Batch: D2 Roll No.: 16010122323

Experiment No.

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementation of Linked List

Objective: To understand the use of linked list as data structures for various application.

Expected Outcome of Experiment:

CO	Outcome
CO 2	Apply linear and non-linear data structure in application development.

Books/ Journals/ Websites referred:

Introduction:

Define Linked List

A linked list is a data structure used in computer science to organize and store a collection of elements, called nodes, in a linear order. Each node in a linked list contains two parts:

- **Data:** This part holds the actual element or value being stored in the list.
- **Pointer (or reference):** This part points to the next node in the sequence, creating a connection between nodes.

The first node in the list is called the "head," and the last node typically points to a null reference, indicating the end of the list. Linked lists provide flexibility in adding or removing elements, making them suitable for dynamic data structures.

Types of linked list:

Different Types of Linked Lists:

1. Singly Linked List:

In a singly linked list, each node points to the next node in the sequence. It is the simplest form of a linked list.Traversal is one-way, from the head to the end. Commonly used for implementing stacks and queues.

2. Doubly Linked List:

In a doubly linked list, each node points to both the next and the previous nodes in the sequence. Allows for bidirectional traversal, making it more efficient for certain operations. Often used in implementations requiring insertion or deletion from both ends.

3. Circular Linked List:

A circular linked list is a variation of a singly or doubly linked list where the last node points back to the first node, creating a loop. Useful for applications like a circular buffer or representing cyclical data.

4. Skip List:

A skip list is a data structure that uses multiple linked lists with different levels of granularity. Allows for efficient searching, insertion, and deletion of elements, similar to a balanced tree but with simpler implementation.

Each type of linked list has its advantages and is chosen based on specific requirements and use cases in different applications.

Algorithm for creation, insertion, deletion, traversal and searching an element in assigned linked list type:

1. Creating a Doubly Linked List:

Algorithm CreateDoublyLinkedList():

Input: None

Output: A new empty doubly linked list

1. Create a pointer "head" and set it to NULL.

2. Return the "head" as the new doubly linked list.

2. Inserting a Node at the Beginning:

Algorithm InsertAtBeginning(head, data):

Input: A pointer to the head of the doubly linked list, and the data to be inserted.

Output: Updated doubly linked list with the new node at the beginning.

1. Create a new node "newNode" with the given data.

2. Set newNode's next pointer to the current head.

3. If the current head is not NULL, set the previous pointer of the current head to newNode.

4. Set head to newNode.

5. Return the updated head.

3. Inserting a Node at the End:

Algorithm InsertAtEnd(head, data):

Input: A pointer to the head of the doubly linked list, and the data to be inserted.

Output: Updated doubly linked list with the new node at the end.

1. Create a new node "newNode" with the given data.

2. If the list is empty (head is NULL), set head to newNode.

3. Otherwise, traverse the list to find the last node.

4. Set the next pointer of the last node to newNode.

5. Set the previous pointer of newNode to the last node.

6. Return the head.

4. Deleting a Node at the Beginning:

Algorithm DeleteAtBeginning(head):

Input: A pointer to the head of the doubly linked list.

Output: Updated doubly linked list with the first node removed.

1. If the list is empty (head is NULL), return NULL.

2. If head has a next node, set the previous pointer of the next node to NULL.

3. Set head to head's next node.

4. Return the updated head.

5. Deleting a Node at the End:

Algorithm DeleteAtEnd(head):

Input: A pointer to the head of the doubly linked list.

Output: Updated doubly linked list with the last node removed.

1. If the list is empty (head is NULL), return NULL.

2. If there is only one node in the list (head's next is NULL), set head to NULL and return NULL.

3. Traverse the list to find the last node.

4. Set the next pointer of the second-to-last node to NULL.

5. Return the head.

6. Traversing the Doubly Linked List:

Algorithm TraverseDoublyLinkedList(head):

Input: A pointer to the head of the doubly linked list.

Output: Display the elements of the doubly linked list.

1. Initialize a pointer "current" to the head of the list.

2. While current is not NULL:

   a. Display the data of the current node.

   b. Move current to the next node.

3. End.

7. Searching for an Element:

Algorithm SearchElement(head, target):

Input: A pointer to the head of the doubly linked list and the target element to search for.

Output: Return the node containing the target element if found; otherwise, return NULL.

1. Initialize a pointer "current" to the head of the list.

2. While current is not NULL:

   a. If the data in the current node matches the target, return the current node.

   b. Move current to the next node.

3. Return NULL (element not found).

Implementation of an application using linked list:

```
C LINKED LIST.c > main()
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Node* insertAtStart(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = head;
    return newNode;
}

struct Node* insertAtPosition(struct Node* head, int data, int position) {
    struct Node* newNode = createNode(data);
    if (position == 1) {
        newNode->next = head;
        return newNode;
    }
    struct Node* current = head;
    for (int i = 1; i < position - 1 && current != NULL; i++) {
        current = current->next;
    }
    if (current == NULL) {
        printf("Position out of range.\n");
        free(newNode);
        return head;
    }
    newNode->next = current->next;
```

```c
    current->next = newNode;
    return head;
}

struct Node* insertAtEnd(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        return newNode;
    }
    struct Node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
    return head;
}

struct Node* deleteAtStart(struct Node* head) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return NULL;
    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
    return head;
}

struct Node* deleteAtPosition(struct Node* head, int position) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return NULL;
    }
    if (position == 1) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
```

```c
        return head;
    }
    struct Node* current = head;
    struct Node* previous = NULL;
    for (int i = 1; i < position && current != NULL; i++) {
        previous = current;
        current = current->next;
    }
    if (current == NULL) {
        printf("Position out of range.\n");
        return head;
    }
    previous->next = current->next;
    free(current);
    return head;
}

struct Node* deleteAtEnd(struct Node* head) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return NULL;
    }
    if (head->next == NULL) {
        free(head);
        return NULL;
    }
    struct Node* current = head;
    struct Node* previous = NULL;
    while (current->next != NULL) {
        previous = current;
        current = current->next;
    }
    previous->next = NULL;
    free(current);
    return head;
}
```

```c

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, data, position, numElements;

    printf("Enter the number of elements in the linked list: ");
    scanf("%d", &numElements);

    printf("Enter the elements:\n");
    for (int i = 0; i < numElements; i++) {
        scanf("%d", &data);
        head = insertAtEnd(head, data);
    }

    do {
        printf("\n1. Insert at the beginning");
        printf("\n2. Insert at a specified position");
        printf("\n3. Insert at the end");
        printf("\n4. Delete from the beginning");
        printf("\n5. Delete from a specified position");
        printf("\n6. Delete from the end");
        printf("\n7. Display");
        printf("\n8. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```c
                printf("Enter data: ");
                scanf("%d", &data);
                head = insertAtStart(head, data);
                break;
            case 2:
                printf("Enter data: ");
                scanf("%d", &data);
                printf("Enter position: ");
                scanf("%d", &position);
                head = insertAtPosition(head, data, position);
                break;
            case 3:
                printf("Enter data: ");
                scanf("%d", &data);
                head = insertAtEnd(head, data);
                break;
            case 4:
                head = deleteAtStart(head);
                break;
            case 5:
                printf("Enter position: ");
                scanf("%d", &position);
                head = deleteAtPosition(head, position);
                break;
            case 6:
                head = deleteAtEnd(head);
                break;
            case 7:
                display(head);
                break;
            case 8:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
```

```c

    } while (choice != 8);

    return 0;
}
```

OUTPUT

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\ag181\C programs>  & 'c:\Users\ag181\.vscode\ex
rosoft-MIEngine-In-v2gzlkym.bq5' '--stdout=Microsoft-MIEngi
oxxae2n.smh' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--in
Enter the number of elements in the linked list: 10
Enter the elements:
1 2 3 4 5 6 7 8 9 10

1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 1
Enter data: 100

1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 2
Enter data: 57
Enter position: 8

1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 3
Enter data: 99
```

```
1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 4

1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 5
Enter position: 9

1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 6

1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 7
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 57 -> 7 -> 9 -> 10 -> NULL
```

```
1. Insert at the beginning
2. Insert at a specified position
3. Insert at the end
4. Delete from the beginning
5. Delete from a specified position
6. Delete from the end
7. Display
8. Exit
Enter your choice: 8
Exiting...
```

IMPLEMENTATION OF DOUBLY LINKED LIST: (CODE)

C DOUBLY LINKED LIST.c > ...

```c
#include <stdio.h>
#include <stdlib.h>

// Node structure for a doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

// Function to insert a node at the end of the list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
```

```c
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
    }
}

// Function to insert a node in the middle of the list after a given value
void insertInMiddle(struct Node** head, int data, int afterValue) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current != NULL && current->data != afterValue) {
            current = current->next;
        }
        if (current != NULL) {
            newNode->next = current->next;
            if (current->next != NULL) {
                current->next->prev = newNode;
            }
            current->next = newNode;
            newNode->prev = current;
        } else {
            printf("Value %d not found in the list. Insertion failed.\n", afterValue);
            free(newNode);
        }
    }
}

// Function to delete a node at the beginning of the list
void deleteAtBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Deletion failed.\n");
```

```
    } else {
        struct Node* temp = *head;
        *head = (*head)->next;
        if (*head != NULL) {
            (*head)->prev = NULL;
        }
        free(temp);
    }
}

// Function to delete a node at the end of the list
void deleteAtEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Deletion failed.\n");
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        if (current->prev != NULL) {
            current->prev->next = NULL;
        } else {
            *head = NULL;
        }
        free(current);
    }
}

// Function to delete a node with a specific value from the list
void deleteInMiddle(struct Node** head, int data) {
    if (*head == NULL) {
        printf("List is empty. Deletion failed.\n");
    } else {
        struct Node* current = *head;
        while (current != NULL && current->data != data) {
            current = current->next;
        }
```

```c
            if (current != NULL) {
                if (current->prev != NULL) {
                    current->prev->next = current->next;
                } else {
                    *head = current->next;
                }
                if (current->next != NULL) {
                    current->next->prev = current->prev;
                }
                free(current);
            } else {
                printf("Value %d not found in the list. Deletion failed.\n", data);
            }
        }
}

// Function to display the doubly linked list
void display(struct Node* head) {
    printf("Doubly Linked List: ");
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, afterValue;

    while (1) {
        printf("\n------ Doubly Linked List Operations ------\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert in Middle\n");
        printf("4. Delete at Beginning\n");
        printf("5. Delete at End\n");
```

```c
        printf("6. Delete in Middle\n");
        printf("7. Display List\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at the beginning: ");
                scanf("%d", &data);
                insertAtBeginning(&head, data);
                break;
            case 2:
                printf("Enter data to insert at the end: ");
                scanf("%d", &data);
                insertAtEnd(&head, data);
                break;
            case 3:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter the value after which to insert: ");
                scanf("%d", &afterValue);
                insertInMiddle(&head, data, afterValue);
                break;
            case 4:
                deleteAtBeginning(&head);
                break;
            case 5:
                deleteAtEnd(&head);
                break;
            case 6:
                printf("Enter the value to delete: ");
                scanf("%d", &data);
                deleteInMiddle(&head, data);
                break;
            case 7:
                display(head);
```

```c
                break;
            case 8:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

OUTPUT

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvement

PS C:\Users\ag181\.1-VS CODE PROGRAMS\DataStruct Programs>  &
bugLauncher.exe' '--stdin=Microsoft-MIEngine-In-mdpvrsva.tjp'
-pid=Microsoft-MIEngine-Pid-vpplhtoo.mvy' '--dbgExe=C:\msys64\

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 34

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 35

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 36
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 37

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 38

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 39

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
8. Exit
Enter your choice: 1
Enter data to insert at the beginning: 40

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 2
Enter data to insert at the end: 41

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 3
Enter data to insert: 43
Enter the value after which to insert: 39

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 4

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

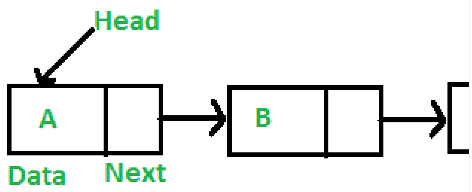
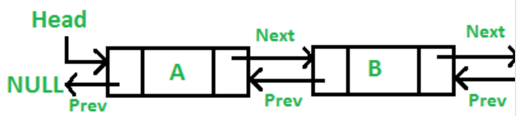
```
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 5

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 6
Enter the value to delete: 39

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 7
Doubly Linked List: 43 -> 38 -> 37 -> 36 -> 35 -> 34 -> NULL

------ Doubly Linked List Operations ------
1. Insert at Beginning
2. Insert at End
3. Insert in Middle
4. Delete at Beginning
5. Delete at End
6. Delete in Middle
7. Display List
8. Exit
Enter your choice: 8
PS C:\Users\ag181\.1-VS CODE PROGRAMS\DataStruct Programs>
```

Conclusion:

Hence the implementation of various operations of doubly linked list was learnt and performed accordingly

Post lab questions:

1. Compare and contrast SLL and DLL

Singly linked list (SLL)	Doubly linked list (DLL)
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.
	
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Complexity of insertion and deletion at a given position is O(n).	Complexity of insertion and deletion at a given position is O(n / 2) = O(n) because traversal can be made from start or from the end.
Complexity of deletion with a given node is O(n), because the	Complexity of deletion with a given node is O(1) because the

Singly linked list (SLL)	Doubly linked list (DLL)
previous node needs to be known, and traversal takes O(n)	previous node can be accessed easily
We mostly prefer to use singly linked list for the execution of stacks.	We can use a doubly linked list to execute heaps and stacks, binary trees.
When we do not need to perform any searching operation and we want to save memory, we prefer a singly linked list.	In case of better implementation, while searching, we prefer to use doubly linked list.
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.
Singly linked list is less efficient. It is preferred when we need to save memory and searching is not required as pointer of single index is stored.	Doubly linked list is more efficient. When memory is not the problem and we need better performance while searching, we use doubly linked list.