
# Green University of Bangladesh
# Department of Computer Science and Engineering (CSE)

**Faculty of Sciences and Engineering**
**Semester:( Spring, Year:2024), B.Sc. in CSE (Day)**

**Lab Report NO: 04**

**Course Title: Data Structure Lab**
**Course Code: CSE 106 Section: 231-D1**

**Lab Experiment Name: Implementation of Doubly Linked List.**

**Student Details**

| | Name | ID |
|---|---|---|
| 1. | Promod Chandra Das | 231002005 |

**Lab Date :**
**Submission Date :**
**Course Teacher's Name : Md. Shihab Hossain**

**Lab Report Status**

**Marks: ……** **Signature:.....**
**Comments:.....** **Date:.....**

# ➢ Objective(s)

- To attain knowledge on **doubly linked list**.
- To implement **doubly linked list** using C.

# ➢ Problem Analysis

**A linked List** is a linear collection of data elements whose order is not given by their physical placement in memory. The elements in a linked list are linked using pointers. It is a data structure consisting of a collection of nodes which together represent a sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.

**Doubly Linked List** - It is also known as two way linked list. A two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in sequence, Therefore, it contains three parts are data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well.
The nodes are connected to each other in back and forth where the value of the next variable of the last node is NULL i.e. next = NULL, which indicates the end of the doubly linked list and value of the previous variable of the first node is NULL i.e. previous = NULL, which indicates the beginning of the doubly linked list

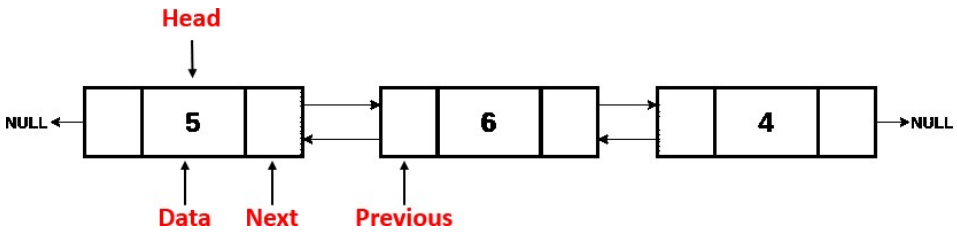


Figure 1: Doubly Linked List

## ✓ Basic Operations of a Doubly Linked List

- **Insert**
  - **At the Beginning Position**
  - **At the Last Position**
  - **At any Specific Position**
- **Delete**
  - **From the Beginning Position**
  - **From the Last Position**
  - **From any Specific Position**
- **Traverse**
- **Display**
- **Search**

- # Lab Exercise (Submit as a report)

1. Find the specific node of element that is present or not in the singly linked list.
2. Call a function that will generate the size of the singly linked list.
3. Insert an element between any specific position of the singly linked list.

## ✓ Answer to the q no : 01

- Find the specific node of element that is present or not in the singly linked list.

```
        #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;
}

void append(struct Node** head_ref, int data) {
   struct Node* newNode = createNode(data);
   struct Node* last = *head_ref;

   if (*head_ref == NULL) {
      *head_ref = newNode;
      return;
   }

   while (last->next != NULL) {
      last = last->next;
   }

   last->next = newNode;
}

int find(struct Node* head, int target) {
   struct Node* current = head;
```

```
    while (current != NULL) {
        if (current->data == target) {
            return 1; // Element found
        }
        current = current->next;
    }
    return 0; // Element not found
}

int main() {
    struct Node* head = NULL;

    append(&head, 1);
    append(&head, 2);
    append(&head, 3);
    append(&head, 4);

    int target = 3;
    if (find(head, target)) {
        printf("Element %d is present in the list.\n",
target);
    } else {
        printf("Element %d is not present in the
list.\n", target);
    }

    target = 5;
    if (find(head, target)) {
        printf("Element %d is present in the list.\n",
target);
    } else {
        printf("Element %d is not present in the
list.\n", target);
    }

    return 0;
}
```

## ✓ Answer to the q no : 02

- Call a function that will generate the size of the singly linked list.

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

// Define the node structure
```

```
struct Node {
    int data;
    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->next = NULL;
    return newNode;
}

// Function to append a node to the end of the list
void append(struct Node** head_ref, int data) {
    struct Node* newNode = createNode(data);
    struct Node* last = *head_ref;

    if (*head_ref == NULL) {
        *head_ref = newNode;
        return;
    }

    while (last->next != NULL) {
        last = last->next;
    }

    last->next = newNode;
}

// Function to calculate the size of the linked list
int getSize(struct Node* head) {
    int count = 0;
    struct Node* current = head;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

// Main function to test the linked list
int main() {
    struct Node* head = NULL;

    append(&head, 1);
    append(&head, 2);
    append(&head, 3);
    append(&head, 4);
```

```
    // Call the function to generate the size of the
linked list
    int size = getSize(head);
    printf("The size of the linked list is: %d\n", size);

    append(&head, 5);

    // Call the function again after adding another
element
    size = getSize(head);
    printf("After adding another element, the size of
the linked list is: %d\n", size);

    return 0;
}
```

- ✓ **Answer to the q no : 03**
  - Insert an element between any specific position of the singly linked list.

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


// Define the node structure
struct Node {
    int data;
    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->next = NULL;
    return newNode;
}

// Function to append a node to the end of the list
void append(struct Node** head_ref, int data) {
    struct Node* newNode = createNode(data);
    struct Node* last = *head_ref;

    if (*head_ref == NULL) {
        *head_ref = newNode;
        return;
    }

    while (last->next != NULL) {
        last = last->next;
    }

    last->next = newNode;
}

// Function to insert a node at a specific position
void insertAtPosition(struct Node** head_ref, int data, int position) {
    struct Node* newNode = createNode(data);

    // If inserting at the head (position 0)
    if (position == 0) {
        newNode->next = *head_ref;
        *head_ref = newNode;
        return;
    }

    struct Node* current = *head_ref;
    for (int i = 0; current != NULL && i < position - 1; i++) {
```

```
        current = current->next;
    }

    // If position is more than number of nodes, insert at the end
    if (current == NULL) {
        printf("Position is greater than the number of nodes. Inserting at the end.\n");
        append(head_ref, data);
        return;
    }

    newNode->next = current->next;
    current->next = newNode;
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}

// Main function to test the linked list
int main() {
    struct Node* head = NULL;

    append(&head, 1);
    append(&head, 2);
    append(&head, 4);
    append(&head, 5);

    printf("Original List:\n");
    printList(head);
```

```
    // Insert element at specific positions
    insertAtPosition(&head, 3, 2); // Inserting 3 at position 2
    printf("\nList after inserting 3 at position 2:\n");
    printList(head);

    insertAtPosition(&head, 0, 0); // Inserting 0 at position 0 (head)
    printf("\nList after inserting 0 at position 0:\n");
    printList(head);

    insertAtPosition(&head, 6, 10); // Inserting 6 at position 10 (beyond the list size)
    printf("\nList after attempting to insert 6 at position 10:\n");
    printList(head);

    return 0;
}
```

## ➢ Discussion & Conclusion

The implementation of a doubly linked list involves nodes with pointers to both their next and previous nodes, facilitating bidirectional traversal. This allows efficient insertion and deletion at both ends but requires additional memory for the extra pointer. Doubly linked lists are beneficial for complex data manipulation compared to singly linked lists.