



The University of Azad Jammu and Kashmir

Name: Zarnab Fatima

Roll NO: 2024-SE-14

Course Title: DSA

Course Code: CS-2101

Submitted To: Engr. Sidra Rafique

Department of Software Engineering

LAB 04

Linked List:

A Linked List is a linear data structure where elements (called nodes) are connected using pointers. Unlike arrays, the elements are not stored at contiguous memory locations.

Structure of a Node

Each node typically contains:

1. Data → Stores the value of the element.
2. Pointer (next) → Points to the next node in the list.

```
+----+----+
| data | next |
+----+----+
```

- Head → Points to the first node.
- Tail / NULL → Last node points to NULL indicating the end.

Types of Linked Lists:

1. Singly Linked List

- Each node points to the next node only.
- Traversal is forward only.

2. Doubly Linked List

- Each node points to both previous and next nodes.
- Traversal is forward and backward.

3. Circular Linked List

- Last node points back to the first node.
- Can be singly or doubly linked.

Advantages

- Dynamic size (can grow/shrink at runtime)
- Easy insertion and deletion (no need to shift elements like arrays)

Disadvantages

- Extra memory for pointer(s)
 - Slower access compared to arrays (no direct indexing)
-

Doubly Linked List (DLL)

A Doubly Linked List (DLL) is a type of linked list in which each node contains a data part and two pointers:

1. `*prev*` → Points to the previous node
2. `*next*` → Points to the next node

This allows traversal in both directions (forward and backward), unlike a singly linked list which can only be traversed in one direction.

Structure of a Node

```
+----+----+----+
| prev | data | next |
+----+----+----+
```

- `prev` → memory address of the previous node (or NULL for the first node)
- `data` → stores the value
- `next` → memory address of the next node (or NULL for the last node)

Advantages of Doubly Linked List

Can traverse forward and backward.

Easier to delete a node when a pointer to it is given (no need to traverse from head).

Can efficiently insert at both ends.

Basic Operations

1. Insert at beginning → Update head and prev pointers.
2. Insert at end → Update tail and next pointers.
3. Delete a node → Update prev and next pointers of adjacent nodes.
4. Display forward → Start from head and follow next pointers.
5. Display backward → Start from tail and follow prev pointers.

Code:

The screenshot shows a code editor window with a black background and white text. The file is named 'LINKED.cpp'. The code defines a 'Node' structure and a 'DoublyLinkedList' class. The 'Node' structure contains an integer 'data' and pointers to the previous and next nodes ('prev' and 'next'). It also includes a constructor that initializes 'data' to 'val', 'prev' to 'nullptr', and 'next' to 'nullptr'. The 'DoublyLinkedList' class has private members 'head' and 'tail', and a public constructor that initializes 'head' to 'nullptr'.

```
LINKED.cpp
1  #include <iostream>
2  using namespace std;
3
4  // Node structure
5  class Node {
6  public:
7      int data;
8      Node* prev;
9      Node* next;
10
11     Node(int val) {
12         data = val;
13         prev = nullptr;
14         next = nullptr;
15     }
16 }
17
18 // Doubly Linked List class
19 class DoublyLinkedList {
20 private:
21     Node* head;
22     Node* tail;
23
24 public:
25     DoublyLinkedList() {
26         head = nullptr;
27     }
28 }
```

Figure 1

```
LINKED.cpp
25     DoublyLinkedList() {
26         head = nullptr;
27         tail = nullptr;
28     }
29
30     // 1. Insert at the beginning
31     void insertAtBeginning(int val) {
32         Node* newNode = new Node(val);
33         if (!head) { // Empty list
34             head = tail = newNode;
35         } else {
36             newNode->next = head;
37             head->prev = newNode;
38             head = newNode;
39         }
40         cout << val << " inserted at the beginning.\n";
41     }
42
43     // 2. Insert at the end
44     void insertAtEnd(int val) {
45         Node* newNode = new Node(val);
46         if (!tail) { // Empty list
47             head = tail = newNode;
48         } else {
49             tail->next = newNode;
50             newNode->prev = tail;
51         }
52         tail = newNode;
53     }
54
55     // 3. Delete from the beginning
56     void deleteFromBeginning() {
57         if (!head) {
58             cout << "List is empty. Nothing to delete.\n";
59             return;
60         }
61         Node* temp = head;
62         cout << "Deleting " << temp->data << " from the beginning.\n";
63         head = head->next;
64         if (head)
65             head->prev = nullptr;
66         else
67             tail = nullptr; // List becomes empty
68         delete temp;
69     }
70 }
```

Figure 2

```
LINKED.cpp
49         tail->next = newNode;
50         newNode->prev = tail;
51         tail = newNode;
52     }
53     cout << val << " inserted at the end.\n";
54 }
55
56     // 3. Delete from the beginning
57     void deleteFromBeginning() {
58         if (!head) {
59             cout << "List is empty. Nothing to delete.\n";
60             return;
61         }
62         Node* temp = head;
63         cout << "Deleting " << temp->data << " from the beginning.\n";
64         head = head->next;
65         if (head)
66             head->prev = nullptr;
67         else
68             tail = nullptr; // List becomes empty
69         delete temp;
70     }
71 }
```

Figure 3

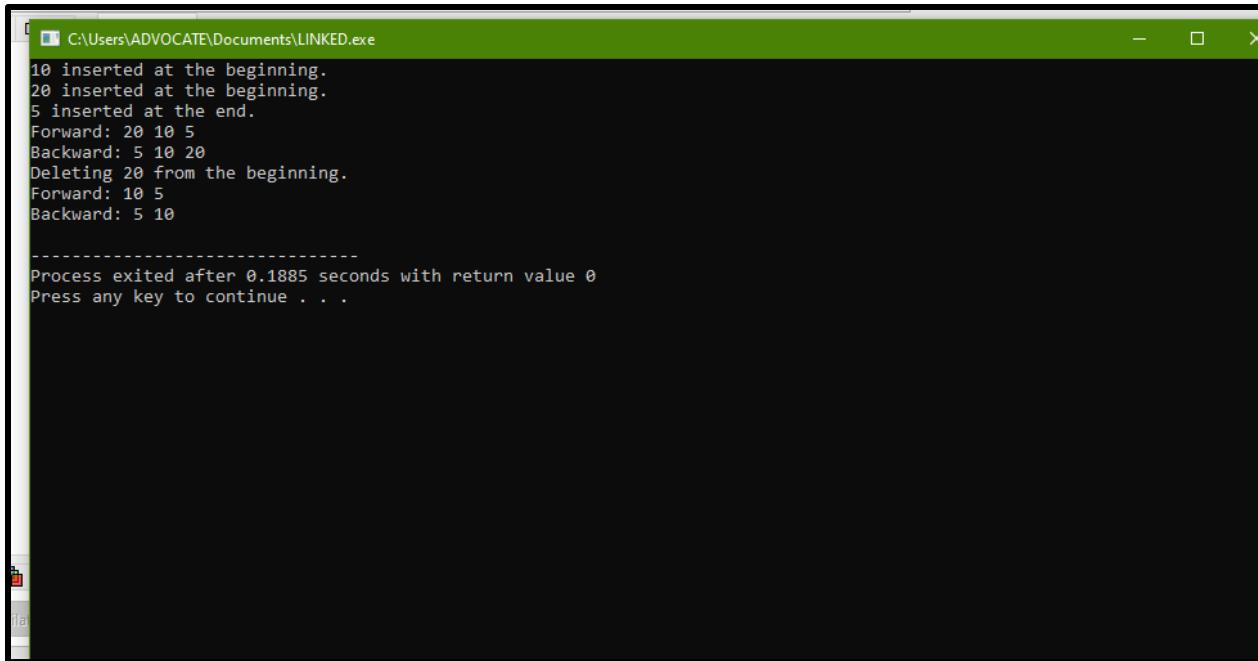
```
LINKED.cpp
70
71
72
73 // 4. Display forward
74 void displayForward() {
75     if (!head) {
76         cout << "List is empty.\n";
77         return;
78     }
79     Node* temp = head;
80     cout << "Forward: ";
81     while (temp) {
82         cout << temp->data << " ";
83         temp = temp->next;
84     }
85     cout << endl;
86 }
87
88 // Display backward
89 void displayBackward() {
90     if (!tail) {
91         cout << "List is empty.\n";
92         return;
93     }
94     Node* temp = tail;
95     cout << "Backward: ";
96     while (temp) {
97         cout << temp->data << " ";
98         temp = temp->prev;
99     }
100    cout << endl;
101 }
102
103 // Main function to test the doubly linked list
104 int main() {
105     DoublyLinkedList list;
106
107     list.insertAtBeginning(10);
108     list.insertAtBeginning(20);
109     list.insertAtEnd(5);
110     list.displayForward();
111     list.displayBackward();
112
113     list.deleteFromBeginning();
114     list.displayForward();
115     list.displayBackward();
116
117     return 0;
118 }
```

Figure 4

```
LINKED.cpp
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
```

Figure 5

Output:



A screenshot of a Windows command-line interface window titled "C:\Users\ADVOCATE\Documents\LINKED.exe". The window contains the following text output:

```
10 inserted at the beginning.
20 inserted at the beginning.
5 inserted at the end.
Forward: 20 10 5
Backward: 5 10 20
Deleting 20 from the beginning.
Forward: 10 5
Backward: 5 10

-----
Process exited after 0.1885 seconds with return value 0
Press any key to continue . . .
```

Explanation of Operations

1. Insert at Beginning: Adds a new node at the start and updates head.
2. Insert at End: Adds a new node at the tail and updates tail.
3. Delete from Beginning: Removes the first node and updates head.
4. Display Forward & Backward: Traverses the list using next and prev pointers.

THANK YOU