## **EXPERIMENT-4**

# AIM:

To perform various operations such as insertion, deletion, display on Singly Linked Lists, Doubly Linked Lists & Circular Linked Lists.

# DESCRIPTION:

A linked list is a dynamic data structure that consists of a sequence of nodes, each containing an element and a reference to the next node in the sequence. The first node is called the head, and the last node is called the tail. Linked lists are used to store collections of data that can be easily modified, such as inserting or deleting elements.

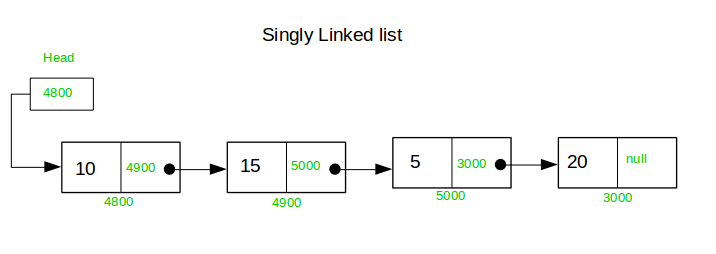
Linked List can be implemented as

• Singly Linked List

• Doubly Linked List

• Circular Linked List

## SINGLY LINKED LIST:

A singly linked list is a linear data structure, each node of which has one or more data item fields (DATA) but only a single link field (LINK). The entire linked list is kept track of by remembering the address of the START node(i.e. the first node in the list). The last node in the linked list has its link field empty and is often referred as NULL link .

# ADVANTAGES :

* Dynamic data structure: A linked list can grow and shrink at runtime by allocating and deallocating memory.
* Efficient insertion and deletion: Insertion and deletion in a linked list are very effective and take less time complexity as compared to an array data structure.
* Memory efficient: A linked list uses memory efficiently because it only allocates memory when it is needed.

# DISADVANTAGES:

* Traversal is more time-consuming as compared to an array.
* Direct access is not possible. Elements must be accessed sequentially starting from the first node.

# APPLICATIONS:

* Implementing stacks, queues, and other abstract data structures.
* Implementing dynamic memory allocation.
* Implementing file systems.

# PERFORMANCE:

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Space Complexity |
| Insert at beginning | O(1) | O(1) |
| Display list | O(n) | O(1) |
| Count number of nodes | O(n) | O(1) |
| Insert at end | O(n) | O(1) |
| Insert at position | O(n) | O(1) |
| Delete at beginning | O(1) | O(1) |
| Delete at end | O(n) | O(1) |
| Delete at position | O(n) | O(1) |
| Search for data | O(n) | O(1) |

# OPERATIONS ON SINGLE LINKED LIST:

## PSEUDOCODE:

1. **Insert at beginning:**

### insertAtBeginning(data):

### newNode = Node(data)

### newNode.next = head

### head = newNode

1. **Display List:**

### displayList():

### current = head

### while current != null:

### print(current.data)

### current = current.next

1. **Count number of nodes:**

### countNodes():

### count = 0

### current = head

### while current != null:

### count += 1

### current = current.next

### return count

1. **Insert at end:**

### insertAtEnd(data):

### newNode = Node(data)

### if head == null:

### head = newNode

### return

### current = head

### while current.next != null:

### current = current.next

### current.next = newNode

1. **Insert at position:**

### 

### insertAtPosition(data, position):

### if position == 0:

### insertAtBeginning(data)

### return

### newNode = Node(data)

### current = head

### for i in range(position - 1):

### if current == null:

### return

### current = current.next

### newNode.next = current.next

### current.next = newNode

1. **Delete at the beginning:**

### deleteAtBeginning():

### if head == null:

### return

### head = head.next

1. **Delete at the end:**

### deleteAtEnd():

### if head == null:

### return

### if head.next == null:

### head = null

### return

### previousToLastNode = head

### while previousToLastNode.next.next != null:

### previousToLastNode = previousToLastNode.next

### previousToLastNode.next = null

1. **Delete at position:**

### deleteAtPosition(position):

### if position == 0:

### deleteAtBeginning()

### return

### previousNode = null

### currentNode = head

### for i in range(position):

### if currentNode == null:

### return

### previousNode = currentNode

### currentNode = currentNode.next

### previousNode.next = currentNode.next;

1. **Search for data:**

### searchForData(data):

### current = head;

### while (current != null):

### if (current.data == data):

### return true;

### current = current.next;

### 

### return false;

# PROGRAM:

### #include <stdio.h>

### #include <stdlib.h>

### // Define the structure of a singly linked list node

### struct Node {

### int data;

### struct Node\* next;

### };

### // Function to create a new node

### struct Node\* createNode(int value) {

### struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

### newNode->data = value;

### newNode->next = NULL;

### return newNode;

### }

### // Function to insert a node at the beginning of the linked list

### void insertAtBeginning(struct Node\*\* head, int value) {

### struct Node\* newNode = createNode(value);

### newNode->next = \*head;

### \*head = newNode;

### }

### // Function to insert a node at the end of the linked list

### void insertAtEnd(struct Node\*\* head, int value) {

### struct Node\* newNode = createNode(value);

### if (\*head == NULL) {

### \*head = newNode;

### return;

### }

### struct Node\* current = \*head;

### while (current->next != NULL) {

### current = current->next;

### }

### current->next = newNode;

### }

### // Function to insert a node at a specified position in the linked list

### void insertAtPosition(struct Node\*\* head, int value, int position) {

### if (position < 1) {

### printf("Invalid position\n");

### return;

### }

### if (position == 1) {

### insertAtBeginning(head, value);

### return;

### }

### struct Node\* newNode = createNode(value);

### struct Node\* current = \*head;

### int currentPosition = 1;

### while (current != NULL && currentPosition < position - 1) {

### current = current->next;

### currentPosition++;

### }

### if (current == NULL) {

### printf("Invalid position\n");

### return;

### }

### newNode->next = current->next;

### current->next = newNode;

### }

### // Function to delete a node from the beginning of the linked list

### void deleteFromBeginning(struct Node\*\* head) {

### if (\*head == NULL) {

### printf("The list is empty\n");

### return;

### }

### struct Node\* temp = \*head;

### \*head = (\*head)->next;

### free(temp);

### }

### // Function to delete a node from the end of the linked list

### void deleteFromEnd(struct Node\*\* head) {

### if (\*head == NULL) {

### printf("The list is empty\n");

### return;

### }

### if ((\*head)->next == NULL) {

### free(\*head);

### \*head = NULL;

### return;

### }

### struct Node\* current = \*head;

### while (current->next->next != NULL) {

### current = current->next;

### }

### free(current->next);

### current->next = NULL;

### }

### // Function to delete a node from a specified position in the linked list

### void deleteFromPosition(struct Node\*\* head, int position) {

### if (\*head == NULL) {

### printf("The list is empty\n");

### return;

### }

### if (position < 1) {

### printf("Invalid position\n");

### return;

### }

### if (position == 1) {

### deleteFromBeginning(head);

### return;

### }

### struct Node\* current = \*head;

### struct Node\* previous = NULL;

### int currentPosition = 1;

### while (current != NULL && currentPosition < position) {

### previous = current;

### current = current->next;

### currentPosition++;

### }

### if (current == NULL) {

### printf("Invalid position\n");

### return;

### }

### previous->next = current->next;

### free(current);

### }

### // Function to search for an element in the linked list

### int searchElement(struct Node\* head, int value) {

### struct Node\* current = head;

### int position = 1;

### while (current != NULL) {

### if (current->data == value) {

### return position;

### }

### current = current->next;

### position++;

### }

### return -1; // Element not found

### }

### // Function to display the linked list

### void display(struct Node\* head) {

### struct Node\* current = head;

### while (current != NULL) {

### printf("%d -> ", current->data);

### current = current->next;

### }

### printf("NULL\n");

### }

### int main() {

### struct Node\* head = NULL;

### int choice, value, position, element, result;

### while (1) {

### printf("1. Insert at the beginning\n");

### printf("2. Insert at the end\n");

### printf("3. Insert at a position\n");

### printf("4. Delete from the beginning\n");

### printf("5. Delete from the end\n");

### printf("6. Delete from a position\n");

### printf("7. Search for an element\n");

### printf("8. Display the list\n");

### printf("9. Exit\n");

### printf("Enter your choice: ");

### scanf("%d", &choice);

### switch (choice) {

### case 1:

### printf("Enter the value to insert: ");

### scanf("%d", &value);

### insertAtBeginning(&head, value);

### break;

### case 2:

### printf("Enter the value to insert: ");

### scanf("%d", &value);

### insertAtEnd(&head, value);

### break;

### case 3:

### printf("Enter the value to insert: ");

### scanf("%d", &value);

### printf("Enter the position to insert: ");

### scanf("%d", &position);

### insertAtPosition(&head, value, position);

### break;

### case 4:

### deleteFromBeginning(&head);

### break;

### case 5:

### deleteFromEnd(&head);

### break;

### case 6:

### printf("Enter the position to delete: ");

### scanf("%d", &position);

### deleteFromPosition(&head, position);

### break;

### case 7:

### printf("Enter the element to search: ");

### scanf("%d", &element);

### result = searchElement(head, element);

### if (result != -1) {

### printf("Element %d found at position %d\n", element, result);

### } else {

### printf("Element %d not found\n", element);

### }

### break;

### case 8:

### printf("Linked list: ");

### display(head);

### break;

### case 9:

### exit(0);

### default:

### printf("Invalid choice\n");

### }

### }

### return 0;

### }

# OUTPUT:

##### Initial Linked List: 10 -> 20 -> 30 -> NULL

1. Insert at the beginning (Insert 5):

### Enter your choice: 1

### Enter the value to insert: 5

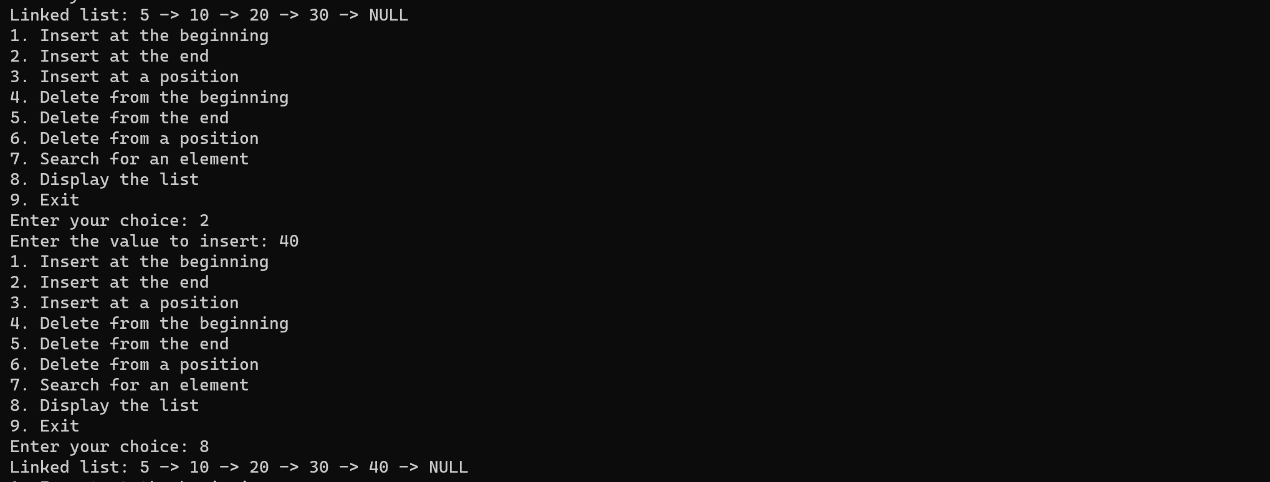
### Linked list after inserting 5 at the beginning: 5 -> 10 -> 20 -> 30 -> NULL

1. Insert at the end (Insert 40):

### Enter your choice: 2

### Enter the value to insert: 40

### Linked list after inserting 40 at the end: 5 -> 10 -> 20 -> 30 -> 40 -> NULL



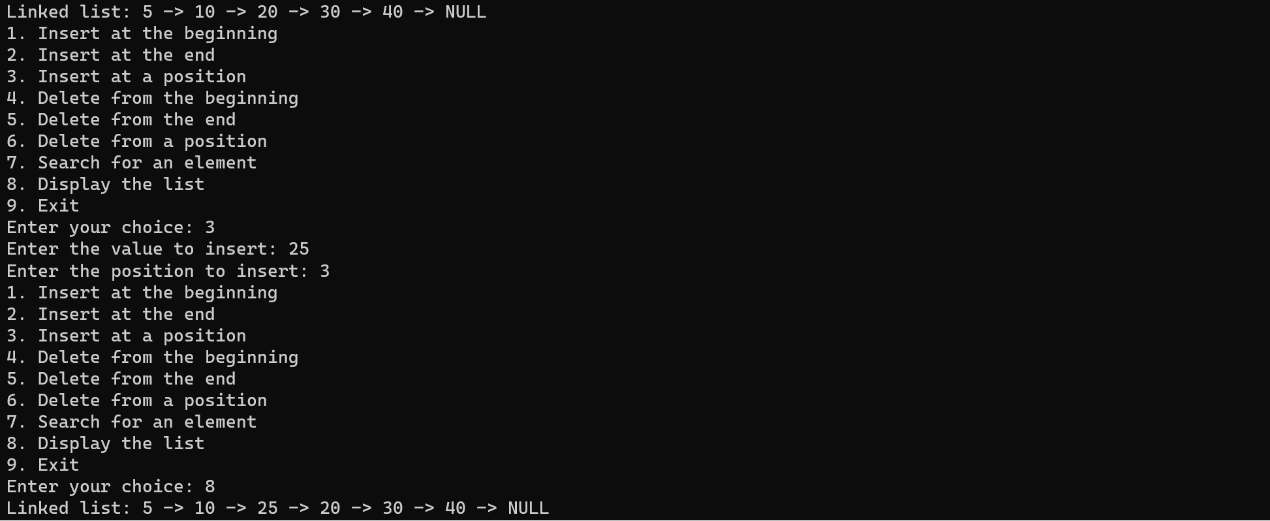
1. Insert at a position (Insert 25 at position 3):

### Enter your choice: 3

### Enter the value to insert: 25

### Enter the position to insert: 3

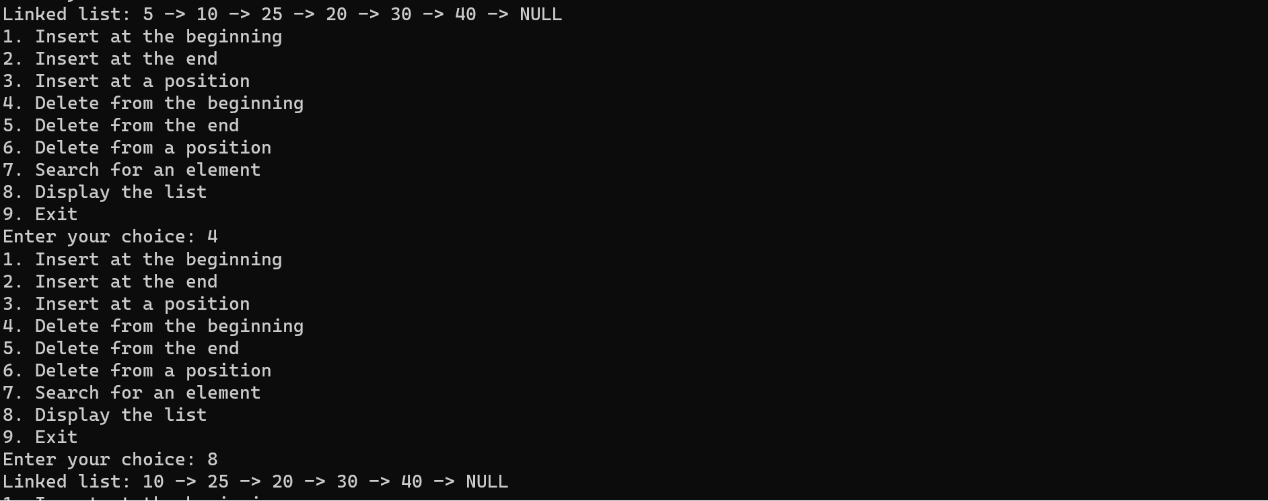
### Linked list after inserting 25 at position 3: 5 -> 10 -> 25 -> 20 -> 30 -> 40 -> NULL



1. Delete from the beginning:

### Enter your choice: 4

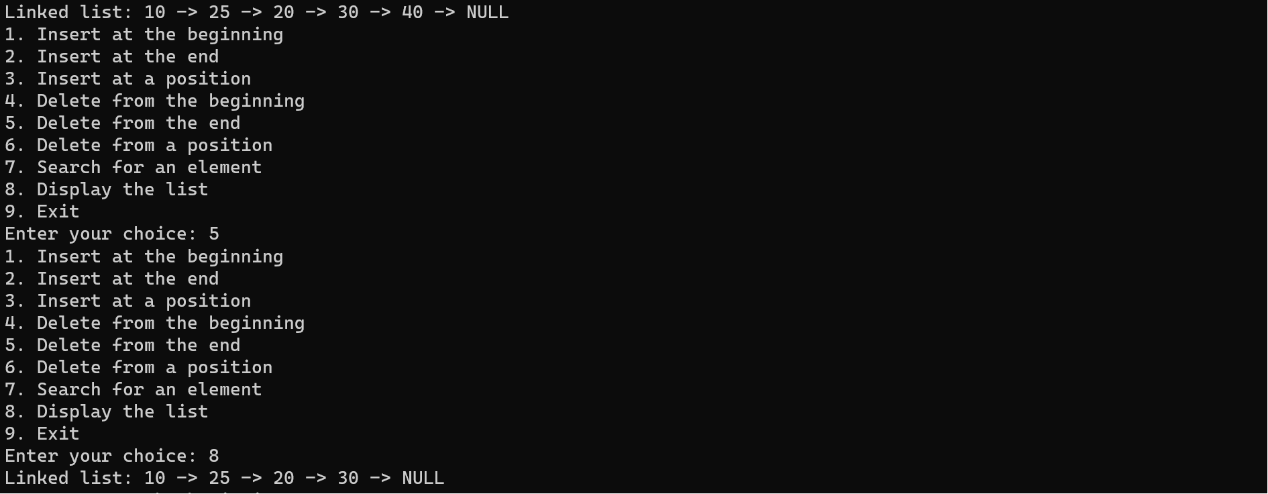
### Linked list after deleting from the beginning: 10 -> 25 -> 20 -> 30 -> 40 -> NULL



1. Delete from the end:

### Enter your choice: 5

### Linked list after deleting from the end: 10 -> 25 -> 20 -> 30 -> NULL



1. Delete from a position (Delete element at position 3):

Enter your choice: 6

Enter the position to delete: 3

Linked list after deleting element at position 3: 10 -> 25 -> 30 -> NULL

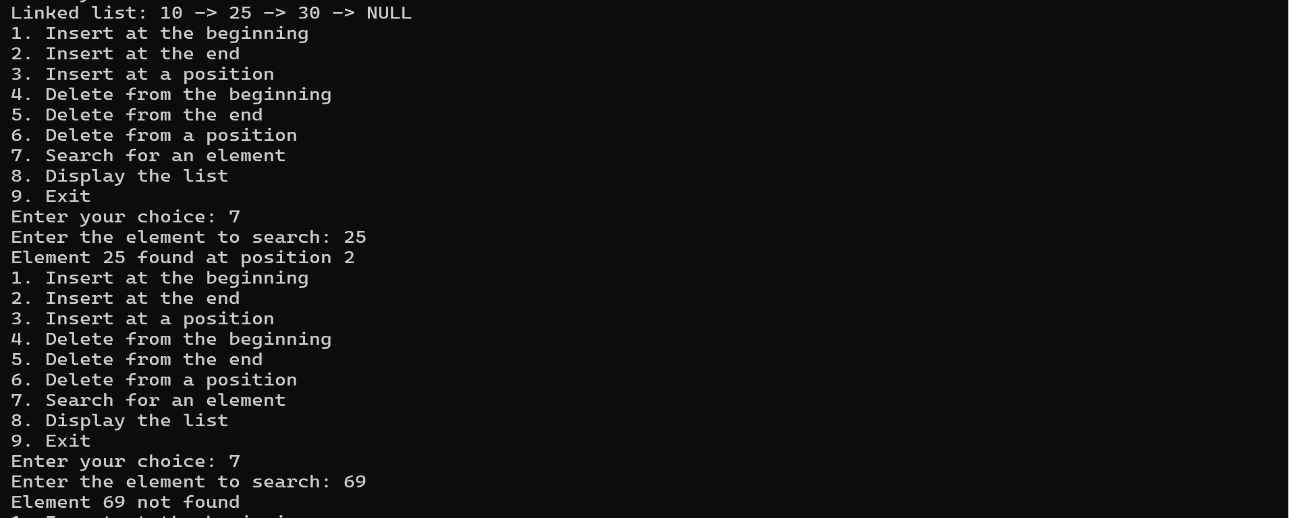


1. Search for an element (Search for 25):

Enter your choice: 7

Enter the element to search: 25

Element 25 found at position 2



## 2.DOUBLE LINKED LIST:

1. Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence. In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields
2. In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list.



|  |  |
| --- | --- |
| Characteristic | Description |
| Traversal | Bidirectional |
| Insertion and deletion | Efficient |
| Memory usage | High |
| Implementation complexity | High |

# ADVANTAGES:

* Reversing the doubly linked list is very easy.
* It can allocate or reallocate memory easily during its execution.
* As with a singly linked list, it is the easiest data structure to implement.
* The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
* Deletion of nodes is easy as compared to a Singly Linked List.

# DISADVANTAGES:

* It uses extra memory when compared to the array and singly linked list.
* Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.
* Traversing a doubly linked list can be slower than traversing a singly linked list.
* Implementing and maintaining doubly linked lists can be more complex than singly linked lists.

# APPLICATIONS:

* Navigation systems where front and back navigation is required.
* Browsers to implement backward and forward navigation of visited web pages that is a back and forward button.
* Representing a classic game deck of cards.
* Implementing undo and redo functionality.
* Constructing MRU/LRU (Most/least recently used) cache.

# PERFORMANCE:

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Space Complexity |
| Insert at beginning | O(1) | O(1) |
| Display list | O(n) | O(1) |
| Count number of nodes | O(n) | O(1) |
| Insert at end | O(1) | O(1) |
| Insert at position | O(n) | O(1) |
| Delete at beginning | O(1) | O(1) |
| Delete at end | O(1) | O(1) |
| Delete at position | O(n) | O(1) |
| Search for data | O(n) | O(1) |

# OPERATIONS ON DOUBLE LINKED LIST :

1. **Insert at beginning:**

### insertAtBeginning(data):

### newNode = Node(data)

### newNode.next = head

### newNode.prev = null

### if head != null:

### head.prev = newNode

### head = newNode

1. **Display List:**

### displayList():

### current = head

### while current != null:

### print(current.data)

### current = current.next

1. **Count number of nodes:**

### countNodes():

### count = 0

### current = head

### while current != null:

### count += 1

### current = current.next

### return count

1. **Insert at end:**

### insertAtEnd(data):

### newNode = Node(data)

### newNode.next = null

### if head == null:

### newNode.prev = null

### head = newNode

### return

### lastNode = head

### while lastNode.next != null:

### lastNode = lastNode.next

### lastNode.next = newNode

### newNode.prev = lastNode

1. **Insert at position:**

### insertAtPosition(data, position):

### if position == 0:

### insertAtBeginning(data)

### return

### newNode = Node(data)

### current = head

### for i in range(position - 1):

### if current == null:

### return

### current = current.next

### newNode.next = current.next

### if current.next != null:

### current.next.prev = newNode

### current.next = newNode

### newNode.prev = current

1. **Delete at the beginning:**

### deleteAtBeginning():

### if head == null:

### return

### if head.next == null:

### head = null

### return

### secondNode = head.next

### secondNode.prev = null

### head.next = null

### head = secondNode;

1. **Delete at the end:**

deleteAtEnd():

if head == null:

return

if head.next == null:

head = null;

return;

lastNode = head;

while (lastNode.next != null):

lastNode = lastNode.next;

secondLastNode = lastNode.prev;

secondLastNode.next = null;

lastNode.prev= null;

1. **Delete at position:**

deleteAtPosition(position):

if (head == None or position <= 0):

return

temp=head

if (position == 1):

head=temp.next

temp=None

return

for i in range(position -1 ):

temp=temp.next

if temp is None:

break

if temp is None:

return

if temp.next is None:

return

next=temp.next.next

temp.next=None

next.prev=temp

temp.next=next

1. **Search for data:**

### searchForData(data):

### node=head

### while node is not None:

### if node.data==data:

### return True

### node=node.next

### return False

1. **Print elements in reverse order:**

printReverse():

node=head

while node is not None:

last=node

node=node.next

while last is not None:

print(last.data)

last=last.prev

# PROGRAM:

### #include <stdio.h>

### #include <stdlib.h>

### // Define the structure of a doubly linked list node

### struct Node {

### int data;

### struct Node\* prev;

### struct Node\* next;

### };

### // Function to create a new node

### struct Node\* createNode(int value) {

### struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

### newNode->data = value;

### newNode->prev = NULL;

### newNode->next = NULL;

### return newNode;

### }

### // Function to insert a node at the beginning of the doubly linked list

### void insertAtBeginning(struct Node\*\* head, int value) {

### struct Node\* newNode = createNode(value);

### newNode->next = \*head;

### if (\*head != NULL) {

### (\*head)->prev = newNode;

### }

### \*head = newNode;

### }

### // Function to insert a node at the end of the doubly linked list

### void insertAtEnd(struct Node\*\* head, int value) {

### struct Node\* newNode = createNode(value);

### if (\*head == NULL) {

### \*head = newNode;

### return;

### }

### struct Node\* current = \*head;

### while (current->next != NULL) {

### current = current->next;

### }

### current->next = newNode;

### newNode->prev = current;

### }

### // Function to insert a node at a specified position in the doubly linked list

### void insertAtPosition(struct Node\*\* head, int value, int position) {

### if (position < 1) {

### printf("Invalid position\n");

### return;

### }

### if (position == 1) {

### insertAtBeginning(head, value);

### return;

### }

### struct Node\* newNode = createNode(value);

### struct Node\* current = \*head;

### int currentPosition = 1;

### while (current != NULL && currentPosition < position - 1) {

### current = current->next;

### currentPosition++;

### }

### if (current == NULL) {

### printf("Invalid position\n");

### return;

### }

### newNode->next = current->next;

### if (current->next != NULL) {

### current->next->prev = newNode;

### }

### current->next = newNode;

### newNode->prev = current;

### }

### // Function to delete a node from the beginning of the doubly linked list

### void deleteFromBeginning(struct Node\*\* head) {

### if (\*head == NULL) {

### printf("The list is empty\n");

### return;

### }

### struct Node\* temp = \*head;

### \*head = (\*head)->next;

### if (\*head != NULL) {

### (\*head)->prev = NULL;

### }

### free(temp);

### }

### // Function to delete a node from the end of the doubly linked list

### void deleteFromEnd(struct Node\*\* head) {

### if (\*head == NULL) {

### printf("The list is empty\n");

### return;

### }

### if ((\*head)->next == NULL) {

### free(\*head);

### \*head = NULL;

### return;

### }

### struct Node\* current = \*head;

### while (current->next->next != NULL) {

### current = current->next;

### }

### free(current->next);

### current->next = NULL;

### }

### // Function to delete a node from a specified position in the doubly linked list

### void deleteFromPosition(struct Node\*\* head, int position) {

### if (\*head == NULL) {

### printf("The list is empty\n");

### return;

### }

### if (position < 1) {

### printf("Invalid position\n");

### return;

### }

### if (position == 1) {

### deleteFromBeginning(head);

### return;

### }

### struct Node\* current = \*head;

### int currentPosition = 1;

### while (current != NULL && currentPosition < position) {

### current = current->next;

### currentPosition++;

### }

### if (current == NULL) {

### printf("Invalid position\n");

### return;

### }

### if (current->prev != NULL) {

### current->prev->next = current->next;

### } else {

### \*head = current->next;

### }

### if (current->next != NULL) {

### current->next->prev = current->prev;

### }

### free(current);

### }

### // Function to search for an element in the doubly linked list

### int searchElement(struct Node\* head, int value) {

### struct Node\* current = head;

### int position = 1;

### while (current != NULL) {

### if (current->data == value) {

### return position;

### }

### current = current->next;

### position++;

### }

### return -1; // Element not found

### }

### // Function to count the number of nodes in the doubly linked list

### int countNodes(struct Node\* head) {

### int count = 0;

### struct Node\* current = head;

### while (current != NULL) {

### count++;

### current = current->next;

### }

### return count;

### }

### // Function to display the doubly linked list from the beginning

### void displayForward(struct Node\* head) {

### struct Node\* current = head;

### while (current != NULL) {

### printf("%d -> ", current->data);

### current = current->next;

### }

### printf("NULL\n");

### }

### // Function to display the doubly linked list from the end

### void displayBackward(struct Node\* head) {

### struct Node\* current = head;

### while (current != NULL && current->next != NULL) {

### current = current->next;

### }

### printf("NULL <- ");

### while (current != NULL) {

### printf("%d -> ", current->data);

### current = current->prev;

### }

### printf("NULL\n");

### }

### int main() {

### struct Node\* head = NULL;

### int choice, value, position, element, result, nodeCount;

### while (1) {

### printf("1. Insert at the beginning\n");

### printf("2. Insert at the end\n");

### printf("3. Insert at a position\n");

### printf("4. Delete from the beginning\n");

### printf("5. Delete from the end\n");

### printf("6. Delete from a position\n");

### printf("7. Search for an element\n");

### printf("8. Display the list (Forward)\n");

### printf("9. Display the list (Backward)\n");

### printf("10. Count nodes\n");

### printf("11. Exit\n");

### printf("Enter your choice: ");

### scanf("%d", &choice);

### switch (choice) {

### case 1:

### printf("Enter the value to insert: ");

### scanf("%d", &value);

### insertAtBeginning(&head, value);

### break;

### case 2:

### printf("Enter the value to insert: ");

### scanf("%d", &value);

### insertAtEnd(&head, value);

### break;

### case 3:

### printf("Enter the value to insert: ");

### scanf("%d", &value);

### printf("Enter the position to insert: ");

### scanf("%d", &position);

### insertAtPosition(&head, value, position);

### break;

### case 4:

### deleteFromBeginning(&head);

### break;

### case 5:

### deleteFromEnd(&head);

### break;

### case 6:

### printf("Enter the position to delete: ");

### scanf("%d", &position);

### deleteFromPosition(&head, position);

### break;

### case 7:

### printf("Enter the element to search: ");

### scanf("%d", &element);

### result = searchElement(head, element);

### if (result != -1) {

### printf("Element %d found at position %d\n", element, result);

### } else {

### printf("Element %d not found\n", element);

### }

### break;

### case 8:

### printf("Linked list (Forward): ");

### displayForward(head);

### break;

### case 9:

### printf("Linked list (Backward): ");

### displayBackward(head);

### break;

### case 10:

### nodeCount = countNodes(head);

### printf("Number of nodes in the list: %d\n", nodeCount);

### break;

### case 11:

### exit(0);

### default:

### printf("Invalid choice\n");

### }

### }

### return 0;

### }

# OUTPUT:

##### Initial Doubly Linked List: NULL <- 10 <-> 20 <-> 30 -> NULL

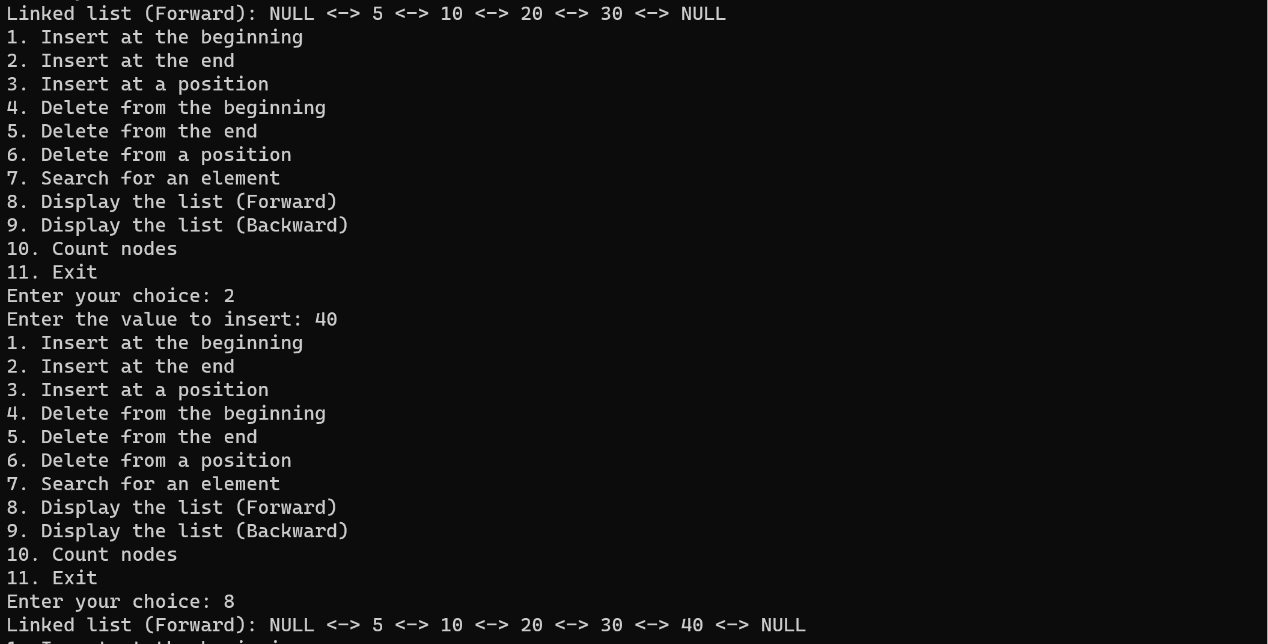
1. Insert at the beginning (Insert 5):

### Linked list after inserting 5 at the beginning: NULL <- 5 <-> 10 <-> 20 <-> 30 -> NULL

### 

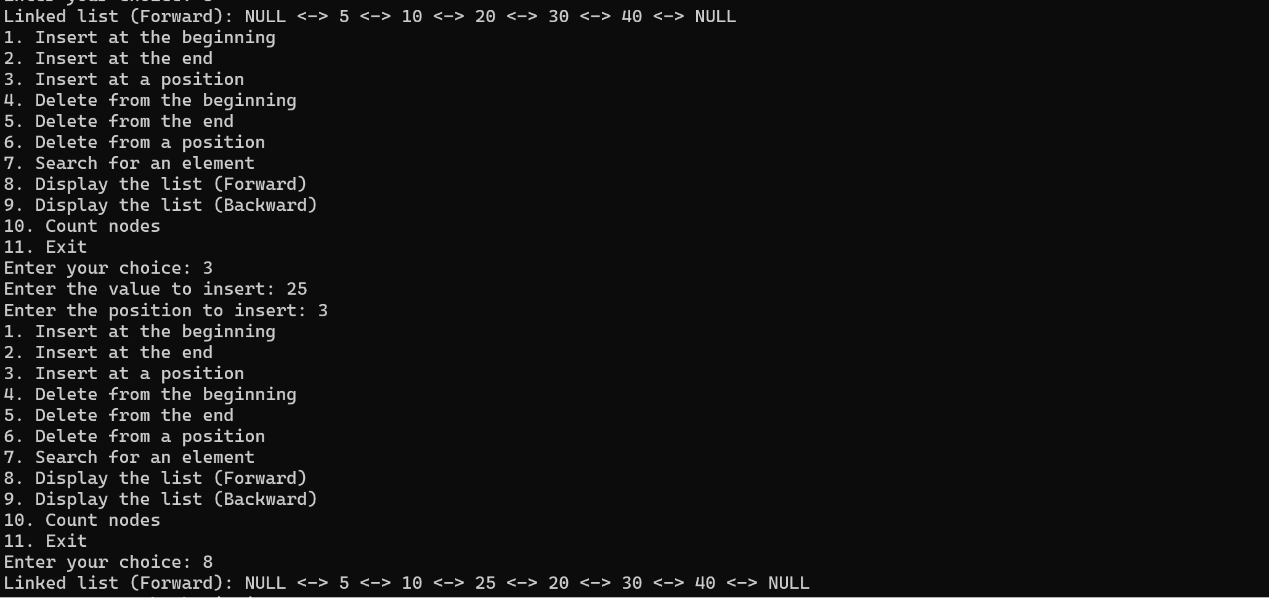
### Insert at the end (Insert 40):

Linked list after inserting 40 at the end: NULL <- 5 <-> 10 <-> 20 <-> 30 <-> 40 -> NULL



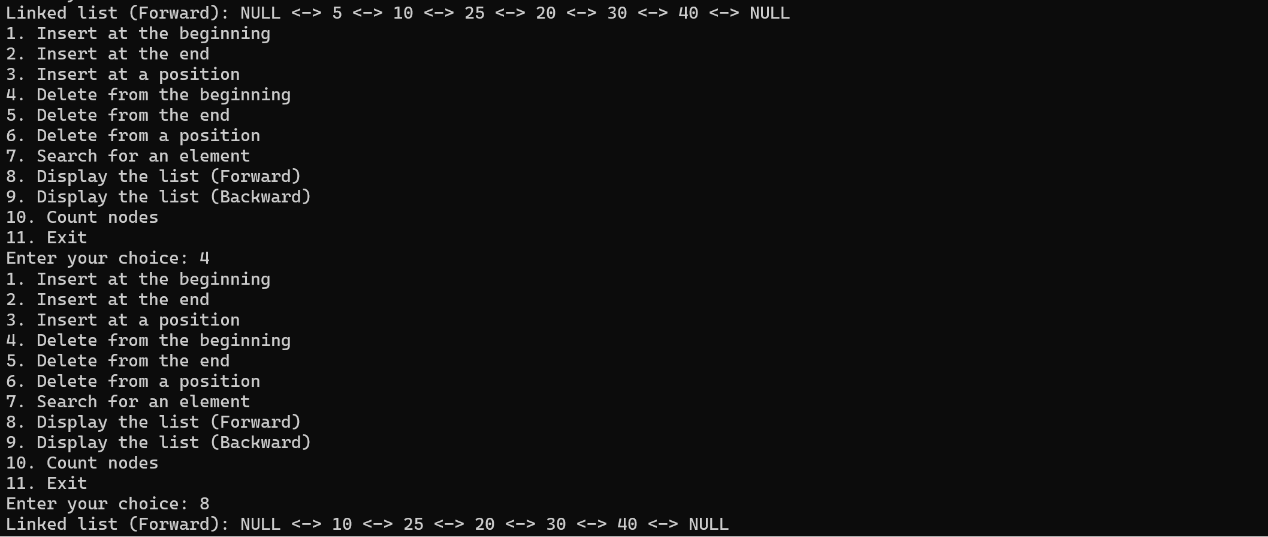
1. Insert at a position (Insert 25 at position 3):

### Linked list after inserting 25 at position 3: NULL <- 5 <-> 10 <-> 25 <-> 20 <-> 30 <-> 40 -> NULL



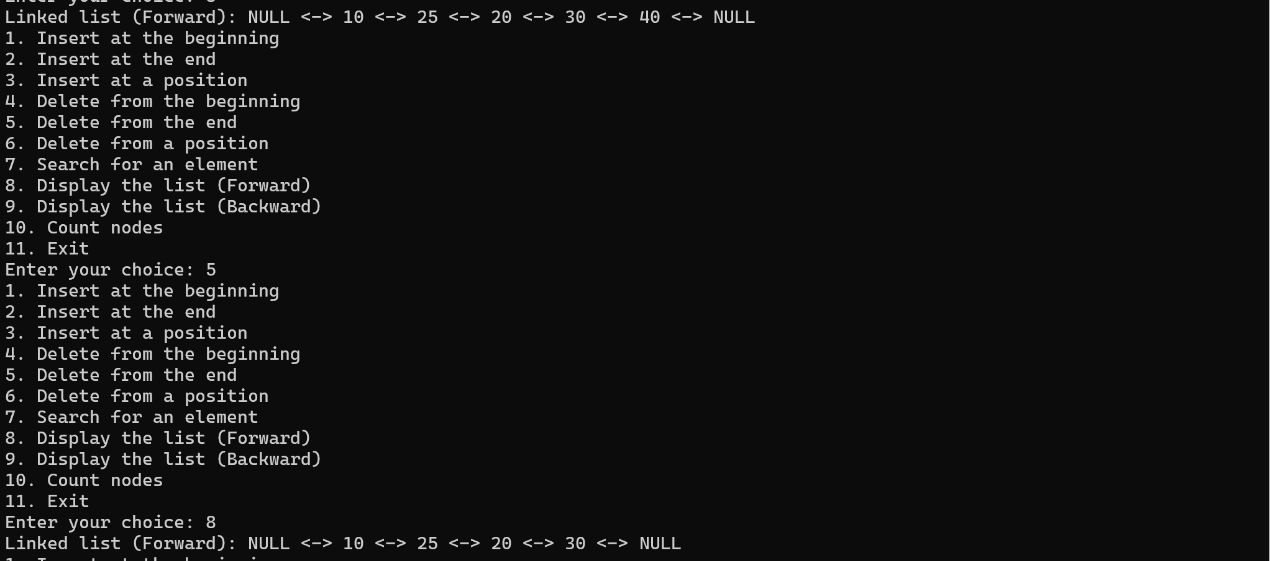
1. Delete from the beginning:

### Linked list after deleting from the beginning: NULL <- 10 <-> 25 <-> 20 <-> 30 <-> 40 -> NULL



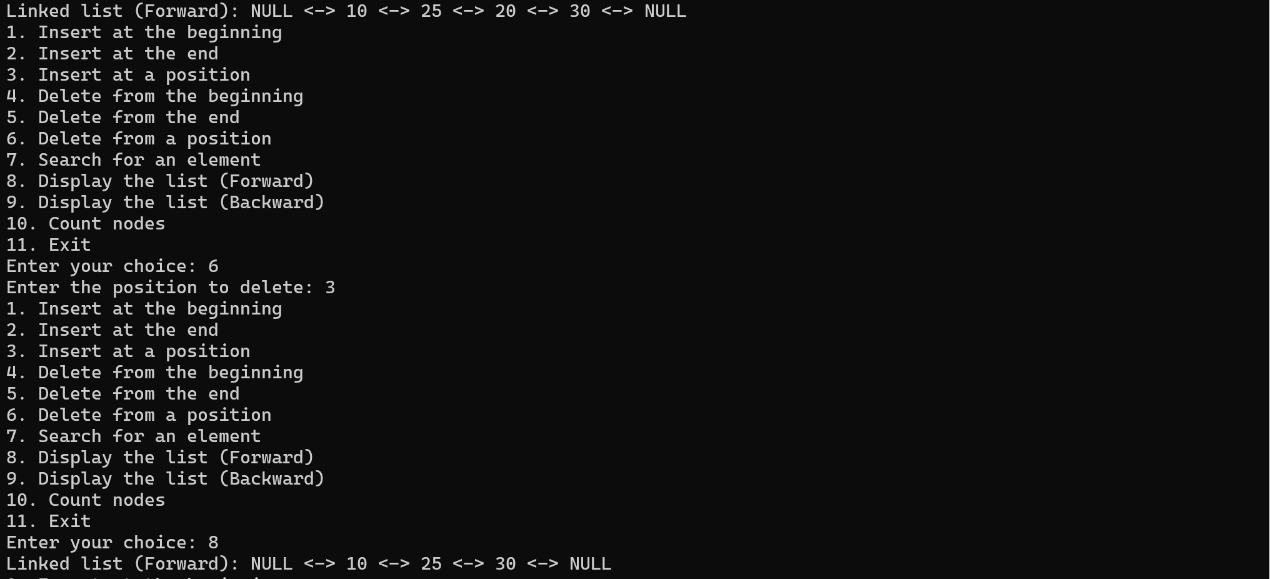
1. Delete from the end:

### Linked list after deleting from the end: NULL <- 10 <-> 25 <-> 20 <-> 30 -> NULL



1. Delete from a position (Delete element at position 3):

### Linked list after deleting element at position 3: NULL <- 10 <-> 25 <-> 30 -> NULL



1. Search for an element (Search for 25):

### Element 25 found at position 2



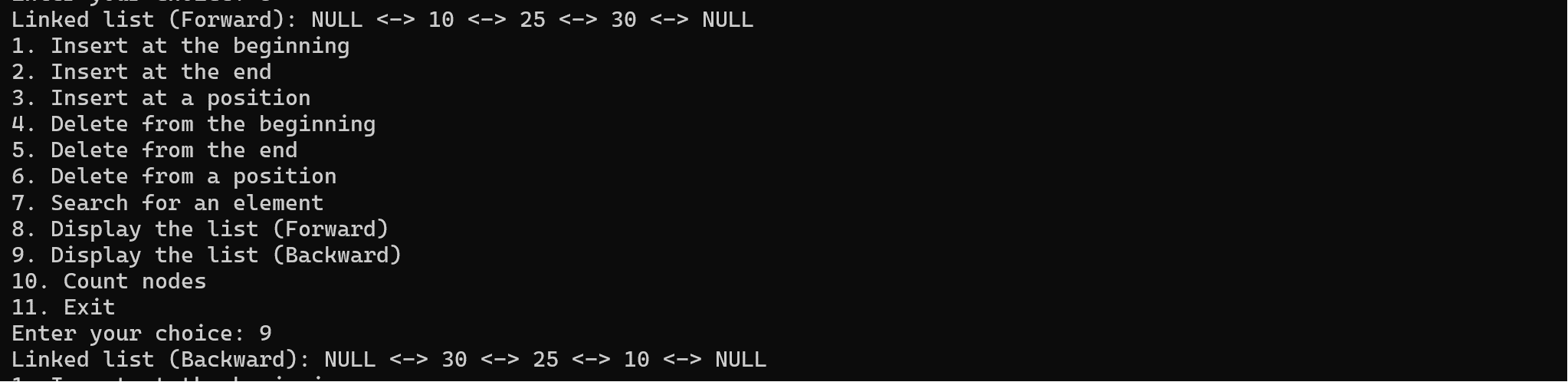
1. Display the list (Forward):

Linked list (Forward): NULL <- 10 <-> 25 <-> 30 -> NULL



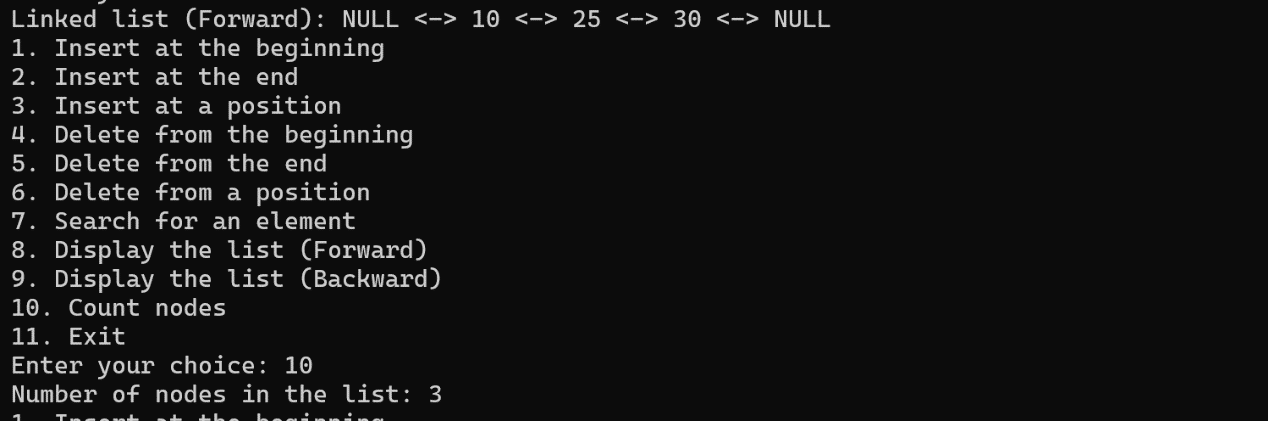
1. Display the list (Backward):

### Linked list (Backward): NULL <-> 30 <-> 25 <-> 10 <- NULL



1. Count nodes:

### Number of nodes in the list: 4

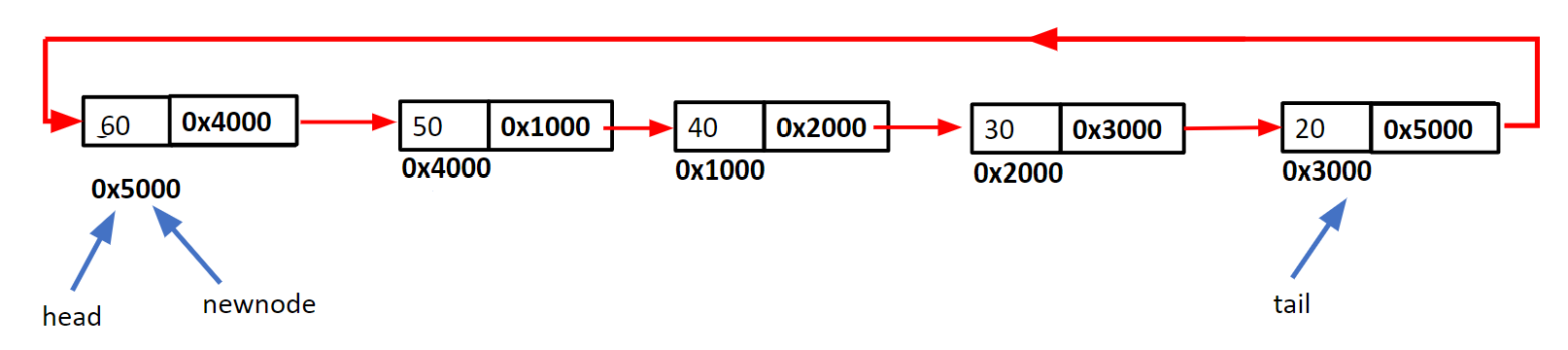


### 

## 3.CIRCULAR LINKED LIST :

# DESCRIPTION:

A circular singly linked list is a data structure that consists of a sequence of elements, where each element points to the next element in the sequence, and the last element points back to the first element, creating a loop or circle. It is a variation of the singly linked list, with the advantage of easy traversal from any node to any other node in a constant time.



# ADVANTAGES:

1. Efficient Traversal: Circular linked lists provide efficient traversal in both directions (forward and backward), as you can easily navigate from any node to its adjacent nodes.
2. Constant-Time Insertions and Deletions: Inserting or deleting a node at the beginning or end of the list takes constant time, O(1), once you have a reference to the node to be inserted or deleted.
3. Memory Efficiency: Circular linked lists can be more memory-efficient than arrays since they allocate memory dynamically as needed.

# DISADVANTAGES:

1. Complexity: Implementing circular linked lists can be more complex than implementing arrays or regular singly linked lists, especially when dealing with complex operations like inserting or deleting nodes at arbitrary positions.
2. No Random Access: Unlike arrays, circular linked lists do not support direct access to elements by index. To access a specific element, you must traverse the list from the beginning or any known reference point.

# APPLICATIONS:

1. Round Robin Scheduling: Circular linked lists are used in scheduling algorithms, such as round-robin, to efficiently manage tasks in a circular manner.
2. Music Playlist: Circular linked lists can be used to create circular playlists in music applications, where songs play one after another in a loop.
3. Memory Allocation: Memory allocation in some embedded systems is managed using circular linked lists to efficiently allocate and deallocate memory blocks.

# Performance (Time and Space Complexity):

|  |  |
| --- | --- |
| Operation | Time Complexity (Worst Case) |
| Insertion at the Beginning | O(1) |
| Insertion at the End | O(n) |
| Insertion at a Position | O(n) |
| Deletion at the Beginning | O(1) |
| Deletion at the End | O(n) |
| Deletion at a Position | O(n) |
| Search | O(n) |
| Traversal (Full List) | O(n) |

# OPERATIONS ON CIRCULAR LINKED LIST :

1. **Insert at Beginning**:

* Create a new node.
* If the list is not empty, set the ‘next’ of this new node to the current first node of the list.
* If the list is empty, set the ‘next’ of this new node to itself.
* Update the ‘next’ of last node to this new node.

1. **Display List**:

* Start from the first node.
* Print the data of the current node.
* Move to the next node.
* Continue this process until you reach the first node again.

1. **Count Number of Nodes**:

* Start from the first node.
* Move to the next node while keeping a count of nodes.
* Continue this process until you reach the first node again.

1. **Insert at End**:

* Create a new node.
* If the list is empty, follow steps from “Insert at Beginning”.
* If not, set ‘next’ of new node to the first node and ‘next’ of last node to this new node. Then update last node to this new node.

1. **Insert at Position**:

* Create a new node.
* If position is 0, follow steps from “Insert at Beginning”.
* If not, traverse to the desired position and set ‘next’ of new node to the ‘next’ of node at this position and ‘next’ of node at this position to this new node.

1. **Delete at Beginning**:

* If list is not empty, point ‘next’ of last node to the second node and update first node to this second node.

1. **Delete at End**:

* Traverse to second last node and set its ‘next’ to first node. Then update last node to this second last node.

1. **Delete at Position**:

* Traverse to the desired position.
* Set ‘next’ of its previous node to its next node.

1. **Search for Data**:

* Start from the first node and compare data of each node with given data.
  + - If found, return its position.
    - Else, return -1 if not found in any nodes.

**Program:**

/\*Operations on Circular Linked List\*/

#include<stdio.h>

#include<stdlib.h>

typedef struct node{

int data;

struct node\* next;

}node;

node\* head;

int count = 0;

void insert\_begin();

void insert\_end();

void delete\_begin();

void delete\_end();

void sort();

void remove\_duplicates();

void display();

int main()

{

char choice;

while(1){

printf("A. Insert at Beginning\n");

printf("B. Insert at End\n");

printf("C. Delete at Beginning\n");

printf("D. Delete at End\n");

printf("E. Sort the list\n");

printf("F. Delete duplicates\n");

printf("Y. Display\n");

printf("Z. Print number of nodes\n");

printf("Others to exit\n");

printf("Enter the choice: ");

scanf(" %c",&choice);

switch(choice)

{

case 'A': insert\_begin();

break;

case 'B': insert\_end();

break;

case 'C': delete\_begin();

break;

case 'D': delete\_end();

break;

case 'E': sort();

break;

case 'F': remove\_duplicates();

break;

case 'Y': display();

break;

case 'Z': printf("Number of nodes are %d\n",count);

break;

default: exit(0);

}

}

return 0;

}

void insert\_begin() {

int value;

printf("Enter value to be inserted: ");

scanf("%d", &value);

node\* temp;

temp = (node\*)malloc(sizeof(node));

if(temp == NULL) {

printf("Memory insufficient\n");

} else {

temp->data = value;

if(head == NULL) {

head = temp;

temp->next = head;

} else {

node\* last = head;

while(last->next != head) {

last = last->next;

}

temp->next = head;

last->next = temp;

head = temp;

}

count++;

}

}

void insert\_end() {

int value;

printf("Enter value to be inserted: ");

scanf("%d", &value);

node\* temp;

temp = (node\*)malloc(sizeof(node));

if(temp == NULL) {

printf("Memory insufficient\n");

} else {

temp->data = value;

if(head == NULL) {

head = temp;

temp->next = head;

} else {

node\* last = head;

while(last->next != head) {

last = last->next;

}

last->next = temp;

temp->next = head;

}

count++;

}

}

void delete\_begin() {

if(head == NULL) {

printf("List is empty\n");

} else {

node\* temp = head;

if(head->next == head) {

head = NULL;

} else {

node\* last = head;

while(last->next != head) {

last = last->next;

}

head = head->next;

last->next = head;

}

free(temp);

count--;

printf("First node is deleted successfully\n");

}

}

void delete\_end() {

if(head == NULL) {

printf("List is empty\n");

} else {

node\* temp = head;

if(head->next == head) {

head = NULL;

} else {

node\* prev = NULL;

while(temp->next != head) {

prev = temp;

temp = temp->next;

}

prev->next = head;

free(temp);

}

count--;

printf("Ending node is deleted\n");

}

}

void sort() {

if(head == NULL) {

printf("List is empty\n");

} else {

node \*current = head, \*index = NULL;

int temp;

if(head->next != head) {

do {

index = current->next;

while(index != head) {

if(current->data > index->data) {

temp = current->data;

current->data = index->data;

index->data = temp;

}

index = index->next;

}

current = current->next;

} while(current->next != head);

}

}

}

void remove\_duplicates() {

if(head == NULL) {

printf("List is empty\n");

} else {

node \*current = head, \*next\_next;

if(head->next != head) {

do {

if(current->data == current->next->data) {

next\_next = current->next->next;

free(current->next);

current->next = next\_next;

count--; // decrement count when a duplicate node is removed

} else {

current = current->next;

}

} while(current->next != head);

}

}

}

void display()

{

if(head == NULL)

printf("List is empty\n");

else

{

node\* p;

p = head;

while(p->next!=head)

{

printf("%d ",p->data);

p = p->next;

}

printf("%d\n",p->data);

}

}

**Test Case-1:** Check if the LIST is empty

Input:

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

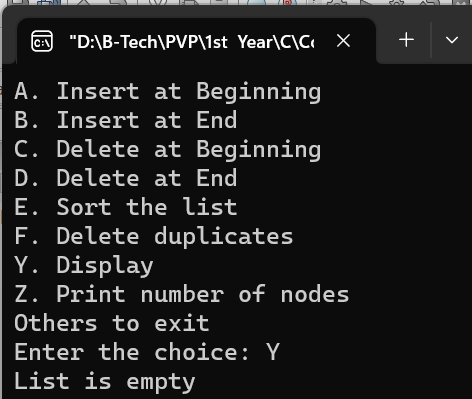
Z. Print number of nodes

Others to exit

Enter the choice: Y

Output:

List is empty



**Test Case-2:** Insert ITEM into the LIST as the first element

Input: List be 3 13 33 57 59, Inserting 2 at beginning

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

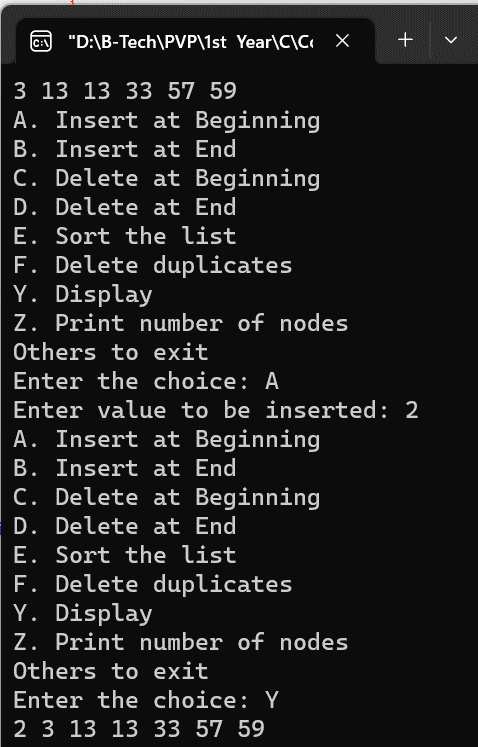
Z. Print number of nodes

Others to exit

Enter the choice: Y

Output:

3 13 13 33 57 59

 2 3 13 13 33 57 59

**Test Case-3:** Insert ITEM into the LIST as the last element

Input: List be 8 9 10 11, Inserting 12 at ending

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

Z. Print number of nodes

Others to exit

Enter the choice: Y

Enter the choice: B

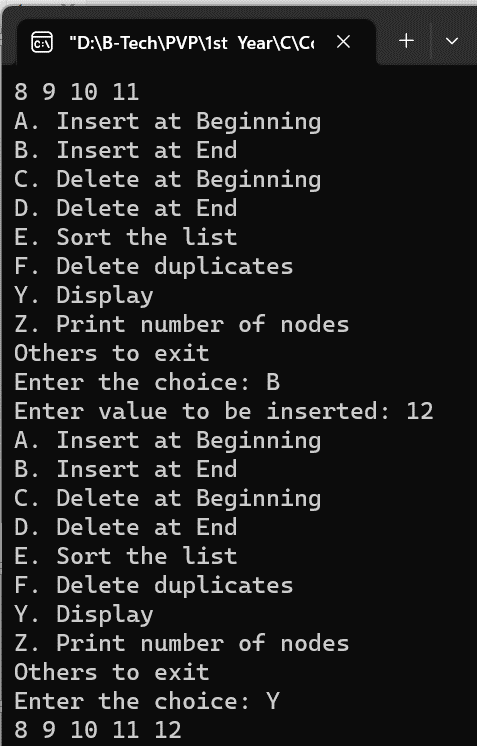
Enter value to be inserted: 12

Enter the choice: Y

Output:

8 9 10 11

8 9 10 11 12



**Test Case-4:** Delete the first node from the list

Input: List be 10 10 11 12 13 14 15

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

Z. Print number of nodes

Others to exit

Enter the choice: Y

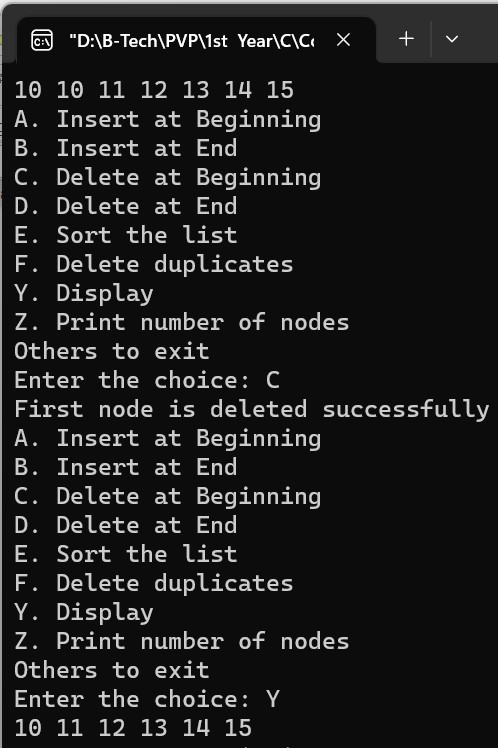
Enter the choice: C

Enter the choice: Y

Output:

10 10 11 12 13 14 15

10 11 12 13 14 15



**Test Case-5:** Delete the last node from the list

Input: List be 11 12 13 14 15 15

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

Z. Print number of nodes

Others to exit

Enter the choice: Y

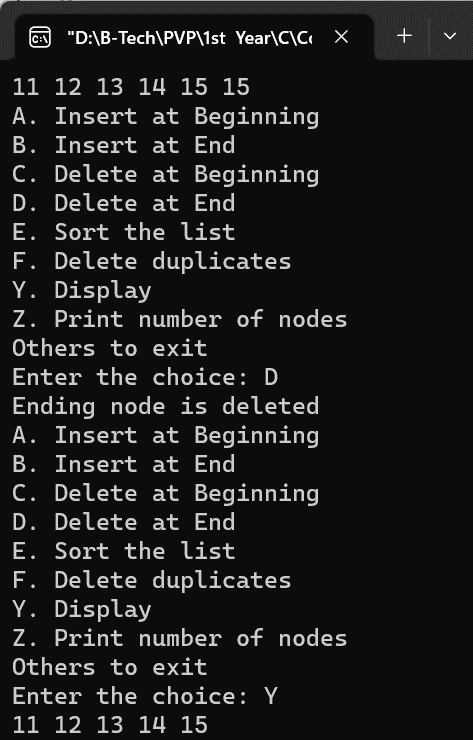
Enter the choice: E

Enter the choice: Y

Output:

11 12 13 14 15 15

11 12 13 14 15



**Test Case-6:** Delete the node from an empty list

Input: List be

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

Z. Print number of nodes

Others to exit

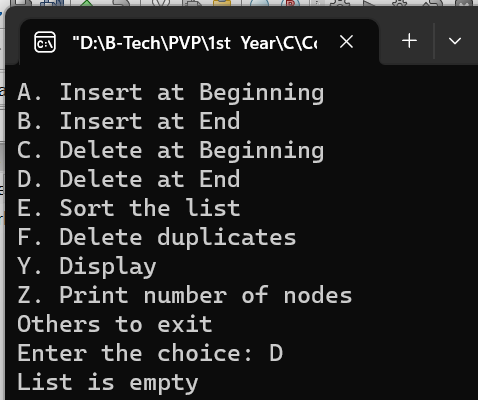
Enter the choice: Y

Enter the choice: F

Enter the position: 16

Output:

List is empty



**Test Case-7:** Sort the LIST

Input: List be 6 5 4 3 2 1

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

Z. Print number of nodes

Others to exit

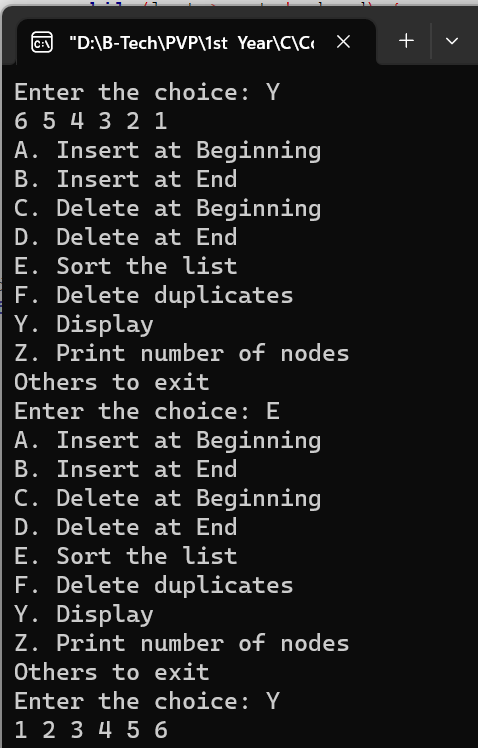
Enter the choice: Y

Enter the choice: E

Enter the choice: Y

Output:

6 5 4 3 2 1

 1 2 3 4 5 6

**Test Case-8:** Given a sorted Circular linked list, delete all duplicates such that each element appear only once.

Input: List be 2 4 4 4 6 6 6 6 6

A. Insert at Beginning

B. Insert at End

C. Delete at Beginning

D. Delete at End

E. Sort the list

F. Delete duplicates

Y. Display

Z. Print number of nodes

Others to exit

Enter the choice: Y

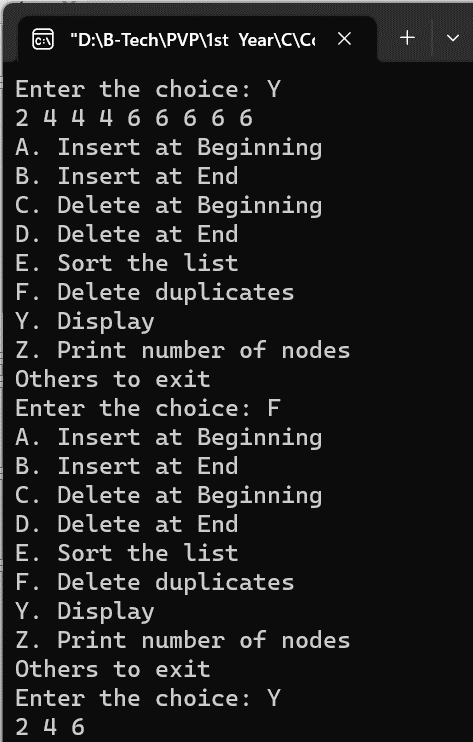
Enter the choice: F

Enter the choice: Y

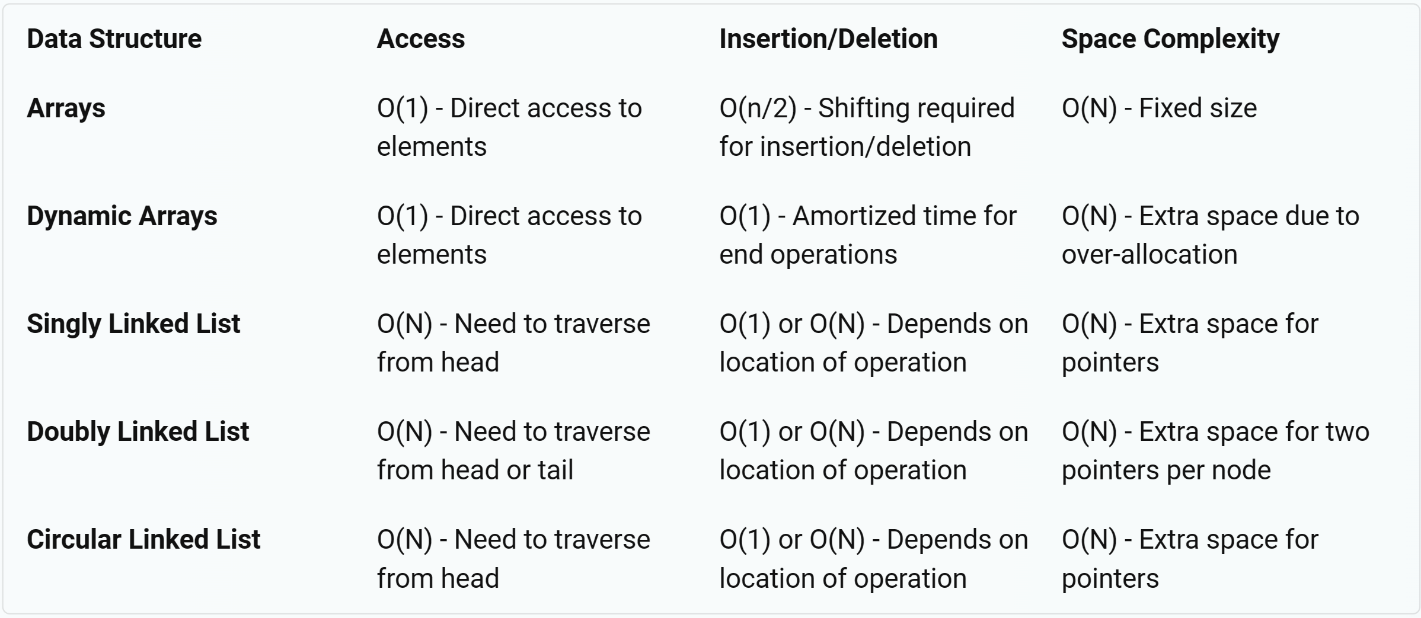
Output:

2 4 4 4 6 6 6 6 6

2 4 6



### **Comparison of Linked Lists with Arrays & Dynamic Arrays:**



### **Viva questions:**

**Single Linked List:**

1. **What is a single linked list?** A single linked list is a type of data structure where each node contains a data field and a reference(link) to the next node in the sequence.
2. **How is a node in a single linked list represented?** A node in a single linked list is represented with two components: data and next. The data component allows storing any kind of data in the node, while the next component is a pointer that points to the next node in the list.
3. **Can you traverse a single linked list in reverse order? Why or why not?** No, you cannot traverse a single linked list in reverse order because each node only has the address of the next node and there is no reference to the previous node.

**Double Linked List:**

1. **What is a double linked list?** A double linked list is similar to a single linked list, but in this case, each node contains a reference to both the next node and the previous node in the sequence.
2. **How is a node in a double linked list represented?** A node in a double linked list is represented with three components: data, next and prev. The data component allows storing any kind of data in the node, while the next and prev are pointers that point to the next and previous nodes in the list respectively.
3. **How does a double linked list allow traversal in both directions?** A double linked list allows traversal in both directions because each node holds references to both the next and previous nodes.

**Circular Linked List:**

1. **What is a circular linked list?** A circular linked list is a variation of a linked list where all nodes are connected to form a circle. There is no NULL at the end. The last node points to the first node in the list.
2. **How does the last node in a circular linked list point to the first node?** In a circular linked list, the last node points to the first node by having its ‘next’ reference point back to the first element of the list instead of pointing to NULL as it would do in a simple linear linked list.