Stack implementation using arrays

| class stack(object): def \_\_init\_\_(self,limit=10): self.stk=[] self.limit=limit def isEmpty(self): return len(self.stk)<=0 def push(self,item): if len(self.stk)>=self.limit: print ("stack overflow") else: self.stk.append(item) print('stack after push',self.stk) def pop(self): if self.isEmpty(): print ('stack underflow') return 0 else: return self.stk.pop() def peek(self): if self.isEmpty(): print('stack underflow') return 0 else: return self.stk[-1] def size(self): return len(self.stk)stack\_obj=stack(10)stack\_obj.push("1")stack\_obj.push("2")stack\_obj.push("3")stack\_obj.push("4")stack\_obj.push("5")stack\_obj.push("6")stack\_obj.push("7")stack\_obj.push("8")print(stack\_obj.peek())print(stack\_obj.pop())print(stack\_obj.peek())print(stack\_obj.pop()) |
| --- |

Output

stack after push ['1']

stack after push ['1', '2']

stack after push ['1', '2', '3']

stack after push ['1', '2', '3', '4']

stack after push ['1', '2', '3', '4', '5']

stack after push ['1', '2', '3', '4', '5', '6']

stack after push ['1', '2', '3', '4', '5', '6', '7']

stack after push ['1', '2', '3', '4', '5', '6', '7', '8']

8

8

7

7

Queue implementation using arrays

| class Queue: # initialize the object def \_\_init\_\_(self,c): self.queue=[] self.front=self.rear=0 self.capacity=c def isFull(self): if(self.capacity == self.rear): return 1 else: return 0  def isEmpty(self): if(self.front == self.rear): return 1 else: return 0  # insert an element def qEnqueue(self,data): if(self.isFull()): print("\n Queue is full") else: self.queue.append(data) self.rear +=1 # function to delete an element def qDequeue(self): if(self.isEmpty()): print("Queue is empty") else: x=self.queue.pop(0) self.rear -=1 # function to print queue elements def qDisplay(self): if(self.isEmpty()): print("Queue is empty") else: for i in self.queue: print(i,"<--",end='') # print front of queue def qFront(self): if(self.isEmpty()): print('Queue is empty') else: print('\n Front element is :', self.queue[self.front]) # driver codeif \_\_name\_\_=='\_\_main\_\_': q=Queue(5) q.qDisplay() q.qEnqueue(10) q.qEnqueue(20) q.qEnqueue(30) q.qEnqueue(40) q.qEnqueue(50) q.qDisplay() q.qEnqueue(60) q.qDisplay() q.qDequeue() q.qDequeue() print("\n\n after two elements deletion") q.qDisplay() q.qFront()  |
| --- |

Output

Queue is empty

10 <--20 <--30 <--40 <--50 <--

 Queue is full

10 <--20 <--30 <--40 <--50 <--

 after two elements deletion

30 <--40 <--50 <--

 Front element is : 30

Implementation of Linked list

| # linked list implementation class Node: def \_\_init\_\_(self,data): self.data=data self.next=Noneclass LinkedList: def \_\_init\_\_(self): self.head=None# insertion at the beginning def insertBegin(self,new\_data): new\_node=Node(new\_data) if self.head==None: self.head=new\_node else: new\_node.next=self.head self.head=new\_node# insert after a particular node def insertAt(self,pos,new\_data): count=1 curr=self.head while count<pos-1 and curr !=None: curr=curr.next count=count+1 new\_node=Node(new\_data) new\_node.next=curr.next curr.next=new\_node# insert at the end def insertEnd(self,new\_data): new\_node=Node(new\_data) if self.head is None: self.head=new\_node return temp=self.head while (temp.next!=None): temp=temp.next temp.next=new\_node# search for an element def search(self,key): temp=self.head while temp is not None: if temp.data == key: return True temp=temp.next return False# deleting a node def deleteBegin(self): try: if self.head==None: raise Exception("Empty list") else: temp=self.head self.head=self.head.next del temp except Exception as e: print(str(e))# deleting the last node def deleteEnd(self): try: if self.head==None: raise Exception("empty list") else: curr=self.head prev=None while curr.next != None: prev=curr curr=curr.next prev.next=curr.next del curr except Exception as e: print(str(e))# deleting in between node def deleteAt(self,pos): try: if self.head== None: raise Exception("empty list") else: curr=self.head prev=None count=1 while curr != None and count<pos: prev=curr curr=curr.next count=count+1 prev.next=curr.next del curr except Exception as e: print(str(e))# print the linked list def printlist(self): temp=self.head while(temp!=None): print(str(temp.data)+" ",end=" ") temp=temp.next print("\n") if \_\_name\_\_=='\_\_main\_\_': list1=LinkedList() # assign item values list1.insertBegin(1) list1.insertEnd(5) list1.printlist() list1.insertAt(2,10) list1.printlist() list1.insertAt(2,8) list1.printlist() list1.insertAt(2,9) list1.printlist() list1.insertEnd(6) list1.printlist() list1.deleteBegin() list1.printlist() list1.deleteEnd() list1.printlist() list1.deleteAt(3) list1.printlist() #print the list list1.printlist() search\_ele=5 if list1.search(search\_ele): print(str(search\_ele)+ " is found") else: print(str(search\_ele)+ " is not found") |
| --- |

Output

1 5

1 10 5

1 8 10 5

1 9 8 10 5

1 9 8 10 5 6

9 8 10 5 6

9 8 10 5

9 8 5

9 8 5

5 is found

**Doubly linked list**

| class Node: def \_\_init\_\_(self, data=None): self.data=data self.next=None self.prev=None # create a doubly linked list class to initialize head and tail referencesclass DDL: def \_\_init\_\_(self): self.head=None self.tail=None #insert node at the front of Doubly linked list def insertBegin(self,data): new\_node=Node(data) new\_node.next=self.head if self.head is not None: self.head.prev=new\_node self.head=new\_node # insert a node at the end def insertEnd(self,data): new\_node=Node(data) if self.head is None: self.head=new\_node return temp=self.head while temp.next!=None: temp=temp.next new\_node.prev=temp temp.next=new\_node return#insert a node after a specific node def insertAt(self,pos,data): new\_node=Node(data) count=1 # check if previous node is null curr=self.head while count<pos-1 and curr !=None: curr=curr.next count=count+1 new\_node.next=curr.next new\_node.prev=curr curr.next=new\_node if new\_node.next!=None: new\_node.next.prev=new\_node  def deleteBegin(self): if self.head is None: print("ths list is empty") return else: temp=self.head self.head=self.head.next self.head.prev=None del temp def deleteEnd(self): if self.head is None: print("the list is empty") return else: curr=self.head prev=None while curr.next!=None: prev=curr curr=curr.next prev.next=curr.next del curr def deleteAt(self,pos): if self.head==None: print("the list is empty") return else: curr=self.head prev=None count=1 while curr!=None and count<pos: prev=curr curr=curr.next count=count+1 prev.next=curr.next curr.next.prev=prev del curr def search\_node(self,value): if self.head==None: print("the list is empty") return else: curr=self.head while curr!=None: if curr.data==value: print("value is present in the list") return else: curr=curr.next if curr is None: print("element is not in the list")  def update\_node(self,old\_value,new\_value): if self.head==None: print("the list is empty") return else: curr=self.head while curr!=None: if curr.data==old\_value: curr.data=new\_value return else: curr=curr.next if curr is None: print("element is not in the list")   def displayFlist(self): if self.head == None: print(" the linked list does not exist") else: temp=self.head while temp: print(temp.data,end=" ") temp=temp.next print("\n")# printing reversely  def displayRlist(self): if self.head == None: print(" the linked list does not exist") else: temp=self.head while temp.next!=None: temp=temp.next  while(temp!=self.head): print(temp.data,end=" ") temp=temp.prev print(temp.data)# initialize the linked list with a new nodedll1=DDL()dll1.insertBegin(5)dll1.displayFlist()dll1.insertBegin(2)dll1.insertEnd(10)dll1.insertEnd(18)dll1.displayFlist()dll1.insertAt(2,15)dll1.displayFlist()dll1.insertAt(2,16)dll1.displayFlist()dll1.insertAt(3,17)dll1.displayFlist()dll1.insertAt(4,20)dll1.displayFlist()dll1.deleteBegin()dll1.displayFlist()dll1.deleteEnd()dll1.displayFlist()dll1.deleteAt(3)dll1.displayFlist()dll1.displayRlist()dll1.search\_node(10)dll1.search\_node(18)dll1.update\_node(10,45)dll1.displayFlist() |
| --- |

Output

5

2 5 10 18

2 15 5 10 18

2 16 15 5 10 18

2 16 17 15 5 10 18

2 16 17 20 15 5 10 18

16 17 20 15 5 10 18

16 17 20 15 5 10

16 17 15 5 10

10 5 15 17 16

value is present in the list

element is not in the list

16 17 15 5 45



**Circular linked list**

| class Node: def \_\_init\_\_(self,data): self.data=data self.next=Noneclass CCL: def \_\_init\_\_(self): self.tail=None  def isEmpty(self): if self.tail==None: return True else: return False  def insertBegin(self,data): new\_node=Node(data) if self.isEmpty(): self.tail=new\_node new\_node.next=new\_node else: new\_node.next=self.tail.next self.tail.next=new\_node def insertEnd(self,data): new\_node=Node(data) if self.isEmpty(): self.tail=new\_node new\_node.next=new\_node else: new\_node.next=self.tail.next self.tail.next=new\_node self.tail=new\_node def insertMiddle(self,data,item): if self.isEmpty(): return None new\_node=Node(data) p=self.tail.next  while p: # if the item is found, place newnode after it if p.data==item: # make the next of the current node as the next of newnode new\_node.next=p.next # put new node to the next of p p.next=new\_node if p==self.tail: self.tail=new\_node return self.tail else: return self.tail p=p.next if p==self.tail.next: print(item," is not present in list") break def deleteBegin(self): if self.isEmpty(): print("list is empty") return # if list has only one node if self.tail.next==self.tail: self.tail.next=None self.tail=None return # if list has more than one node first\_node=self.tail.next second\_node=first\_node.next self.tail.next=second\_node del(first\_node) def deleteEnd(self): if self.isEmpty(): print("list is empty") return # if list has only one node if self.tail.next==self.tail: self.tail.next=None self.tail=None return # if list has more than one node last\_node=self.tail prev\_node=last\_node.next while prev\_node.next!=last\_node: prev\_node=prev\_node.next prev\_node.next=last\_node.next self.tail=prev\_node del(last\_node) def deleteNode(self,key): temp=self.tail d=None while temp.next!=self.tail and temp.next.data !=key: temp=temp.next if temp.next.data==key: d=temp.next temp.next=d.next  def search(self,key): curr=self.tail while curr.next!=self.tail: if curr.data==key: return True curr=curr.next return False def printlist(self): if self.tail==None: print("list is empty") return temp=self.tail.next while temp: print(temp.data, end="-->") temp=temp.next if temp == self.tail.next: break print("\n")my\_list=CCL()my\_list.insertBegin(5)my\_list.printlist()my\_list.insertBegin(15)my\_list.printlist()my\_list.insertBegin(20)my\_list.printlist()my\_list.insertBegin(25)my\_list.printlist()my\_list.insertBegin(3)my\_list.printlist()my\_list.insertEnd(40)my\_list.insertEnd(23)my\_list.insertEnd(33)my\_list.printlist()my\_list.deleteBegin()my\_list.printlist()my\_list.deleteBegin()my\_list.printlist()my\_list.deleteEnd()my\_list.printlist()my\_list.insertMiddle(6,15)my\_list.printlist()my\_list.deleteNode(40)my\_list.printlist()flag=my\_list.search(9)if flag: print("element is present in the list")else: print("element is absent in the list") |
| --- |

Output

5-->

15-->5-->

20-->15-->5-->

25-->20-->15-->5-->

3-->25-->20-->15-->5-->

3-->25-->20-->15-->5-->40-->23-->33-->

25-->20-->15-->5-->40-->23-->33-->

20-->15-->5-->40-->23-->33-->

20-->15-->5-->40-->23-->

20-->15-->6-->5-->40-->23-->

20-->15-->6-->5-->23-->

element is absent in the list

Binary search Tree

| # Binary Search treeclass Node: def \_\_init\_\_(self,data): self.left=None self.right=None self.data=data def printTree(self): if self.left: self.left.printTree() print(self.data) if self.right: self.right.printTree() def inorder(self,root): if root: self.inorder(root.left) print(str(root.data)+"->", end="") self.inorder(root.right) def preorder(self,root): if root: print(str(root.data)+"->", end="") self.preorder(root.left) self.preorder(root.right) def postorder(self,root): if root: self.postorder(root.left) self.postorder(root.right) print(str(root.data)+"->", end="") def insertTree(self,data): # compare the new value with the parent node if self.data: if data < self.data: if self.left is None: self.left=Node(data) else: self.left.insertTree(data) elif data > self.data: if self.right is None: self.right=Node(data) else: self.right.insertTree(data) else: self.data=data def findNode(self,key): if key<self.data: if self.left is None: return str(key)+" Not found" else: return self.left.findNode(key) elif key > self.data: if self.right is None: return str(key) + "Not found" else: return self.right.findNode(key) else: return str(self.data)+" is found"  def inorder\_succ(self,cur): while cur.left!=None: cur=cur.left return cur def deleteNode(self,root,key): #pointer to store the parent of current node parent=None # start with root node cur=root #search key in the BST and set its parent pointer while cur and cur.data!=key: #update the parent to the current node parent=cur #if the given key is less than the current node, go to left subtree # other wise go to the right subtree if key < cur.data: cur=cur.left else: cur=cur.right # return if the key is not found in the tree if cur is None: return root #case 1 : node to be deleted has no children - leaf node if cur.left is None and cur.right is None: # if the node to be deleted is not a root node, then set # its parent left/right child to None if cur != root: if parent.left==cur: parent.left=None else: parent.right=None # if the tree has only a root node, set it to None else: root=None # case 2: node to be deleted has two children elif cur.left and cur.right: #find the inorder successor node successor=self.inorder\_succ(cur.right) val=successor.data #recursively delete the successor. # the successor will have at most one child (right child) self.deleteNode(root,successor.data) cur.data=val # case 3: node to be deleted has only one child else: # choose a child node if cur.left: child=cur.left else: child=cur.right # if the node to be deleted is not a root node, set its parent # to its child if cur!=root: if cur==parent.left: parent.left=child else: parent.right=child else: root=child return root   root=Node(27)root.insertTree(14)root.insertTree(35)root.insertTree(10)root.insertTree(19)root.insertTree(31)root.insertTree(42)root.printTree()print("\n inorder Traversal")root.inorder(root)print("\n preorder Traversal")root.preorder(root)print("\n postorder Traversal")root.postorder(root)print("\n")print(root.findNode(14))print(root.findNode(77))root.deleteNode(root,14)root.inorder(root) |
| --- |

Output

10

14

19

27

31

35

42

inorder Traversal

10->14->19->27->31->35->42->

preorder Traversal

27->14->10->19->35->31->42->

postorder Traversal

10->19->14->31->42->35->27->

14 is found

77Not found

10->19->27->31->35->42->



Here, n is the number of nodes in the tree.

### **Space Complexity**

The space complexity for all the operations is O(n).

Hash Table Implementation

| class HashTable: def \_\_init\_\_(self): self.Max=10 self.array1=[[] for i in range(self.Max)] def calc\_hash(self,key): h=0 for char in key: h=h+ord(char) return h % self.Max def add(self,key,val): h=self.calc\_hash(key) found=False for idx,element in enumerate(self.array1[h]): if len(element)==2 and element[0]==key: self.array1[h][idx]=(key,val) found=True break if not found: self.array1[h].append((key,val)) def get(self,key): h=self.calc\_hash(key) for element in self.array1[h]: if element[0]==key: return element[1]   def del\_item(self,key): h=self.calc\_hash(key) for index,element in enumerate(self.array1[h]): if element[0]==key: del self.array1[h][index] def printhash(self,key,val): h=self.calc\_hash(key) found=False for idx,element in enumerate(self.array1[h]): print('idx :',idx,'element',element[0],element[1],'length',len(element))t=HashTable()t.add('march 6',120)t.add('march 8',130)t.add('march 10',110)t.add('march 17',459)t.array1 t.printhash('march 17',459) |
| --- |

Output

[[],

 [('march 8', 130)],

 [('march 10', 110)],

 [],

 [],

 [],

 [],

 [],

 [],

 [('march 6', 120), ('march 17', 459)]]

idx : 0 element march 6 120 length 2

idx : 1 element march 17 459 length 2

Map implementation (add country and its capitals)

| # implementation of Map ADT using a single listclass Map: # create an empty map instance def \_\_init\_\_(self): self.\_entryList=list()# will return the length of the object def \_\_len\_\_(self): return len(self.\_entryList) # determine if the map contains the given key def \_\_contains\_\_(self,key): ndx=self.\_findPosition(key) return ndx is not None# adds a new entry to the map if the key does not exist# otherwise replace the old value with the new value def add(self,key,value): ndx=self.\_findPosition(key) if ndx is not None: self.\_entryList[ndx].value=value else: entry=MapEntry(key,value) self.\_entryList.append(entry) return True# returns the value associated with the key def valueOf(self,key): ndx=self.\_findPosition(key) assert ndx is not None, "Invalid map key" return self.\_entryList[ndx].value # removes the entry associated with the key def remove(self,key): ndx=self.\_findPosition(key) assert ndx is not None, "Invalid map key" self.\_entryList.pop(ndx)  #return an iterator for traversing the keys in the map def \_\_iter\_\_(self): return iter(self.\_entryList)  #find the index position of a category # if key is not found , none is returned def \_findPosition(self,key): for i in range(len(self)): if self.\_entryList[i].key==key: return i return None def printmap(self): for entry in iter(self): print(entry.key," ", entry.value)# storage class for holding the key /value pairs.class MapEntry: def \_\_init\_\_(self, key, value): self.key=key self.value=valuemap=Map()map.add("India","New Delhi")map.add("Afghanistan","Kabul")map.add("Bangladesh","Dhaka")map.add("Belgium","Brussels")map.add("Canada","Ottawa")map.add("China","Beijing")map.add("Egypt","Cairo")map.printmap()map.remove("China")print('map after removal ---------')print('------------------')map.printmap() |
| --- |

Output

India New Delhi

Afghanistan Kabul

Bangladesh Dhaka

Belgium Brussels

Canada Ottawa

China Beijing

Egypt Cairo

map after removal ---------

------------------

India New Delhi

Afghanistan Kabul

Bangladesh Dhaka

Belgium Brussels

Canada Ottawa

Egypt Cairo

Binary Search algorithm

| # Binary search in python- recursive methoddef binarySearch(a,x,low,high): if high>=low: mid=(low+high)//2 if a[mid]==x: return mid elif a[mid]>x: return binarySearch(a,x,low,mid-1) else: return binarySearch(a,x,mid+1,high) else: return -1a=[3,4,5,6,7,8,9]x=7result=binarySearch(a,x,0,len(a)-1)if result!=-1: print("element is present at index "+str(result))else: print("element not present") |
| --- |

Output

element is present at index 4

Merge sort algorithm

| def merge1(lefthalf,righthalf,A): i=j=k=0 while i < len(lefthalf) and j<len(righthalf): if lefthalf[i] < righthalf[j]: A[k]=lefthalf[i] i=i+1 else: A[k]=righthalf[j] j=j+1 k=k+1 while i < len(lefthalf): A[k]=lefthalf[i] i=i+1 k=k+1 while j<len(righthalf): A[k]=righthalf[j] j=j+1 k=k+1def Mergesort1(A): if len(A)>1: mid=len(A)//2 lefthalf=A[:mid] righthalf=A[mid:] print(lefthalf) print(righthalf) Mergesort1(lefthalf) print("---",lefthalf)  Mergesort1(righthalf) print("---",righthalf) merge1(lefthalf,righthalf,A) print("A=",A) A=[534,246,933,127,277,321,454,565,220]print("unsorted array is")print(A)Mergesort1(A)print(A) |
| --- |

Output

unsorted array is

[534, 246, 933, 127, 277, 321, 454, 565, 220]

[534, 246, 933, 127]

[277, 321, 454, 565, 220]

[534, 246]

[933, 127]

[534]

[246]

--- [534]

--- [246]

A= [246, 534]

--- [246, 534]

[933]

[127]

--- [933]

--- [127]

A= [127, 933]

--- [127, 933]

A= [127, 246, 534, 933]

--- [127, 246, 534, 933]

[277, 321]

[454, 565, 220]

[277]

[321]

--- [277]

--- [321]

A= [277, 321]

--- [277, 321]

[454]

[565, 220]

--- [454]

[565]

[220]

--- [565]

--- [220]

A= [220, 565]

--- [220, 565]

A= [220, 454, 565]

--- [220, 454, 565]

A= [220, 277, 321, 454, 565]

--- [220, 277, 321, 454, 565]

A= [127, 220, 246, 277, 321, 454, 534, 565, 933]

[127, 220, 246, 277, 321, 454, 534, 565, 933]

Minimum and maximum of the given array

| import sysdef FindMax(A,n): if n ==1: return A[0] return max(A[n-1],FindMax(A,n-1))def FindMin(A,n): if n ==1: return A[0] return min(A[n-1],FindMin(A,n-1))A=[]n=int(input("enter the size of the array"))print("enter the element of the array")for i in range(0,n): num=int(input()) A.append(num)print(" Maximum element of the array is", FindMax(A,len(A)))print(" Minimum element of the array is", FindMin(A,len(A))) |
| --- |

Output

enter the size of the array7

enter the element of the array

4

6

9

3

8

2

4

 Maximum element of the array is 9

 Minimum element of the array is 2