node in the linked list.

**NOTE:**

That these complexities are based on the operations and logic provided in the code snippet. Additional operations or modifications may alter the time and space complexity.

**Program 2 :**

**Algorithm of heapify is :**

1. Create an empty list called heap.
2. Import the heapq module, which provides functions to manipulate heap data structures.
3. Use the heappush function from heapq to push elements into the heap. Each element is a tuple containing two values: the priority (an integer) and the corresponding data (a string).
4. Repeat step 3 for all the elements you want to add to the heap.
5. Print "The order is:" to indicate the start of the output.
6. Iterate over the elements in the heap and print each element's priority and data.
7. End the program.

**Source code for heapify is :**

import heapq

heap=[10,20,30,1,2,3]

heapq.heapify(heap)

print(heap)

heap

**Output for heapify:**



**Working principle of heapify is :**

1. Create an empty list called heap. This list will be used to store elements in a heap structure.
2. Import the heapq module, which provides functions to manipulate heap data structures.
3. Use the heappush function from the heapq module to push elements into the heap. Each element is a tuple containing two values: the priority (an integer) and the corresponding data (a string). The heappush function ensures that the element is inserted into the heap while maintaining the heap property, where the element with the highest (or lowest) priority is always at the root position.
4. Repeat step 3 for all the elements you want to add to the heap.
5. Print "The order is:" to indicate the start of the output.
6. Iterate over the elements in the heap and print each element's priority and data. Since the elements are stored in a heap, they will be printed in the order of their priorities.

**Time and Space complixty of heapify is :**

* 1. **Time Complexity:**
  2. Creating an empty list takes constant time, O(1).
  3. Importing the heapq module has a negligible time complexity, as it is a built-in module.
  4. For each call to heappush, the time complexity is O(log n), where n is the number of elements in the heap. This is because heappush performs the necessary operations to maintain the heap property.
  5. The iteration over the elements in the heap and the subsequent printing have a time complexity of O(n), where n is the number of elements in the heap.
  6. Therefore, the overall time complexity of the code is O(n log n), where n is the number of elements in the heap.

1. **Space Complexity:**
   1. The space complexity is primarily determined by the heap list and any additional memory used by the heapq module.
   2. The heap list stores the elements in the heap. The space complexity for the heap list is O(n), where n is the number of elements in the heap.
   3. Importing the heapq module requires a small constant amount of memory and does not significantly contribute to the overall space complexity.
   4. Therefore, the overall space complexity of the code is O(n), where n is the number of elements in the heap.

**Note:**

That the space complexity mentioned above does not include the space required for the input data itself, as it is assumed to be provided as part of the code or initialization.

**Program 3:**

**Algorithm of SparseMatrix is :**

1. Define a function displayMatrix that takes a matrix as input:
2. Define a function convertToSparseMatrix that takes a matrix as input:
3. Create a normal matrix called normalMatrix with some values.
4. Call the convertToSparseMatrix function, passing normalMatrix as the argument.

**Source code for SparseMatrix is :**

def displayMatrix(matrix):

for row in matrix:

for element in row:

print(element, end=" ")

print()

def convertToSparseMatrix(matrix):

sparseMatrix = []

for i in range(len(matrix)):

for j in range(len(matrix[0])):

if matrix[i][j] != 0:

temp = []

temp.append(i)

temp.append(j)

temp.append(matrix[i][j])

sparseMatrix.append(temp)

print("\nSparse Matrix: ")

displayMatrix(sparseMatrix)

normalMatrix = [[1, 0, 0, 0],

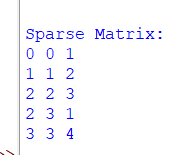
[0, 2, 0, 0],

[0, 0, 3, 1],

[0, 0, 0, 4]]

convertToSparseMatrix(normalMatrix)

**Output for SparseMatrix is :**



**The working principle of SparseMatrix is :**

1. The displayMatrix function takes a matrix as input and iterates over each row and element in the matrix. It prints each element followed by a space and moves to the next row by printing a newline character. This function is used to display matrices in a readable format.
2. The convertToSparseMatrix function takes a matrix as input and converts it into a sparse matrix representation. It initializes an empty list called sparseMatrix to store the non-zero elements of the matrix along with their row and column indices. It then iterates over each element in the matrix and checks if the element is non-zero. If it is non-zero, it creates a temporary list called temp and appends the row index, column index, and element value to temp. Finally, temp is appended to the sparseMatrix list.
3. After converting the matrix to a sparse matrix representation, the function prints "Sparse Matrix: " to indicate the start of the output. It then calls the displayMatrix function, passing the sparseMatrix as an argument, to display the sparse matrix in a readable format.
4. The code initializes a normal matrix called normalMatrix with some values.
5. The convertToSparseMatrix function is called with normalMatrix as the argument, which converts normalMatrix into its sparse matrix representation and displays it.
6. The output of the code will be the sparse matrix representation of the given normal matrix. The sparse matrix will consist of the non-zero elements from the normal matrix along with their corresponding row and column indices.

**Note:**

That the code snippet you provided already implements this working principle, and the expected output will be the sparse matrix representation of the normal matrix.

**The time and space complexity is :**

**Time Complexity:**

1. The displayMatrix function iterates over each element in the matrix and prints it. Since there are m rows and n columns in the matrix, the time complexity of the displayMatrix function is O(m\*n).
2. The convertToSparseMatrix function iterates over each element in the matrix and checks if it is non-zero. Since there are m rows and n columns in the matrix, the time complexity of the convertToSparseMatrix function is also O(m\*n).
3. Inside the convertToSparseMatrix function, when a non-zero element is found, a temporary list temp is created and the row index, column index, and element value are appended to it. This operation takes constant time, O(1).
4. The temp list is then appended to the sparseMatrix list. Since there can be a maximum of m\*n non-zero elements in the matrix, the total number of iterations for appending temp to sparseMatrix will be at most m\*n. Therefore, the time complexity of this operation is O(m\*n).
5. Finally, the displayMatrix function is called to print the sparse matrix. As mentioned earlier, the time complexity of the displayMatrix function is O(m\*n).

Therefore, the overall time complexity of the code is O(m\*n), where m is the number of rows in the matrix and n is the number of columns in the matrix.

**Space Complexity:**

1. The sparseMatrix list is used to store the non-zero elements of the matrix along with their row and column indices. The maximum number of non-zero elements in the matrix can be m\*n, so the space complexity of sparseMatrix is O(m\*n).
2. Other than sparseMatrix, the code uses a few temporary variables and function call stack frames, which require a constant amount of space and do not significantly contribute to the overall space complexity.

Therefore, the overall space complexity of the code is O(m\*n), where m is the number of rows in the matrix and n is the number of columns in the matrix.

**Note:**

That the space complexity mentioned above does not include the space required to store the input normalMatrix itself, as it is assumed to be provided as part of the code or initialization.

**Program 4:**

**Algorithm of pre-in-out order is :**

1. Define a class Node that represents a node in a binary tree.
2. Create a root node with a key value of 10.
3. Create a list called list1 with some values to be inserted into the binary tree.
4. Iterate over each value in list1.
5. Print "preorder :" to indicate the start of the preorder traversal.
6. Print "inorder :" to indicate the start of the inorder traversal.
7. Print "postorder :" to indicate the start of the postorder traversal.

The output of the code will be the keys of the nodes in the binary tree printed in the specified order (preorder, inorder, and postorder).

**Note:**

That the code snippet you provided already implements this algorithm, and the expected output will be the preorder, inorder, and postorder traversals of the binary tree.

**Source code for pre-in-post order :**

class Node:

def \_\_init\_\_(self,data):

self.key=data

self.lchild=None

self.rchild=None

def insert(self,data):

if self.key is None:

self.key=data

return

if self.key>=data:

if self.lchild:

self.lchild.insert(data)

else:

self.lchild=Node(data)

else:

if self.rchild:

self.rchild.insert(data)

else:

self.rchild=Node(data)

def search(self,data):

if self.key == data:

print("Found")

return

if data<self.key:

if self.lchild:

self.lchild.search(data)

else:

print("Not Found")

else:

if self.rchild:

self.rchild.search(data)

else:

print("Not Found")

def preorder(self):

print(self.key,end=" ")

if self.lchild:

self.lchild.preorder()

if self.rchild:

self.rchild.preorder()

def inorder(self):

print(self.key,end=" ")

if self.lchild:

self.lchild.inorder()

if self.rchild:

self.rchild.inorder()

def postorder(self):

print(self.key,end=" ")

if self.lchild:

self.lchild.postorder()

if self.rchild:

self.rchild.postorder()

root=Node(10)

list1=(20,4,30,4,1,5,6)

for i in list1:

root.insert(i)

print("preorder :")

root.preorder()

print()

print("inorder :")

root.inorder()

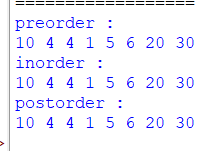
print()

print("postorder :")

root.postorder()

print()

**Output for pre-in-post order is :**



**The working principle of the pre-in-post order is :**

1. The code defines a class called Node that represents a node in a binary tree. Each node has a key value, a left child node, and a right child node.
2. The insert method in the Node class is used to insert a new node into the binary tree. It recursively traverses the tree, comparing the key of the current node with the value to be inserted, and decides whether to insert it in the left subtree or the right subtree.
3. The search method in the Node class is used to search for a node with a specific value in the binary tree. It recursively traverses the tree, comparing the key of the current node with the search value, and moves to the left or right subtree accordingly. If the node with the desired value is found, it prints "Found". Otherwise, it prints "Not Found".
4. The preorder, inorder, and postorder methods in the Node class are used to perform different types of tree traversals: preorder, inorder, and postorder, respectively. These methods print the key of the current node and then recursively call themselves on the left and right child nodes, following the specified order.
5. The code creates a root node with a key value of 10.
6. A list called list1 is defined, which contains a set of values.
7. The code iterates over each value in list1 and inserts it into the binary tree using the insert method.
8. The code then performs three types of tree traversals: preorder, inorder, and postorder. For each traversal, it calls the corresponding method on the root node, which recursively traverses the tree and prints the key values of the nodes in the specified order.

The output of the code will be the keys of the nodes in the binary tree printed in the specified order: preorder, inorder, and postorder.

The working principle of the code is to demonstrate the insertion of nodes into a binary tree, search for a specific value, and perform different types of tree traversals to visit the nodes in a specific order.

**The time and space complexity of pre-in-post order is:**

**Time Complexity:**

1. Insertion: The time complexity of inserting a new node into a binary tree depends on the height of the tree. In the worst case, if the tree is unbalanced, the height can be equal to the number of nodes in the tree. Therefore, the time complexity of the insert method is O(n), where n is the number of nodes in the tree.
2. Search: Similar to insertion, the time complexity of searching for a node in a binary tree also depends on the height of the tree. In the worst case, if the tree is unbalanced, the time complexity of the search method is O(n), where n is the number of nodes in the tree.
3. Traversals: Preorder, inorder, and postorder traversals visit each node in the binary tree exactly once. Therefore, the time complexity of each traversal method (preorder, inorder, and postorder) is O(n), where n is the number of nodes in the tree.
4. Overall, the time complexity of the code is dominated by the time complexity of the insertion or search operation, which is O(n), where n is the number of nodes in the tree.

**Space Complexity:**

1. The space complexity of the binary tree itself is O(n), where n is the number of nodes in the tree.
2. Recursive Function Calls: The space complexity of recursive function calls depends on the maximum depth of the recursion, which is equal to the height of the tree. In the worst case, if the tree is unbalanced, the height can be equal to the number of nodes in the tree. Therefore, the space complexity of recursive function calls is O(n), where n is the number of nodes in the tree.
3. Additional Variables: The code uses a few additional variables such as data, temp, and list1, which require a constant amount of space and do not significantly contribute to the overall space complexity.