**LINKED LIST**

1)You are working on a real-time inventory management system where you need to maintain a list of product IDs in a linked list. As new products are added to the inventory, their IDs are appended at the end of the list. You have been tasked with writing a program to handle this operation.

Write a program to insert a new product ID at the end of the linked list. After the completion of insertion, print the entire linked list to confirm that the new ID has been added.

**Input Format**

1. The first line contains an integer N, the number of product IDs to be initially added to the linked list.
2. The second line contains N space-separated integers, representing the product IDs.

**Constraints**

NA

**Output Format**

After inserting the product ID, print the current linked list, with elements separated by a space.

**Sample Input 0**

5

101 102 103 104 105

**Sample Output 0**

101 102 103 104 105

**Explanation 0**

Initially the linked list is empty. The first product ID 101 is inserted at the end, and the list becomes 101. The second product ID 102 is inserted at the end, and the list becomes 101 102. The third product ID 103 is inserted at the end, and the list becomes 101 102 103. The fourth product ID 104 is inserted at the end, and the list becomes 101 102 103 104. The fifth product ID 105 is inserted at the, and the list becomes 101 102 103 104 105.

**Sample Input 1**

3

101 102 103

**Sample Output 1**

101 102 103

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def display(self):

if self.first is None:

return

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

for data in datas:

ll.add\_nodes(data)

ll.display()

2) You are developing a simple task management application where tasks are represented by unique IDs. Users can add new tasks at the beginning of the task list for quick access. You need to write a program to handle the insertion of task IDs at the beginning of a linked list. After all insertions are completed, the final list of task IDs should be printed to confirm the additions.

Write a program to insert new task IDs at the beginning of the linked list. After all insertions, print the entire linked list to show the updated list of task IDs.

**Input Format**

1. The first line contains an integer N, the number of task IDs to be initially added to the linked list.
2. The second line contains N space-separated integers representing the task IDs.

**Constraints**

NA

**Output Format**

After completing all insertions, print the current linked list with elements separated by a space.

**Sample Input 0**

5

201 202 203 204 205

**Sample Output 0**

205 204 203 202 201

**Explanation 0**

Initially the linked list is empty. The first task ID 201 is inserted at the beginning, and the list becomes 201. The second task ID 202 is inserted at the beginning, and the list becomes 202 201. The third task ID 203 is inserted at the beginning, and the list becomes 203 202 201. The fourth task ID is inserted at the beginning, and the list becomes 204 203 202 201. The fifth task ID is inserted at the beginning, and the list becomes 205 204 203 202 201.

**Sample Input 1**

3

10 20 30

**Sample Output 1**

30 20 10

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

p.next = self.first

self.first = p

def display(self):

if self.first is None:

return

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

for data in datas:

ll.add\_nodes(data)

ll.display()

==========================================================================

3) You are developing a task tracker for a project where tasks are assigned unique IDs. These tasks are arranged in a linked list, based on the order in which they need to be completed. Sometimes, a new task needs to be inserted at a specific position in the list, ensuring it is prioritized correctly. Your task is to write a program to insert a new task at a given position in the task list and print the updated list after the insertion.

Write a program that inserts a new task (represented by a unique task ID) at a specific position in the task list. After the insertion, print the updated list of task IDs. If the entered position value is out the range, print the message as “Position out of range”.

**Input Format**

1. The first line contains an integer N, the number of task IDs initially added to the linked list.
2. The second line contains N space-separated integers representing the task IDs.
3. The third line contains an integer P, the position at which the new task ID should be inserted (1-based index).
4. The fourth line contains an integer M, the new task ID to be inserted.

**Constraints**

The value of P will always be between 1 and N+1 (both inclusive).

**Output Format**

Print the updated task list with the elements separated by a space.

**Sample Input 0**

4

101 102 103 104

2

999

**Sample Output 0**

101 999 102 103 104

**Explanation 0**

The task list is initialized with 4 task IDs: 101 102 103 104. The new task ID 999 is inserted at position 2 (between 101 and 102). The updated list after insertion is 101 999 102 103 104.

**Sample Input 1**

3

201 202 203

1

555

**Sample Output 1**

555 201 202 203

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def add\_middle(self,data,pos):

q = Node(data)

if pos < 0:

return

if pos == 1: # adding at pos n-1, which means 0th position

q.next = self.first

self.first = q

return

count = 1

p = self.first

while(p and count < pos-1):

p = p.next

count +=1

q.next = p.next

p.next = q

def display(self):

if self.first is None:

return

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

pos = int(input())

ele = int(input())

for data in datas:

ll.add\_nodes(data)

ll.add\_middle(ele,pos)

ll.display()

=============================================================================

4) You are developing a task management application where tasks are represented by unique IDs. Users can delete tasks from the beginning of the task list to quickly remove the most urgent tasks. You need to write a program to handle the deletion of task IDs from the beginning of a linked list. After all deletions are completed, the final list of task IDs should be printed to confirm that the tasks have been removed.

Write a program to delete task IDs from the beginning of the linked list. After all deletions, print the entire linked list to show the updated list of task IDs. If the list is empty, Print the message as “List is empty”.

**Input Format**

1. The first line contains an integer N, the number of task IDs to be initially added to the linked list.
2. The second line contains N space-separated integers representing the task IDs.
3. The third line contains an integer M, the number of tasks to be deleted from the beginning.

**Constraints**

NA

**Output Format**

After completing all deletions, print the current linked list with elements separated by a space.

**Sample Input 0**

5

301 302 303 304 305

3

**Sample Output 0**

304 305

**Explanation 0**

The linked list is initialized with 5 task IDs: 301 302 303 304 305. After the first deletion, the task ID 301 is removed. After the second deletion, the task ID 302 is removed. After the third deletion, the task ID 303 is removed. Finally, the linked list becomes 304 305.

**Sample Input 1**

1

999

1

**Sample Output 1**

List is empty

**Explanation 1**

The linked list is initialized with a single task ID: 999.Since there is only one task in the list (999), deleting this single task will make the list empty.After performing the deletion, the list becomes empty.So that as per program's requirement print "List is empty".

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def delete\_node\_n(self,n):

count = 0

if n < 0 :

return

p = self.first

while(p and count < n):

p = p.next

count +=1

self.first = p

def display(self):

if self.first is None:

print("List is empty")

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

c = int(input())

for data in datas:

ll.add\_nodes(data)

ll.delete\_node\_n(c)

ll.display()

5) You are developing a project management application for a team of software developers. In this application, tasks are represented by unique IDs, and team members can add tasks to their project. As tasks are completed, they need to be removed from the task list to maintain an organized view of ongoing tasks. The team wants to delete the completed tasks from the end of the list, ensuring they focus only on active work.

Write a program to manage the deletion of completed tasks from the end of the task list. After all deletions, print the updated list of remaining task IDs to show which tasks are still pending. If the list is empty, Print the message as “List is empty”.

**Input Format**

1. The first line contains an integer N, the number of task IDs to be initially added to the task list.
2. The second line contains N space-separated integers representing the task IDs.
3. The third line contains an integer M, the number of completed tasks to be deleted from the end of the list.

**Constraints**

NA

**Output Format**

After completing all deletions, print the current task list with elements separated by a space.

**Sample Input 0**

6

201 202 203 204 205 206

4

**Sample Output 0**

201 202

**Explanation 0**

The initial task list contains 6 task IDs: 201 202 203 204 205 206. After the first deletion, the task ID 206 (the last task) is removed. After the second deletion, the task ID 205 is removed. After the third deletion, the task ID 204 is removed. After the fourth deletion, the task ID 203 is removed. Finally, the remaining task list contains 201 202, indicating these are the active tasks left to complete.

**Sample Input 1**

5

101 102 103 104 105

5

**Sample Output 1**

List is empty

**Explanation 1**

The initial task list contains 5 task IDs: 101, 102, 103, 104, 105.After the first deletion, the task ID 105 (the last task) is removed from the list.After the second deletion, the task ID 104 is removed.After the third deletion, the task ID 103 is removed.After the fourth deletion, the task ID 102 is removed.After the fifth deletion, the task ID 101 is removed, which is the only task left in the list.Finally, after all deletions, the list becomes empty, indicating that all tasks have been completed and there are no active tasks left to work on. Thus, the output will be "List is empty."

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self, data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def delete\_node\_n(self, n):

if n <= 0:

return

# Get length of list

length = 0

temp = self.first

while temp:

length += 1

temp = temp.next

# If n >= length, delete entire list

if n >= length:

self.first = None

self.last = None

return

# Traverse to (length - n - 1)th node

count = 1

p = self.first

while p and count < (length - n):

p = p.next

count += 1

if p and p.next:

p.next = None

self.last = p

def display(self):

if self.first is None:

print("List is empty")

return

p = self.first

while p:

print(p.data, end=" ")

p = p.next

# ----------- Main Code ------------

n = int(input())

datas = list(map(int, input().split()))

c = int(input())

ll = SLL()

for data in datas:

ll.add\_nodes(data)

ll.delete\_node\_n(c)

ll.display()

6) You are building a task tracker for project management, where each task is represented by a unique ID and tasks are stored in the order they need to be completed. Sometimes, due to changes in project priorities, certain tasks need to be removed from the list based on their position. Your task is to write a program that deletes a task at a specified position from the list and prints the updated list of remaining tasks.

Write a program that deletes a task (represented by a unique ID) from a given position in the linked list. After the deletion, print the updated list of task IDs. If the list is empty, Print the message as “List is empty”. If the entered position value is out the range, print the message as “Position out of range”.

**Input Format**

1. The first line contains an integer N, the number of task IDs initially added to the linked list.
2. The second line contains N space-separated integers representing the task IDs.
3. The third line contains an integer P, the position of the task to be deleted (1-based index).

**Constraints**

The value of P will always be between 1 and N (both inclusive).

**Output Format**

Print the updated task list with the elements separated by a space.

**Sample Input 0**

5

101 102 103 104 105

3

**Sample Output 0**

101 102 104 105

**Explanation 0**

The task list is initialized with 5 task IDs: 101 102 103 104 105. The task at position 3 (103) is removed from the list. The updated list after deletion is 101 102 104 105.

**Sample Input 1**

0

1

**Sample Output 1**

List is empty

**Explanation 1**

The task list is initially empty. Attempting to delete at position 1 returns "List is empty."

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self, data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def display(self):

if self.first is None:

print("List is empty")

return

p = self.first

while p:

print(p.data, end=" ")

p = p.next

def remove\_pos(self, pos, n):

if pos <= 0 or pos > n:

return

if pos == 1:

self.first = self.first.next

return

p = self.first

count = 1

while p and count < pos - 1:

p = p.next

count += 1

if p and p.next:

q = p.next

p.next = q.next

q.next = None

# ---------- Main ----------

n = int(input())

# Handle empty list case early

if n == 0:

print("List is empty")

else:

datas = list(map(int, input().split()))

ll = SLL()

for data in datas:

ll.add\_nodes(data)

pos = int(input())

ll.remove\_pos(pos, n)

ll.display()

7) You are designing a digital library system where books are represented by unique book IDs. The library catalog is stored as a doubly linked list, allowing users to navigate both forwards and backwards through the list of books. When a new book is added to the library, it needs to be appended to the end of the catalog. Your task is to write a program that inserts a new book at the end of the catalog and prints the updated list of book IDs.

Write a program to append a new book (represented by a unique book ID) to the end of the doubly linked list representing the library catalog. After the insertion, print the updated list of book IDs.

**Input Format**

1. The first line contains an integer N, the number of books initially in the catalog.
2. The second line contains N space-separated integers representing the book IDs.

**Constraints**

1. The input values are all positive integers.
2. There will be at least 1 book in the catalog initially.

**Output Format**

Print the updated list of book IDs in sequence from the start to the end of the catalog.

**Sample Input 0**

4

201 202 203 204

**Sample Output 0**

201 202 203 204

**Explanation 0**

Initially, the catalog is empty. First, the new book ID 201 is appended to the end of the catalog. The list becomes 201. Then the second book ID 202 is appended to the end of the catalog, making the list 201 202. Next, the third book ID 203 is appended to the end of the catalog, updating the list to 201 202 203. Finally, the fourth book ID 204 is appended to the end of the catalog, resulting in the final list of book IDs: 201 202 203 204.

**Sample Input 1**

3

101 102 103

**Sample Output 1**

101 102 103

class Node:

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

self.prev = None

self.data = data

self.next = None

class DLL:

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

self.first = None

self.last = None

def add\_node\_end(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

p.next = self.last.next

p.prev = self.last

self.last.next = p

self.last = p

def display(self):

p = self.first

while(p):

print(p.data, end = " ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

dll = DLL()

for data in datas:

dll.add\_node\_end(data)

dll.display()

==============================================================================

8) You are building a digital library system where each book is represented by a unique book ID. The library catalog is stored as a doubly linked list, allowing users to navigate through the list of books. Occasionally, books are removed from the library, and when a book is removed from the end of the catalog, it should no longer appear in the list. Your task is to write a program that deletes the last book from the catalog and prints the updated list of book IDs.

Write a program to delete the last book from the doubly linked list representing the library catalog. After the deletion, print the updated list of book IDs. If the list is empty, Print the message as “List is empty”.

**Input Format**

1. The first line contains an integer N, the number of books initially in the catalog.
2. The second line contains N space-separated integers representing the book IDs.

**Constraints**

1. The input values are all positive integers.
2. There will be at least 1 book in the catalog initially.

**Output Format**

Print the updated list of book IDs in sequence from the start to the end of the catalog after the last book has been removed.

**Sample Input 0**

5

101 102 103 104 105

**Sample Output 0**

101 102 103 104

**Explanation 0**

The catalog is initialized with 5 book IDs: 101 102 103 104 105. The last book ID 105 is removed from the end of the catalog. The updated list after deletion is 101 102 103 104.

**Sample Input 1**

1

701

**Sample Output 1**

List is empty

**Explanation 1**

The catalog initially contains one book ID: 701. After removing the last book (701), the list becomes empty.So that print "List is empty"

class Node:

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

self.prev = None

self.data = data

self.next = None

class DLL:

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

self.first = None

self.last = None

def add\_node\_end(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

p.next = self.last.next

p.prev = self.last

self.last.next = p

self.last = p

def remove\_last\_node(self):

if self.last is None:

return

if self.first == self.last:

# Only one node in the list

self.first = None

self.last = None

else:

p = self.last.prev

p.next = None

self.last = p

def display(self):

if self.first is None:

print("List is empty")

p = self.first

while(p):

print(p.data, end = " ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

dll = DLL()

for data in datas:

dll.add\_node\_end(data)

dll.remove\_last\_node()

dll.display()

=============================================================================

9) You are developing a task management system where tasks are stored in a circular to-do list. Each task is represented by a unique task ID. The circular nature of the list allows users to loop through their tasks continuously. Your task is to write a program that inserts a new task at the end of the circular linked list and prints the updated list of task IDs.

Write a program to append a new task (represented by a unique task ID) to the end of the singly circular linked list representing the to-do list. After the insertion, print the updated list of task IDs by traversing the list once starting from the head.

**Input Format**

1. The first line contains an integer N, the number of initial tasks in the to-do list.
2. The second line contains N space-separated integers representing the task IDs.

**Constraints**

1. The input values are all positive integers.
2. There will be at least one task initially in the to-do list.

**Output Format**

Print the updated list of task IDs in sequence from the head of the list, traversing the circular list once.

**Sample Input 0**

4

101 102 103 104

**Sample Output 0**

101 102 103 104

**Explanation 0**

The circular linked list is initially empty, and the first task with ID 101 is inserted, making it the head, pointing to itself. The second task ID 102 is inserted at the end, and the last node 101 now points to 102, which points back to the head. The third task ID 103 is added, with the last node 102 pointing to 103, and 103 pointing back to 101. The fourth task ID 104 is inserted at the end, with 103 pointing to 104, and 104 pointing back to 101. Finally, task ID 105 is appended, and the last node 104 points to 105, which in turn points back to the head 101.

**Sample Input 1**

3

201 202 203

**Sample Output 1**

201 202 203

class Node:

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

self.data = data

self.next = None

class CLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

self.last.next = self.first

else:

p.next = self.last.next

self.last.next = p

self.last = p

def display(self):

if self.first is None:

print("List is empty")

p = self.first

while True:

print(p.data,end=" ")

p = p.next

if p == self.first:

break

n = int(input())

datas = list(map(int,input().split()))

ll = CLL()

for data in datas:

ll.add\_nodes(data)

ll.display()

==============================================================================

10) Imagine you are managing a long-term project with a series of tasks arranged in a linear sequence. Each task is dependent on the completion of the previous one, and the tasks are stored in a singly linked list. At certain points in the project, you want to identify the critical task, which lies in the middle of the task flow, so you can allocate additional resources to it and ensure timely completion.

In case there are two middle tasks, you want to focus on the second middle task as it represents the next crucial step in the project.

You are given the head of a singly linked list, where each node represents a task in a project workflow. Your task is to find and return the middle task. If the list has two middle tasks, return the second middle task and all the tasks after it.

**Input Format**

1. The input consists of a linked list represented by its head node.
2. Each node contains an integer value representing the task ID or any relevant information.

**Constraints**

NA

**Output Format**

* The output will be a list of integers starting from the middle node to the end of the linked list.
* If there are two middle nodes, the output should start from the second middle node.

**Sample Input 0**

5

1 2 3 4 5

**Sample Output 0**

3 4 5

**Explanation 0**

* The list contains tasks with IDs: 1 -> 2 -> 3 -> 4 -> 5.
* The total number of tasks is 5, and the middle task is task 3.
* Since the list has an odd number of tasks, the function returns task 3 and all tasks following it (3 -> 4 -> 5).
* Thus, task 3 is the critical point in the workflow, and the system returns the tasks starting from the middle node.

**Sample Input 1**

6

10 20 30 40 50 60

**Sample Output 1**

40 50 60

import math

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def delete\_node\_n(self,n):

count = 0

if n < 0 :

return

p = self.first

while(p and count < n):

p = p.next

count +=1

self.first = p

def display(self):

if self.first is None:

print("List is empty")

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

c = n//2

c = math.ceil(c)

for data in datas:

ll.add\_nodes(data)

ll.delete\_node\_n(c)

ll.display()

==========================================================================

11) Imagine you have a playlist of songs that you want to listen to. The songs are arranged in the order they were added to the playlist. Sometimes, you want to reverse the order of the playlist to listen to your favourite songs in the opposite order. This task can be represented using a singly linked list, where each node contains a song.

Your task is to reverse the playlist so that you can listen to the songs in the opposite order.

**Input Format**

1. The first line contains an integer n, representing the number of songs in the playlist.
2. The next n lines each contain a string, representing the name of a song.

**Constraints**

NA

**Output Format**

The output will be a single line containing the names of the songs in reverse order, separated by spaces.

**Sample Input 0**

5

Song1

Song2

Song3

Song4

Song5

**Sample Output 0**

Song5 Song4 Song3 Song2 Song1

**Explanation 0**

**Input Playlist**

You have a playlist with songs ordered from Song1 to Song5.

**Reversing Process**

You start at the head of the playlist (Song1) and begin to change the direction of the links between the songs.

After reversing, Song1 will point to null (as it becomes the last song), and the order will change so that Song5 becomes the first song in the playlist.

**Final Output**

You get the new playlist in the order you desire, from Song5 down to Song1.

**Sample Input 1**

3

SongA

SongB

SongC

**Sample Output 1**

SongC SongB SongA

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def display(self,p):

if p is None:

return

self.display(p.next)

print(p.data,end=" ")

n = int(input())

datas = [input() for \_ in range(n)]

ll = SLL()

for data in datas:

ll.add\_nodes(data)

ll.display(ll.first)

12) Imagine you're using a task manager application that keeps a list of jobs scheduled for execution. Each job has a priority, and they are arranged in a list based on when they were added. Due to certain changes in requirements, you need to swap the priority of a specific job (at a given position from the start) with another job that is located the same distance from the end of the list.

For example, if the task at position k = 2 from the beginning needs to be swapped with the task at position k = 2 from the end, this will ensure that important tasks can move up or down based on their urgency.

Swap the job at the k-th position from the start of the list with the k-th position from the end.

**Input Format**

1. The first line contains a list of integers (or job IDs) representing the task list in the task manager.
2. The second line contains an integer k, representing the position of the task to swap from both the start and the end.

**Constraints**

NA

**Output Format**

The output is a list of job IDs where the k-th job from the start and the k-th job from the end have been swapped.

**Sample Input 0**

5

1 2 3 4 5

2

**Sample Output 0**

1 4 3 2 5

**Explanation 0**

The job list [1, 2, 3, 4, 5] and k = 2. Swap the second task from the start (Job 2) with the second task from the end (Job 4). After swapping, the list becomes [1, 4, 3, 2, 5].

**Sample Input 1**

6

1 2 3 4 5 6

3

**Sample Output 1**

1 2 4 3 5 6

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self, data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def swap\_nodes(self, pos, n):

start = pos

end = n - pos + 1

if start < 0 or end > n:

return

p = q = None

ptr = self.first

i = 1

while ptr:

if i == start:

p = ptr

if i == end:

q = ptr

ptr = ptr.next

i += 1

if p and q:

p.data, q.data = q.data, p.data

def display(self):

if self.first is None:

return

p = self.first

while p:

print(p.data, end=" ")

p = p.next

# Input reading

n = int(input())

datas = list(map(int, input().split()))

pos = int(input())

ll = SLL()

for data in datas:

ll.add\_nodes(data)

ll.swap\_nodes(pos, n)

ll.display()

13) Imagine you are managing a task scheduler that tracks a list of tasks to be executed in the future. Each task has a position in the list, representing its priority in execution. Sometimes, a task near the end of the list might become irrelevant or outdated due to changing priorities or deadlines, and you need to remove it.

For example, if a task becomes unnecessary due to an updated project deadline, you may want to remove the task from the list. However, you only know the position of the task relative to the end of the list (e.g., the 2nd last task). The system must be capable of efficiently removing tasks from the scheduler based on their relative position from the end of the list.

Remove the task at the n-th position from the end of the task list and return the updated list of tasks.

**Input Format**

1. The first line contains an integer n, representing the number of nodes (tasks) in the list.
2. The next line contains n integers, each representing the task ID in the list.
3. The last line contains an integer n, representing the position of the node from the end that needs to be removed.

**Constraints**

NA

**Output Format**

The output is a list of task IDs with the n-th task from the end removed.

**Sample Input 0**

5

1 2 3 4 5

2

**Sample Output 0**

1 2 3 5

**Explanation 0**

The task list [1, 2, 3, 4, 5] and n = 2. Remove the 2nd task from the end, which is 4. After removing 4, the list becomes [1, 2, 3, 5].

**Sample Input 1**

6

10 20 30 40 50 60

3

**Sample Output 1**

10 20 30 50 60

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def remove\_pos(self, pos, n):

pos = n - pos +1

if pos <= 0 or pos > n:

return

if pos == 1:

self.first = self.first.next

return

p = self.first

count = 1

while p and count < pos - 1:

p = p.next

count += 1

if p and p.next:

q = p.next

p.next = q.next

q.next = None

def display(self):

if self.first is None:

return

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

pos = int(input())

for data in datas:

ll.add\_nodes(data)

ll.remove\_pos(pos,n)

ll.display()

14) In a task scheduling system, tasks are arranged in a linked list, with each node representing a task. Sometimes, tasks need to be swapped in pairs for optimizing execution priority, based on a new requirement where adjacent tasks should alternate in priority. The system needs to swap every two adjacent tasks without altering their actual data but just rearranging the nodes.

For instance, if Task1 and Task2 are adjacent and Task3 and Task4 follow, after swapping the tasks in pairs, the list should have Task2 first, followed by Task1, then Task4 followed by Task3. This rearrangement optimizes task priority while maintaining data integrity.

Swap every two adjacent tasks in a linked list.

**Input Format**

1. The first line contains an integer n, representing the number of nodes (tasks) in the list.
2. The next line contains n integers, each representing the task IDs in the list.

**Constraints**

NA

**Output Format**

The output is a list of task IDs with every two adjacent tasks swapped.

**Sample Input 0**

4

1 2 3 4

**Sample Output 0**

2 1 4 3

**Explanation 0**

The program accepts the number of elements in the linked list and then takes the values of the list elements from the user. The swapPairs function swaps every two adjacent nodes without altering their values. It uses a dummy node and pointer manipulation to perform the swap. The modified linked list is displayed after performing the swaps in pairs.

**Sample Input 1**

6

10 20 30 40 50 60

**Sample Output 1**

20 10 40 30 60 50

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self, data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def display(self):

if self.first is None:

print("List is empty")

return

p = self.first

while p:

print(p.data, end=" ")

p = p.next

print()

def swap\_adj\_nodes(self):

p = self.first

while p and p.next:

q = p.next

p.data, q.data = q.data, p.data

p = q.next

n = int(input())

datas = list(map(int, input().split()))

ll = SLL()

for data in datas:

ll.add\_nodes(data)

ll.swap\_adj\_nodes()

ll.display()

15) n a hospital, two departments—Emergency and Outpatient—maintain their own sorted lists of patients based on the time of arrival. When a patient is ready to be treated, the hospital needs to merge these two lists into one sorted list to efficiently manage patient care and ensure that all patients are treated in the order they arrived.

For example, if the Emergency department has patients arriving at [1, 3, 5] (representing time of arrival in hours) and the Outpatient department has patients arriving at [2, 4, 6], the hospital must merge these lists to ensure the treatment occurs in the correct order. Merge two sorted linked lists into one sorted linked list

**Input Format**

1. The first line contains an integer n, representing the number of nodes in the first list.
2. The next line contains n integers representing the sorted values in the first list.
3. The following line contains an integer m, representing the number of nodes in the second list.
4. The last line contains m integers representing the sorted values in the second list.

**Constraints**

NA

**Output Format**

The output is a single sorted linked list containing all the nodes from both lists.

**Sample Input 0**

3

1 3 5

3

2 4 6

**Sample Output 0**

1 2 3 4 5 6

**Explanation 0**

The first department (Emergency) has patients arriving at [1, 3, 5], and the second department (Outpatient) has patients arriving at [2, 4, 6].

Start with the first patient from each list: compare 1 (from the Emergency list) and 2 (from the Outpatient list). Since 1 is smaller, it is added to the merged list.

Next, compare 3 and 2. Since 2 is smaller, it is added next.

Continue comparing the remaining elements in both lists until all patients are added to the merged list:

Merge steps:

1 (from Emergency)

2 (from Outpatient)

3 (from Emergency)

4 (from Outpatient)

5 (from Emergency)

6 (from Outpatient)

The merged list is [1, 2, 3, 4, 5, 6], ensuring that patients are treated in the order of their arrival.

**Sample Input 1**

2

5 15

4

1 2 3 10

**Sample Output 1**

1 2 3 5 10 15

**Explanation 1**

The first department (Emergency) has patients arriving at [5, 15].

The second department (Outpatient) has patients arriving at [1, 2, 3, 10].

Start with the first patient from each list: compare 5 (from the Emergency list) and 1 (from the Outpatient list). Since 1 is smaller, it is added to the merged list.

Next, compare 5 and 2. Since 2 is smaller, it is added next.

Next, compare 5 and 3. Since 3 is smaller, it is added next.

Finally, the remaining patients 5 and 10 are added in order.

The merged list is [1, 2, 3, 5, 10, 15], ensuring that patients are treated in the order of their arrival.

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last =p

else:

self.last.next = p

self.last = p

def display(self,head):

if head is None:

print("List is empty")

return

p = head

while(p):

print(p.data,end=" ")

p = p.next

def merge(self,ll2):

p = self.first

q = ll2.first

if p is None:

return q

if q is None:

return p

if p.data < q.data:

head = last = p

p = p.next

last.next = None

else:

head = last = q

q = q.next

last.next = None

while p and q:

if p.data < q.data:

last.next = p

last = p

p = p.next

else:

last.next = q

last = q

q = q.next

if p:

last.next =p

else:

last.next = q

return head

n = int(input())

datas1 = list(map(int,input().split()))

m = int(input())

datas2 = list(map(int,input().split()))

ll1 = SLL()

ll2 = SLL()

for data in datas1:

ll1.add\_nodes(data)

for data in datas2:

ll2.add\_nodes(data)

head = ll1.merge(ll2)

ll1.display(head)

============================================================================

16) You are managing an order tracking system for an e-commerce platform, where each order has a unique tracking ID. The system maintains these tracking IDs in a linked list for real-time monitoring. After every few hours, you need to check if the sequence of tracking IDs forms a palindrome, which means the list reads the same forward and backward.

For example, if the tracking IDs are 121, 202, 121, then the list is a palindrome, as it reads the same in both directions. You are required to implement a program that checks whether the sequence of tracking IDs is a palindrome or not.

**Input Format**

1. The first line contains an integer N, the number of tracking IDs initially in the system.
2. The second line contains N space-separated integers representing the tracking IDs.

**Constraints**

NA

**Output Format**

* Print "Yes" if the tracking ID sequence is a palindrome.
* Print "No" if the sequence is not a palindrome.

**Sample Input 0**

5

121 202 303 202 121

**Sample Output 0**

Yes

**Explanation 0**

Initially, the linked list contains 121, 202, 303, 202, 121. Since the sequence reads the same forward and backward, the output is "Yes."

**Sample Input 1**

4

1 2 3 1

**Sample Output 1**

No

**Explanation 1**

In the list 1 2 3 1, the middle element is identified as 3, and the second half (3 1) is reversed to become 1 3. When comparing the first half (1 2) with the reversed second half (1 3), the values do not match at the second position (2 vs. 3). Since the halves differ, the list is not a palindrome, so the output is "No."

class Node:

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

self.prev = None

self.data = data

self.next = None

class DLL:

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

self.first = None

self.last = None

def add\_node\_end(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

p.next = self.last.next

p.prev = self.last

self.last.next = p

self.last = p

def check\_palindrome(self):

p = self.first

q = self.last

while(p and q):

if (p.data != q.data):

return False

if p == q :

break

p = p.next

q = q.prev

return True

n = int(input())

datas = list(map(int,input().split()))

dll = DLL()

for data in datas:

dll.add\_node\_end(data)

if dll.check\_palindrome():

print("Yes")

else:

print("No")

17) You are managing a real-time inventory system where each product has a unique product ID stored in a linked list. To optimize product tracking, you need to segregate the product IDs into even and odd categories. All even product IDs should appear before the odd product IDs, while maintaining their original relative order within each group.

Your task is to write a program that segregates the even and odd product IDs in the linked list. After segregation, print the entire linked list with even IDs first, followed by odd IDs.

**Input Format**

* + The first line contains an integer N, the number of product IDs initially in the linked list.
  + The second line contains N space-separated integers representing the product IDs.

**Constraints**

NA

**Output Format**

Print the modified linked list with even IDs followed by odd IDs, with elements separated by a space.

**Sample Input 0**

7

101 102 103 104 105 106 107

**Sample Output 0**

102 104 106 101 103 105 107

**Explanation 0**

* Initially, the product IDs are 101, 102, 103, 104, 105, 106, 107.
* After segregating, the even product IDs 102, 104, 106 appear first, followed by the odd product IDs 101, 103, 105, 107.

**Sample Input 1**

6

2 3 4 5 6 7

**Sample Output 1**

2 4 6 3 5 7

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def display(self,head):

if head is None:

return

p = head

while(p):

print(p.data,end=" ")

p = p.next

def segregate(self):

if self.first is None:

return None

p = self.first

odd = even = elast = olast = None

while p:

next\_node = p.next

p.next = None

if p.data % 2 == 0:

if even is None:

even = elast = p

else:

elast.next = p

elast = p

else:

if odd is None:

odd = olast = p

else:

olast.next = p

olast = p

p = next\_node

if even is None:

return odd

elif odd is None:

return even

else:

elast.next = odd

return even

n = int(input())

datas = list(map(int,input().split()))

ll = SLL()

for data in datas:

ll.add\_nodes(data)

head = ll.segregate()

ll.display(head)

18) You are working on an inventory management system where each product ID is stored in a linked list. The product IDs are represented in their normal order (most significant digit first), and you need to sum two product IDs stored in this form. Your task is to write a program that adds two such linked lists without reversing the lists, returning the result as a new linked list.

**Input Format**

1. The first line contains an integer N1, the number of digits in the first product ID.
2. The second line contains N1 space-separated integers representing the first product ID.
3. The third line contains an integer N2, the number of digits in the second product ID.
4. The fourth line contains N2 space-separated integers representing the second product ID.

**Constraints**

NA

**Output Format**

Print the resulting linked list that represents the sum of the two numbers, with elements separated by a space.

**Sample Input 0**

3

2 4 3

3

5 6 4

**Sample Output 0**

8 0 7

**Explanation 0**

* The first product ID is 243.
* The second product ID is 564.
* Their sum is 243 + 564 = 807, which is represented as 8 0 7.

You are working on an inventory management system where each product ID is stored in a linked list. The product IDs are represented in their normal order (most significant digit first), and you need to sum two product IDs stored in this form. Your task is to write a program that adds two such linked lists without reversing the lists, returning the result as a new linked list.

**Sample Input 1**

2

1 2

2

3 4

**Sample Output 1**

4 6

class Node:

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

self.prev = None

self.data = data

self.next = None

class DLL:

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

self.first = None

self.last = None

def add\_node\_end(self,data):

p = Node(data)

if self.first is None :

self.first = self.last = p

else:

p.next = self.last.next

p.prev = self.last

self.last.next = p

self.last = p

def add\_node\_front(self,data):

p = Node(data)

if self.first is None:

self.first = self.last = p

else:

p.next = self.first

p.prev = self.first.prev

self.first.prev = p

self.first =p

def display(self):

p = self.first

if p is None:

print("List is empty")

return

while(p):

print(p.data,end=" " )

p = p.next

def sum(self,dll):

p = self.last

q = dll.last

carry = 0

result = DLL()

while p or q or carry:

val1 = p.data if p else 0

val2 = q.data if q else 0

total = val1 + val2 +carry

carry = total//10

result.add\_node\_front(total%10)

if p: p = p.prev

if q: q = q.prev

return result

n = int(input())

datas1 = list(map(int,input().split()))

m= int(input())

datas2 = list(map(int,input().split()))

dll1 = DLL()

dll2 = DLL()

for data in datas1:

dll1.add\_node\_end(data)

for data in datas2:

dll2.add\_node\_end(data)

sum = dll1.sum(dll2)

sum.display()

========================================================================

19) You are working on a real-time inventory management system that maintains product IDs in two separate linked lists, each sorted in non-decreasing order. You are tasked with finding the product IDs that are common in both linked lists (intersection) and printing these common IDs.

Write a program to find the intersection of two sorted linked lists. After finding the intersection, print the product IDs that appear in both inventories.

**Input Format**

1. The first line contains an integer N1, the number of product IDs in the first linked list.
2. The second line contains N1 space-separated integers representing the product IDs in the first list.
3. The third line contains an integer N2, the number of product IDs in the second linked list.
4. The fourth line contains N2 space-separated integers representing the product IDs in the second list.

**Constraints**

NA

**Output Format**

Print the product IDs that appear in both lists, separated by a space. If there is no intersection, print "No Intersection".

**Sample Input 0**

5

101 102 103 104 105

4

102 104 106 108

**Sample Output 0**

102 104

**Explanation 0**

* The first linked list contains product IDs: 101, 102, 103, 104, 105.
* The second linked list contains product IDs: 102, 104, 106, 108.
* The intersection of these two lists is 102 and 104.

**Sample Input 1**

5

10 20 30 40 50

4

5 15 25 35

**Sample Output 1**

No Intersection

**Explanation 1**

* The first linked list contains product IDs: 10, 20, 30, 40, 50.
* The second linked list contains product IDs: 5, 15, 25, 35.
* There are no common IDs.So print "No Intersection".

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def add\_nodes(self,data):

p = Node(data)

if self.first is None:

self.first = p

self.last = p

else:

self.last.next = p

self.last = p

def intersection(self,ll2):

p = self.first

q = ll2.first

if p is None:

return q

if q is None:

return p

result = SLL()

while p and q:

if p.data == q.data:

result.add\_nodes(p.data)

p = p.next

q = q.next

elif p.data < q.data:

p = p.next

else:

q = q.next

return result

def display(self):

if self.first is None:

print("No Intersection")

return

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

n = int(input())

datas1 = list(map(int,input().split()))

m = int(input())

datas2 = list(map(int,input().split()))

ll1 = SLL()

ll2 = SLL()

for data in datas1:

ll1.add\_nodes(data)

for data in datas2:

ll2.add\_nodes(data)

result = ll1.intersection(ll2)

result.display()

20) You are working on a real-time inventory management system where product IDs are maintained in an unsorted linked list. However, due to data entry errors, some product IDs may be duplicated in the inventory. Your task is to write a program that removes duplicate product IDs from the linked list while preserving the original order of the first occurrence of each product ID.

**Input Format**

1. The first line contains an integer N, the number of product IDs to be initially added to the linked list.
2. The second line contains N space-separated integers representing the product IDs.

**Constraints**

NA

**Output Format**

After removing duplicates, print the remaining product IDs in the linked list, separated by a space.

**Sample Input 0**

7

101 102 103 101 104 105 102

**Sample Output 0**

101 102 103 104 105

**Explanation 0**

* The initial linked list contains product IDs: 101, 102, 103, 101, 104, 105, 102.
* The duplicates (101 and 102) are removed, and the resulting list retains the first occurrence of each product ID.

**Sample Input 1**

6

1 2 2 3 4 4

**Sample Output 1**

1 2 3 4

class Node:

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

self.data = data

self.next = None

class SLL:

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

self.first = None

self.last = None

def display(self):

if self.first is None:

return

p = self.first

while(p):

print(p.data,end=" ")

p = p.next

def add\_node\_end(self,data):

p = Node(data)

if self.first is None:

self.first = self.last = p

else:

self.last.next = p

self.last = p

def remove\_duplicate(self):

seen = set()

p = self.first

prev = None

while p :

if p.data not in seen:

seen.add(p.data)

prev = p

p = p.next

else:

prev.next = p.next

p = p.next

n = int(input())

arr = list(map(int,input().split()))

ll = SLL()

for data in arr:

ll.add\_node\_end(data)

ll.remove\_duplicate()

ll.display()