RATHINAM TECHNICAL CAMPUS

RATHINAM TECHZONE

EACHANARI, COIMBATORE-641021.

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING (REGIONAL)

RECORD NOTE BOOK

22CS301R DATA STRUCTURES LABORATORY

NAME :

REGISTER NUMBER :

YEAR/SEMESTER :

ACADEMIC YEAR :

RATHINAM TECHNICAL CAMPUS

RATHINAM TECHZONE

EACHANARI, COIMBATORE-641021.

BONAFIDE CERTIFICATE

NAME :

ACADEMIC YEAR :

YEAR/SEMESTER :

BRANCH :

UNIVERSITY REGISTER NUMBER: …….

Certified that this is the bonafide record of work done by the above student in the

_____Laboratory during the year 2022-2023.

Head of the Department **Staff-in-Charge**

Submitted for the Practical Examination held on _____

Internal Examiner **External Examiner**

INDEX

S. No	Date	Experiment Name	Page No:	Marks	Staff Sign

INDEX

S.No	Date	Experiment Name	Page No:	Marks	Staff Sign

Register No: 23106026

EXPT NO :
DATE :

Implement Binary tree

Create a Python program that implements a binary tree data structure. The program should take dynamic input from the user to build the tree. Each input will represent a node, and you need to insert it into its appropriate position within the binary tree structure. You don't need to implement any specific tree traversal algorithms for this task; just focus on the insertion logic.

Example:

Sample Input: 1

5
2 4 1 3 5

Sample Output: 1

Tree structure successfully built!

Sample Input: 2

8
4 2 6 1 3 5 7 8

Sample Output: 2

Tree structure successfully built!

Sample Input: 3

10
8 4 12 2 6 10 14 1 3 5 7 9 11 13 15

Sample Output: 3

Tree structure successfully built!

Input Format:

The first line of input indicates the number of nodes. Subsequent lines contain space-separated integer values representing the nodes.

Output Format:

The program should print 'Tree structure successfully built!' upon successful creation of the binary tree.

Constraints:

Input will always be valid, and the first number represents the total number of nodes to be inserted. Duplicate values are allowed in the tree.

Hint:

Use a class to represent a node in the binary tree. Each node should have data, left child, and right child attributes. Implement an insert method to add new nodes, adhering to the properties of a binary search tree.

Naming Conventions:

Follow Python naming conventions - use CamelCase for class names and lowercase_with_underscores for variables and methods.

Solution

```
class node:
    def __init__ (self,data):
        self.data=data
        self.left=None
        self.right=None
class binarytree:
    def __init__(self):
        self.root=None
    def insert(self,data):
        newnode=node(data)
        if(self.root==None):
            self.root=newnode
            return
        queue=[self.root]
        while queue:
            temp=queue.pop(0)
            if temp.left==None:
                temp.right=newnode
                break
            else:
                queue.append(temp.left)
            if temp.right==None:
                temp.right=newnode
                break
            else:
                queue.append(temp.right)
    def display(self):
        print("Tree structure successfully built!");
tree=binarytree()
m=int(input())
nums=list(map(int,input().split()))
for i in range(m):
    tree.insert(nums[i])
tree.display()
```

Register No: 23106026

Analysis:

Time Complexity	**- O(n)**
Space Complexity	**- O(n)**
Logic Analysis	**- Beginner**
Algorithmic Analysis	**- Beginner**
Code Proficiency	**- Beginner**
Coding Conventions	**- Beginner**
Error Handling	**- Beginner**
Code Reusability	**- Beginner**
Code Accuracy	**- 100%**

Register No: 23106026

EXPT NO :
DATE :

Implement Binary Search tree

Imagine you are building a library management system. You need to store book records efficiently so that searching for a specific book by its ID is fast. Implement a Binary Search Tree in Python to store book records (represented by unique integer IDs) dynamically based on user input. Your program should handle insertion of new book IDs and display the tree's inorder traversal, which represents the sorted order of book IDs.

Example:

Sample Input: 1

5
10
5
15
2
7

Sample Output: 1

2 5 7 10 15

Sample Input: 2

8
50
30
70
20
40
60
80
10

Sample Output: 2

10 20 30 40 50 60 70 80

Sample Input: 3

9
8
3
10

1
6
14
4
7
13Sample Output: 3

1 3 4 6 7 8 10 13 14

Input Format:

The first line contains an integer 'n' representing the number of book IDs. The following 'n' lines each contain a single integer representing a book ID.

Output Format:

Print the space-separated inorder traversal of the constructed BST.

Constraints:

The input will consist of positive integers representing book IDs. Assume all book IDs are unique.

Hint:

Use a class to represent a node in the BST and implement methods for insertion and inorder traversal.

Naming Conventions:

Use descriptive variable names like 'root' for the root of the tree, 'data' for the node's value, 'left' and 'right' for child nodes.

Solution

```
class node:
    def __init__(self,data):
        self.data=data
        self.left=None
        self.right=None
class Bst:
    def __init__(self):
        self.root=None
    def insert(self,data):
        newnode=node(data)
        if self.root==None:
            self.root=newnode
            return
        temp=self.root
        while True:
            if data < temp.data:
                if temp.left==None:
                    temp.left=newnode
                    break
```

```
                else:
                    temp=temp.left
            elif data > temp.data:
                if temp.right==None:
                    temp.right=newnode
                    break
                else:
                    temp=temp.right
            else:
                break
    def display(self,node):
        if node:
            self.display(node.left)
            print(node.data,end=" ")
            self.display(node.right)
tree=Bst()
n=int(input())
for i in range (n):
    s=int(input())
    tree.insert(s)
tree.display(tree.root)
```

Analysis:

Time Complexity	**- O(n)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		

Register No: 23106026

EXPT NO :
DATE :

Singly linked list implementation using python

Your task is to implement a Singly Linked List in Python. A Singly Linked List is a linear data structure where each element (node) points to the next element in the sequence. The last element's next pointer points to null, indicating the end of the list. Your implementation should include the following operations:

1. Insertion:
Insert at the beginning of the list
Insert at the end of the list
Insert after a given node
2. Deletion:
Delete the head node
Delete the tail node
Delete a node with a given value
3. Traversal:
Print all the elements of the linked list

Example:

Sample Input: 1

8
1 10
1 20
2 30
3 25 20
4
6 30
5
4

Sample Output: 1

20 25 10 30
20 25

Sample Input: 2

10
1 15
1 10
2 20
2 25
3 20 30

4
5
4
6 10
4

Sample Output: 2

10 15 20 25
10 15 20
15 20

Sample Input: 3

6
1 5
2 10
4
5
4
6 5

Sample Output: 3

5 10
5

Input Format:

1. The first line contains an integer n (number of operations).
2. The next n lines contain the following operations:
    - 1 x: Insert x at the beginning.
    - 2 x: Insert x at the end.
    - 3 x y: Insert x after the node with value y.
    - 4: Print the current linked list.
    - 5: Delete the tail node.
    - 6 x: Delete the node with value x.

Output Format:

For every 4 operation, print the linked list elements in one line, separated by a space. If the list is empty, print "The list is empty."

Constraints:

1. The number of operations (insertions, deletions, printing) will be within the range [1, 1000]. 2. The values to be inserted will be integers within the range [-1000, 1000].

Hint:

Use a class to represent the Node of the linked list, containing data and a reference to the next node. Another class can represent the LinkedList itself, containing the head of the list and methods for various operations.

Naming Conventions:

Use CamelCase for class names (e.g., `LinkedList`, `Node`) and lowercase with underscores for variable and function names (e.g., `head_node`, `insert_at_end`).

Solution

```
class node:
    def __init__(self,data):
        self.data=data
        self.next=None
class SLL:
    def __init__(self):
        self.head=None
        self.temp=None
    def insert_at_beg(self,data):
        newnode=node(data)
        if self.head==None:
            self.head=newnode
            self.temp=newnode
        else:
            newnode.next=self.head
            self.head=newnode
    def insert_at_end(self,data):
        newnode=node(data)
        self.temp=self.head
        while self.temp.next!=None:
            self.temp=self.temp.next
        self.temp.next=newnode
    def insert_after(self,x,y):
        self.temp=self.head
        while self.temp.next!=None:
            if self.temp.data==y:
                newnode=node(x)
                newnode.next=self.temp.next
                self.temp.next=newnode
                break
            else:
                self.temp=self.temp.next
    def display(self):
        self.temp=self.head
        if self.head!=None:
            while self.temp!=None:
                print(self.temp.data,end=" ")
                self.temp=self.temp.next
            print()
        else:
            print("The list is empty.")
    def delete_tail(self):
        self.temp=self.head
        while self.temp.next.next!=None:
            self.temp=self.temp.next
        self.temp.next=None
    def delete_node(self,x):
```

```
        self.temp=self.head
        if self.head.data==x:
            self.head=self.head.next
        while self.temp.next!=None:
            if self.temp.next.data==x:
                temp=self.temp.next.next
                self.temp.next=temp
            else:
                self.temp=self.temp.next
obj=SLL()
n=int(input())
for i in range(n):
    s=list(map(int,input().split()))
    if s[0]==1:
        obj.insert_at_beg(s[1])
    elif s[0]==2:
        obj.insert_at_end(s[1])
    elif s[0]==3:
        obj.insert_after(s[1],s[2])
    elif s[0]==4:
        obj.display()
    elif s[0]==5:
        obj.delete_tail()
    elif s[0]==6:
        obj.delete_node(s[1])
```

Analysis:

Time Complexity	**- O(n)**	Coding Conventions	**- Intermediate**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Intermediate**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Intermediate**	Code Accuracy	**- 100%**
Code Proficiency	**- Intermediate**		

Register No: 23106026

EXPT NO :
DATE :

List implementation of Circular Queue ADTs

Imagine you're designing a system to manage song requests in a music streaming app. Users can add songs to a queue, and the system plays them in the order they were added. However, to avoid extremely long wait times, you've decided to limit the queue's size. Your task is to implement this song request queue using a circular queue data structure in Python. Since the number of song requests can vary greatly, your implementation should dynamically adjust the queue's size as needed.

Example:

Sample Input: 1

5
ENQUEUE Faded
ENQUEUE Closer
ENQUEUE Let Her Go
DEQUEUE
ENQUEUE One Last Time

Sample Output: 1

Faded added to the queue
Closer added to the queue
Let Her Go added to the queue
Faded is removed from the queue
One Last Time added to the queue

Sample Input: 2

7
ENQUEUE Shape of You
ENQUEUE Despacito
ENQUEUE Believer
DEQUEUE
DEQUEUE
ENQUEUE Thunder
ENQUEUE Happy

Sample Output: 2

Shape of You added to the queue
Despacito added to the queue
Believer added to the queue
Shape of You is removed from the queue
Despacito is removed from the queue

Thunder added to the queue
Happy added to the queue

Sample Input: 3

4
ENQUEUE Song A
ENQUEUE Song B
DEQUEUE
DEQUEUE

Sample Output: 3

Song A added to the queue
Song B added to the queue
Song A is removed from the queue
Song B is removed from the queue

Input Format:

The first line of input indicates the number of operations. Subsequent lines each contain an operation: 'ENQUEUE ' to add a song or 'DEQUEUE' to remove a song.

Output Format:

For each operation, print a message indicating the result: 'Queue is Empty' if DEQUEUE is called on an empty queue, ' added to the queue' after adding a song, or ' is removed from the queue' after removing a song.

Constraints:

The maximum number of songs in the queue at any given time will not exceed 1000.

Hint:

Use a fixed-size list and manage its elements in a circular fashion using modulo operation for index calculation. Use front and rear pointers to keep track of the start and end of the queue.

Naming Conventions:

Use descriptive variable names like 'queue', 'front', 'rear', 'size' etc.

Solution

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

```
        return (self.data)
class Queue:
    def __init__(self):
        self.front=None
        self.rear=None
    def enqueue(self,data):
        newnode=node(data)
        if self.rear is None:
            self.front=newnode
            self.rear=newnode
            self.rear.next=self.front
        else:
            self.rear.next=newnode
            self.rear=newnode
            self.rear.next=self.front
        print(f"{newnode} added to the queue")
    def dequeue(self):
        if self.front is None:
            print("Queue is empty")
            return
        removed=self.front
        if self.front==self.rear:
            self.front=None
            self.rear=None
        else:
            self.front=self.front.next
            self.rear.next=self.front
        print(f"{removed} is removed from the queue")
obj=Queue()
n=int(input())
for i in range(n):
    op=input().split()
    oper=op[0]
    s=' '.join(op[1:])
    if oper=="ENQUEUE":
        obj.enqueue(s)
    elif oper=="DEQUEUE":
        obj.dequeue()
```

Analysis:

Register No: 23106026

Time Complexity	**- O(1)**
Space Complexity	**- O(n)**
Logic Analysis	**- Beginner**
Algorithmic Analysis	**- Beginner**
Code Proficiency	**- Beginner**
Coding Conventions	**- Beginner**
Error Handling	**- Beginner**
Code Reusability	**- Beginner**
Code Accuracy	**- 100%**

Register No: 23106026

EXPT NO :
DATE :

List implementation of Stack ADTs

Implement a Stack Abstract Data Type (ADT) in Python using a list as the underlying data structure. Your implementation should handle a dynamic number of inputs, allowing the user to perform stack operations like push, pop, peek, and check if the stack is empty.

Example:

Sample Input: 1

5
push 1
push 2
peek
pop
peek

Sample Output: 1

2
1

Sample Input: 2

7
push 10
push 20
push 30
pop
peek
push 40
peek

Sample Output: 2

20
40

Sample Input: 3

9
push 100
push 200

push 300
pop
pop
peek
push 400
pop
peek

Sample Output: 3

100
100

Input Format:

The first line contains an integer 'n' representing the number of operations. The following 'n' lines each contain an operation in the format: 'operation' or 'operation value', where 'operation' can be 'push', 'pop', 'peek', or 'isEmpty'.

Output Format:

For each 'peek' operation, print the top element of the stack on a new line. For 'pop' operations, output nothing if the stack is empty.

Edge Case scenario:
if the stack correctly handles multiple pop operations and returns `True` when it's empty.(captured in hidden test case)

Constraints:

The input will always be valid and follow the specified format. The number of inputs (operations) will be a positive integer.

Hint:

Use a Python list to store stack elements. Implement methods for push (append to the list), pop (remove from the end of the list), peek (access the last element), and check if the list is empty.

Naming Conventions:

Use descriptive variable names like 'stack', 'element', 'operation', etc.

Solution

```
class stack:
    def __init__(self):
        self.stack=[]
    def is_empty(self):
```

```
        return len(self.stack)==0
    def push(self,data):
        self.stack.append(data)
    def pop(self):
        if self.is_empty():
            print("True")
        return self.stack.pop()
    def peek(self):
        if self.is_empty():
            return "Stack is empty!"
        return self.stack[-1]
    def display(self):
        return self.stack
obj=stack()
n=int(input())
for i in range(n):
    l=list(map(str,input().split(" ")))
    if l[0]=="push":
        obj.push(int(l[1]))
    elif l[0]=="pop":
        obj.pop()
    elif l[0]=="peek":
        val=obj.peek()
        print(val)
```

Analysis:

Time Complexity	**- O(1)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		

Register No: 23106026

EXPT NO :
DATE :

Implement Graph Traversal Technique (BFS).

You are tasked with implementing a Python program that performs Breadth-First Search (BFS) on a graph. Your program should take a graph represented as an adjacency list and a starting node as input. The goal is to traverse the graph level by level from the starting node, visiting all reachable nodes.

Example:

Sample Input: 1

4
1 2
1 3
2 4
1

Sample Output: 1

1 2 3 4

Sample Input: 2

7
1 2
1 3
2 4
2 5
3 6
3 7
1

Sample Output: 2

1 2 3 4 5 6 7

Sample Input: 3

10
1 2
1 3
2 4
3 5
3 6

4 7
5 8
6 9
7 10
1

Sample Output: 3

1 2 3 4 5 6 7 8 9 10

Input Format:

The input consists of multiple lines. The first line contains an integer 'V' representing the number of vertices in the graph. The subsequent 'V' lines each contain two space-separated integers 'u' and 'v', indicating an edge between nodes 'u' and 'v'. The last line contains an integer representing the starting node.

Output Format:

Print the nodes in the order they are visited during the BFS traversal, separated by spaces.

Constraints:

The graph will have a maximum of 100 nodes.

Hint:

Use a queue to store nodes at each level and mark visited nodes to avoid cycles.

Naming Conventions:

Variable names should be descriptive (e.g., 'graph', 'start_node', 'queue').

Solution

```
def adjacencyList(ver,edge):
    li=[i+1 for i in range(ver)]
    di={}
    for i in li:
        di[i]=[]
    for i,j in edge:
        di[i].append(j)
        di[j].append(i)
    return di
def bfs(adjLis):
    st=int(input())
    que=[st]
    vis=[]
    while(que):
        x=que.pop(0)
        vis.append(x)
        for i in adjLis[x]:
```

```
            if i not in que and i not in vis:
                que.append(i)
    print(*vis)
n=int(input())
li=[]
for i in range(n-1):
    li.append(list(map(int,input().split())))
aLi=adjacencyList(n,li)
bfs(aLi)
```

Analysis:

Time Complexity	**- O(V + E)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(V)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		

Register No: 23106026

EXPT NO :
DATE :

Implement Graph using Adjacency List.

Imagine you're organizing a social network where users can be friends. Your task is to represent this network efficiently using an adjacency list in Python. Each person will be a node in our graph, and a friendship between two people will be represented by an edge. An adjacency list is a way to store this information by using a dictionary: the keys are the nodes, and the value associated with each key is a list of nodes directly connected to it (its neighbors). Your goal is to write a Python program that allows you to add users (nodes) and friendships (edges) to this social network graph. Since it's an undirected graph, if User A is friends with User B, then User B is also friends with User A.

Example:

Sample Input: 1

3
AddNode 0
AddNode 1
AddEdge 0 1

Sample Output: 1

{0: [1], 1: [0]}

Sample Input: 2

5
AddNode 0
AddNode 1
AddEdge 0 1
AddNode 2
AddEdge 1 2

Sample Output: 2

{0: [1], 1: [0, 2], 2: [1]}

Sample Input: 3

8
AddNode 0
AddNode 1
AddNode 2
AddEdge 0 1

AddEdge 1 2
AddNode 3
AddEdge 0 3
AddEdge 1 3

Sample Output: 3

{0: [1, 3], 1: [0, 2, 3], 2: [1], 3: [0, 1]}

Input Format:

The input will consist of multiple lines. The first line contains an integer 'N' representing the number of operations. The following 'N' lines will each contain an operation, either 'AddNode x' (where x is the node value) or 'AddEdge x y' (where x and y are the nodes to connect).

Output Format:

The output should be a string representation of the adjacency list, with nodes as keys and their corresponding neighbor lists as values, enclosed in curly braces '{}'.

Constraints:

Assume input nodes are non-negative integers. 'AddEdge' should handle the undirected nature of the graph.

Hint:

Use a dictionary in Python where keys represent nodes and values are lists of adjacent nodes. Implement 'AddNode' to add keys and 'AddEdge' to update adjacency lists.

Naming Conventions:

Use descriptive variable names like 'graph', 'node', 'neighbor' for better code readability.

Solution

```
def Add_edge(di,edge):
    u,v=edge
    di[u].append(v)
    di[v].append(u)
def Add_node(di,ver):
    di[ver]=[]
def display(di):
    print(di)
di={}
for i in range(int(input())):
    li=input().split()
    if li[0]=="AddNode":
        Add_node(di,int(li[1]))
    elif li[0]=="AddEdge":
        Add_edge(di,[int(li[1]),int(li[2])])
display(di)
```

Register No: 23106026

Analysis:

Time Complexity	**- O(V + E)**
Space Complexity	**- O(V + E)**
Logic Analysis	**- Beginner**
Algorithmic Analysis	**- Beginner**
Code Proficiency	**- Beginner**
Coding Conventions	**- Beginner**
Error Handling	**- Beginner**
Code Reusability	**- Beginner**
Code Accuracy	**- 100%**

Register No: 23106026

EXPT NO :
DATE :

Implement Graph Traversal Tech

Imagine you're navigating a maze. You can only move forward until you hit a dead end, then you must backtrack and try another path. This is the essence of Depth First Search (DFS). Your challenge is to write a Python program that performs DFS on a graph, represented using an adjacency list. Given a starting node, your program should explore the graph as far as possible along each branch before backtracking.

Example:

Sample Input: 1

```
5 5
0 1
0 2
1 3
1 4
2 4
0
```

Sample Output: 1

```
0 1 3 4 2
```

Sample Input: 2

```
6 7
0 1
0 2
1 3
1 4
2 5
3 5
4 5
0
```

Sample Output: 2

```
0 1 3 5 2 4
```

Sample Input: 3

```
7 6
0 1
0 2
```

1 3
2 4
3 5
4 6
0

Sample Output: 3

0 1 3 5 2 4 6

Input Format:

The first line contains two integers V and E separated by space, representing the number of vertices and edges in the graph. The next E lines each contain two integers u and v, representing an edge connecting vertex u and v. The last line contains a single integer, the starting node for DFS.

Output Format:

Print the nodes in the order they are visited during the DFS traversal separated by space.

Constraints:

The graph will have at most 100 nodes (0-indexed). Input edges will be valid and non-directional.

Hint:

Use a stack (or recursion) to keep track of the nodes to visit. Mark visited nodes to avoid cycles.

Naming Conventions:

Use meaningful variable names like 'graph', 'visited', 'stack', etc.

Solution

```
def adjacencyTraversal(ver,edge):
    li=[]
    for i in range(ver):
        li.append([0]*ver)
    for x,y in edge:
        li[x][y]=1
        li[y][x]=1
    return li
def dfs(Mli,st,n):
    stac=[st]
    vis=[]
    while(stac):
        ini=stac.pop()
        if ini not in vis:
            vis.append(ini)
            for j in range(n-1,-1,-1):
                if Mli[ini][j]==1:
                    if j not in vis:
```

```
                        stac.append(j)
    print(*vis)
ver,edge=map(int,input().split())
edj=[]
for i in range(edge):
    edj.append(list(map(int,input().split())))
adj=adjacencyTraversal(ver,edj)
st=int(input())
dfs(adj,st,ver)
```

Analysis:

Time Complexity	- **O(V + E)**
Space Complexity	- **O(V)**
Logic Analysis	- **Beginner**
Algorithmic Analysis	- **Beginner**
Code Proficiency	- **Beginner**
Coding Conventions	- **Beginner**
Error Handling	- **Beginner**
Code Reusability	- **Beginner**
Code Accuracy	- **100%**

Register No: 23106026

EXPT NO :
DATE :

Implement Dijkstra’s Algorithm.

Implement Dijkstra's algorithm to find the shortest path between two nodes in a graph. The graph will be represented as an adjacency list, where each key is a node and its value is a list of its neighbors and the corresponding edge weights. You need to return the shortest distance from the source node to the destination node. If there is no path between the source and destination node, return -1.

Example:

Sample Input: 1

4 4 1 2 1 1 3 4 2 3 2 3 4 3 1 4

Sample Output: 1

6

Sample Input: 2

5 6 1 2 2 1 4 1 2 3 3 2 5 4 4 3 2 4 5 3 1 5

Sample Output: 2

4

Sample Input: 3

6 7 1 2 5 1 3 2 2 4 1 3 4 3 3 5 6 4 6 4 5 6 2 1 6

Sample Output: 3

9

Input Format:

The first line contains two integers, N and M, representing the number of nodes and edges, respectively. The next M lines contain three integers each, representing an edge between two nodes and its weight. The last line contains two integers, representing the source and destination nodes.

Output Format:

An integer representing the shortest distance between the source and destination nodes, or -1 if no path exists.

Constraints:

1 <= number of nodes <= 10^5, 1 <= number of edges <= 10^5, 1 <= edge weight <= 10^4

Hint:

Use a priority queue to keep track of the nodes with the shortest distances from the source node.

Naming Conventions:

Variable names should be descriptive and follow Python conventions (e.g., use snake_case).

Solution

```
import heapq
import sys
from collections import defaultdict

def dijkstra(graph,source,destination):
    priority_queue=[]
    shortest_distance={node: float('inf') for node in graph}
    shortest_distance[source]=0
    heapq.heappush(priority_queue,(0,source))

    while priority_queue:
        current_distance,current_node=heapq.heappop(priority_queue)
        if current_node==destination:
            return current_distance
        if current_distance>shortest_distance[current_node]:
            continue
        for neighbor,weight in graph[current_node]:
            distance=current_distance+weight
            if distance<shortest_distance[neighbor]:
                shortest_distance[neighbor]=distance
                heapq.heappush(priority_queue,(distance,neighbor))
    return -1

def main():
    input_data=input().strip().split()
    index=0
    N=int(input_data[index]);index+=1
    M=int(input_data[index]);index+=1

    graph=defaultdict(list)
    for _ in range(M):
        u=int(input_data[index]);index+=1
        v=int(input_data[index]);index+=1
        weight=int(input_data[index]);index+=1
        graph[u].append((v,weight))
        graph[v].append((u,weight))
    source=int(input_data[index]);index+=1
    destination=int(input_data[index])

    result=dijkstra(graph,source,destination)
    print(result)
if __name__ == "__main__":
    main()
```

Analysis:

Time Complexity - **O((V + E) log V)**	Coding Conventions - **Intermediate**
Space Complexity - **O(V)**	Error Handling - **Beginner**
Logic Analysis - **Intermediate**	Code Reusability - **Beginner**
Algorithmic Analysis - **Intermediate**	Code Accuracy - **100%**
Code Proficiency - **Intermediate**	

Register No: 23106026

EXPT NO :

DATE :

Implement Open Addressing (Linear Probing & Quadratic Probing)

Implement a Python program that handles hash collisions in a hash table using open addressing with both linear probing and quadratic probing techniques. **Open Addressing** Open addressing is a method of collision resolution in hash tables. In open addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys. When a new element is to be inserted, the hash table is examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. **Linear Probing** Linear probing is a scheme used in open addressing for resolving collisions in hash tables. In linear probing, we linearly search for the next empty slot in the table. **Quadratic Probing** Instead of using a constant "skip" value like linear probing, quadratic probing uses a quadratic function to calculate the skip value based on the number of attempts made to find an empty slot.

Example:

Sample Input: 1

```
8
10
20
30
40
50
60
70
80
```

Sample Output: 1

```
Hash Table using Linear Probing:
40
80
10
50
20
60
30
70

Hash Table using Quadratic Probing:
40
80
10
50
20
```

60
30
70

Sample Input: 2

10
23
43
12
78
54
87
98
11
32
45

Sample Output: 2

Hash Table using Linear Probing:
32
11
12
23
43
54
45
87
78
98

Hash Table using Quadratic Probing:
45
11
12
23
43
54
32
87
78
98

Sample Input: 3

12
5
15

25
35
45
55
65
75
85
95
105
115

Sample Output: 3

Hash Table using Linear Probing:
95
25
85
15
75
5
65
55
115
45
105
35

Hash Table using Quadratic Probing:
95
25
85
15
75
5
65
55
115
45
105
35

Input Format:

The first line of input represents the number of elements to be inserted into the hash table. Subsequent lines will contain the elements themselves.

Output Format:

Print the hash table content after inserting all the elements using both Linear Probing and Quadratic Probing. Each element should be printed on a new line along with its index.

Constraints:

1 <= Number of Inputs <= 10^5 0 <= Each input element <= 10^9

Hint:

For Linear Probing: h(x, i) = (h'(x) + i) % m For Quadratic Probing: h(x, i) = (h'(x) + i*i) % m where, h'(x) is the hash function, i is the attempt number (starting from 0), and m is the size of the hash table.

Naming Conventions:

Adhere to standard Python variable naming conventions.

Solution

```
size=int(input())
elements=[]
print()

hash_table_linear=[None]*size
hash_table_quadratic=[None]*size

for i in range(size):
    elements.append(int(input()))

def linear_probing(hash_table,element):
    index=element%size
    i=1
    while hash_table[index] is not None:
        index=(index+i)%size
        i+=1
    hash_table[index]=element

def quadratic_probing(hash_table,element):
    index=element%size
    i=1
    while hash_table[index] is not None:
        index=(index + i * i)%size
        i+=1
    hash_table[index]=element

print("Hash Table using Linear Probing: ")
for element in elements:
    linear_probing(hash_table_linear,element)

for i in range(size):
    print(f"{hash_table_linear[i]}  ")

print("\nHash Table using Quadratic Probing: ")
for element in elements:
    quadratic_probing(hash_table_quadratic,element)

for i in range(size):
    print(f"{hash_table_quadratic[i]}  ")
```

Register No: 23106026

Analysis:

Time Complexity	**- O(n)**	Coding Conventions	**- Intermediate**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Intermediate**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Intermediate**	Code Accuracy	**- 100%**
Code Proficiency	**- Intermediate**		

Register No: 23106026

EXPT NO :
DATE :

Linked List implementation of Balancing of Symbol using Stack

You are tasked with implementing a program to check the balance of symbols in a given expression using a stack data structure implemented as a linked list. Your program should handle parentheses '()', square brackets '[]', and curly braces '{}', ensuring that each opening symbol has a corresponding closing symbol in the correct order. For example, the expression "{ [( )] }" is balanced, while "{ [( ] }" is not. The input will be provided dynamically.

Example:

Sample Input: 1

3
{[()]}
{[(])}
((()))

Sample Output: 1

Balanced
Not Balanced
Balanced

Sample Input: 2

5
{}{}{}
[]{}()
{[]
([{}])
(([{}))

Sample Output: 2

Balanced
Balanced
Not Balanced
Balanced
Not Balanced

Sample Input: 3

7
{{{{}}}}((())([[[]]]))
{[](([{{([{}])}}]))}
{[(])}
([{}])
(([{}))
{[()][({})]{}]}
{[](([{{([{}])}}]))}]]

Sample Output: 3

Not Balanced
Balanced
Not Balanced
Balanced
Not Balanced
Not Balanced
Not Balanced

Input Format:

The first line contains an integer N representing the number of expressions. The following N lines each contain an expression.

Output Format:

For each expression, print "Balanced" if the symbols are balanced, otherwise print "Not Balanced" on a new line.

Constraints:

The input expression will consist of printable ASCII characters. The maximum length of the expression is 1000.

Hint:

Use a linked list to implement the stack. Iterate through the expression; push opening symbols onto the stack and pop them off when encountering corresponding closing symbols. Ensure the stack is empty at the end for a balanced expression.

Naming Conventions:

Use descriptive variable names like 'expression', 'stack', 'topNode', etc.

Solution

```
def check(s):
    stack=[]
    for char in s:
        if char in ["{","[","("]:
```

```
            stack.append(char)
        else:
            if not stack:
                return False
            val=stack.pop()
            if val=="(":
                if char!=")":
                    return False
            if val=="{":
                if char!="}":
                    return False
            if val=="[":
                if char!="]":
                    return False
    if stack:
        return False
    return True
n=int(input())
for i in range(n):
    s=input()
    if(check(s)):
        print("Balanced")
    else:
        print("Not Balanced")
```

Analysis:

Time Complexity	**- O(n)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		

Register No: 23106026

EXPT NO :
DATE :

Linked List implementation of Infix to Postfix Conversion using Stack

Create a Python program that implements the infix-to-postfix conversion algorithm using a linked list for the postfix expression and a stack data structure. The program should accept dynamic input, allowing the user to enter infix expressions of varying lengths and complexities.

Example:

Sample Input: 1

a+b*c-d/e

Sample Output: 1

abc*+de/-

Sample Input: 2

(a+b)*(c-d)/e

Sample Output: 2

ab+cd-*e/

Sample Input: 3

a+b*(c^d-e)^(f+g*h)-i

Sample Output: 3

abcd^e-fgh*+^+i-

Input Format:

The input will be a single line string representing the infix expression.

Output Format:

The output will be a single line string representing the postfix expression.

Constraints:

The input infix expressions will contain only single-letter variables, operators (+, -, *, /, ^), and parentheses ((,)).

Hint:

Use a stack to store operators and parentheses. Process the infix expression character by character, appending operands directly to the postfix expression and handling operators based on precedence.

Naming Conventions:

Follow standard Python naming conventions: use lowercase with underscores for variable and function names (e.g., `infix_expression`, `push_to_stack`).

Solution

```
def prec(c):
    if c == "^":
        return 2
    elif c == "/" or c == "*":
        return 1
    elif c == "+" or c == "-":
        return 0
    else:
        return -1
def infixtopostfix(s):
    postfix=[]
    stack=[]
    for i in range(len(s)):
        c=s[i]
        if("a"<=c<="z")or("A"<=c<="Z")or("0"<=c<="9"):
            postfix.append(c)
        elif c=="(":
            stack.append(c)
        elif c==")":
            while stack and stack[-1]!="(":
                postfix.append(stack.pop())
            stack.pop()
        else:
            while stack and (prec(s[i])<prec(stack[-1]) or
prec(s[i])==prec(stack[-1])):
                postfix.append(stack.pop())
            stack.append(s[i])
    while stack:
        postfix.append(stack.pop())
    print("".join(postfix))
s=input()
infixtopostfix(s)
```

Analysis:

Register No: 23106026

Time Complexity - **O(n)**	Coding Conventions - **Beginner**
Space Complexity - **O(n)**	Error Handling - **Beginner**
Logic Analysis - **Intermediate**	Code Reusability - **Beginner**
Algorithmic Analysis - **Intermediate**	Code Accuracy - **100%**
Code Proficiency - **Intermediate**	

Register No: 23106026

EXPT NO :
DATE :

Linked List implementation of Queue ADTs.

Implement a Queue abstract data type (ADT) in Python using a linked list as the underlying data structure. Your implementation should include the following methods: 1. `enqueue(data)`: Adds an element with `data` to the rear of the queue. 2. `dequeue()`: Removes and returns the element at the front of the queue. Raises an exception if the queue is empty. 3. `front()`: Returns the element at the front of the queue without removing it. Raises an exception if the queue is empty. 4. `is_empty()`: Returns `True` if the queue is empty, `False` otherwise. 5. `size()`: Returns the number of elements in the queue.

Example:

Sample Input: 1

```
4
enqueue 10
enqueue 20
dequeue
size
```

Sample Output: 1

```
10
1
```

Sample Input: 2

```
3
enqueue 5
enqueue 15
dequeue
```

Sample Output: 2

```
5
```

Sample Input: 3

```
3
enqueue 30
dequeue
size
```

Sample Output: 3

30
0

Input Format:

The first line of input contains an integer 'n' representing the number of operations. The following 'n' lines each contain an operation to be performed on the queue. Each operation is specified by a keyword followed by optional arguments (e.g., 'enqueue 5').

Output Format:

For each 'dequeue' and 'front' operation, print the returned value on a new line. For each 'size' operation, print the size of the queue on a new line.

Constraints:

The input will always be valid and follow the specified format. The maximum size of the queue is not specified.

Hint:

Use a class to represent the queue and a separate class to represent the nodes in the linked list. Each node should store the data and a reference to the next node.

Naming Conventions:

Use descriptive names like 'Queue' for the queue class, 'Node' for the node class, 'data' for the data stored in a node, 'front' for the front of the queue, 'rear' for the rear of the queue, etc.

Solution

```
class node:
    def __init__(self,data):
        self.data=data
        self.next=None
class Queue:
    def __init__(self):
        self.front=None
        self.rear=None
        self._size=0
    def enqueue(self,data):
        newnode=node(data)
        if self.is_empty():
            self.front=self.rear=newnode
        else:
            self.rear.next=newnode
            self.rear=newnode
        self._size+=1
    def dequeue(self):
        if self.is_empty():
```

```
            return "Queue is empty"
        dequeued_value=self.front.data
        self.front=self.front.next
        if self.front is None:
            self.rear=None
        self._size-=1
        return dequeued_value
    def front_value(self):
        if self.is_empty():
            return"Queue is empty"
        return self.front.data
    def is_empty(self):
        return self.front is None
    def size(self):
        return self._size
queue=Queue()
n=int(input())
for _ in range(n):
    oper=input().split()
    if oper[0]=="enqueue":
        queue.enqueue(int(oper[1]))
    elif oper[0]=="dequeue":
        print(queue.dequeue())
    elif oper[0]=="front":
        print(queue.front_value())
    elif oper[0]=="is_empty":
        print(queue.is_empty)
    elif oper[0]=="size":
        print(queue.size())
```

Analysis:

Time Complexity	**- O(1)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		

Register No: 23106026

EXPT NO :
DATE :

List implementation of Double ended Queue ADTs

Implement a double-ended queue (deque) data structure in Python using a list as the underlying storage mechanism. Your implementation should handle dynamic input, meaning users can add or remove elements from both ends of the queue. The program should support the following operations: 1. `append(value)`: Adds an element to the rear of the deque. 2. `appendleft(value)`: Adds an element to the front of the deque. 3. `pop()`: Removes and returns the element at the rear of the deque. Raises an exception if the deque is empty. 4. `popleft()`: Removes and returns the element at the front of the deque. Raises an exception if the deque is empty. 5. `peek()`: Returns the element at the rear of the deque without removing it. Raises an exception if the deque is empty. 6. `peekleft()`: Returns the element at the front of the deque without removing it. Raises an exception if the deque is empty. 7. `is_empty()`: Returns `True` if the deque is empty, `False` otherwise. 8. `size()`: Returns the number of elements in the deque.

`0Example:

Sample Input: 1

```
5
append 1
append 2
appendleft 0
peekleft
pop
```

Sample Output: 1

```
0
2
```

Sample Input: 2

```
4
append 10
appendleft 20
pop
size
```

Sample Output: 2

```
10
1
```

Sample Input: 3

3
appendleft 30
peekleft
size
Sample Output: 3

30

1

Input Format:

The first line of input indicates the number of operations to be performed. Subsequent lines each contain a single operation followed by its input (if any), separated by a space.

Output Format:

For each 'peek', 'pop', 'popleft', or 'size' operation, output the result on a new line. If an operation results in an error, output the error message.

Constraints:

The maximum size of the deque is limited by the available memory.

Hint:

Utilize the list methods `append()`, `insert()`, `pop()`, and `pop(0)` to efficiently implement the deque operations. Remember to handle edge cases like popping from an empty deque.

Naming Conventions:

Use descriptive variable names like 'deque', 'value', etc.

Solution

```
class queue:
    def __init__(self):
        self.arr=[]
    def enrear(self,val):
        return self.arr.append(val)
    def derear(self):
        return self.arr.pop()
    def enfront(self,val):
        return self.arr.insert(0,val)
    def defront(self):
        return self.arr.pop(0)
    def rearpeek(self):
        return self.arr[-1]
    def frontpeek(self):
        return self.arr[0]
    def display(self):
        print(len(self.arr))
```

```
res=queue()
n=int(input())
for i in range(n):
    oper=input().split()
    if len(oper)>=2:
        op=oper[0]
        val=oper[1]
    else:
        op=oper[0]
    if op=="append":
        res.enrear(val)
    elif op=="appendleft":
        res.enfront(val)
    elif op=="pop":
        r=res.derear()
        print(r)
    elif op=="popleft":
        r=res.defront()
    elif op=="peek":
        r=res.rearpeek()
        print(r)
    elif op=="peekleft":
        r=res.frontpeek()
        print(r)
    elif op=="size":
        res.display()
```

Analysis:

Time Complexity	**- O(1)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		

Register No: 23106026

EXPT NO :
DATE :

Linked List implementation of Stack ADTs

You need to implement a Stack Abstract Data Type (ADT) using a Linked List in Python. Your program should handle a dynamic number of inputs. The Stack ADT should support the following operations:

1. **Push (x):** Adds an element 'x' to the top of the stack.

2. **Pop ():** Removes and returns the element at the top of the stack. If the stack is empty, it should indicate an error (e.g., return a specific message or raise an exception).

3. **Peek():** Returns the element at the top of the stack without removing it. If the stack is empty, handle it appropriately.

4. **isEmpty():** Checks if the stack is empty. Returns True if empty, False otherwise.

5. **Size():** Returns the number of elements in the stack.

Example:

Sample Input: 1

```
4
push 10
push 20
peek
pop
```

Sample Output: 1

```
20
20
```

Sample Input: 2

```
3
push 5
pop
isEmpty
```

Sample Output: 2

```
5
True
```

Sample Input: 3

3
push 15
peek
isEmpty

Sample Output: 3

15
False

Input Format:

The first line of input contains an integer 'n' indicating the number of operations. The following 'n' lines contain operations in the format: 'operationName value' (for push) or 'operationName' (for other operations).

Output Format:

The output should print the results of each 'peek', 'pop', 'isEmpty', and 'size' operation on separate lines, as demonstrated in the sample input-output pairs.

Constraints:

The input will always be valid and follow the specified format. The number of operations (n) will be within a reasonable range (1 <= n <= 1000).

Hint:

Use a Node class to represent elements in the linked list. Each node should store the data and a reference to the next node. The stack can be represented by a head pointer, initially pointing to None.

Naming Conventions:

Use descriptive variable names like 'Node', 'data', 'next', 'head', 'push', 'pop', 'peek', etc.

Solution

```
class node:
    def __init__(self,data):
        self.data=data
        self.next=None
class stack:
    def __init__(self):
        self.top=None
    def push(self,x):
        newnode=node(x)
        newnode.next=self.top
        self.top=newnode
    def pop(self):
```

```
        if self.isEmpty():
            return "True"
        popped_value=self.top.data
        self.top=self.top.next
        return popped_value
    def peek(self):
        if self.isEmpty():
            return "Stack is empty"
        return self.top.data
    def isEmpty(self):
        return self.top is None
    def size(self):
        count=0
        current=self.top
        while current:
            current+=1
            current=current.next
        return count
obj=stack()
n=int(input())
for _ in range(n):
    oper=input().split()
    if oper[0]=="push":
        obj.push(int(oper[1]))
    elif oper[0]=="pop":
        print(obj.pop())
    elif oper[0]=="peek":
        print(obj.peek())
    elif oper[0]=="isEmpty":
        print(obj.isEmpty())
    elif oper[0]=="size":
        print(obj.size())
```

Analysis:

Time Complexity	**- O(1)**	Coding Conventions	**- Beginner**
Space Complexity	**- O(n)**	Error Handling	**- Beginner**
Logic Analysis	**- Beginner**	Code Reusability	**- Beginner**
Algorithmic Analysis	**- Beginner**	Code Accuracy	**- 100%**
Code Proficiency	**- Beginner**		