Data Structures
Lab Manual 4

Topic: Doubly Link List

Session: Spring 2023

Faculty of Information Technology

UCP Lahore Pakistan

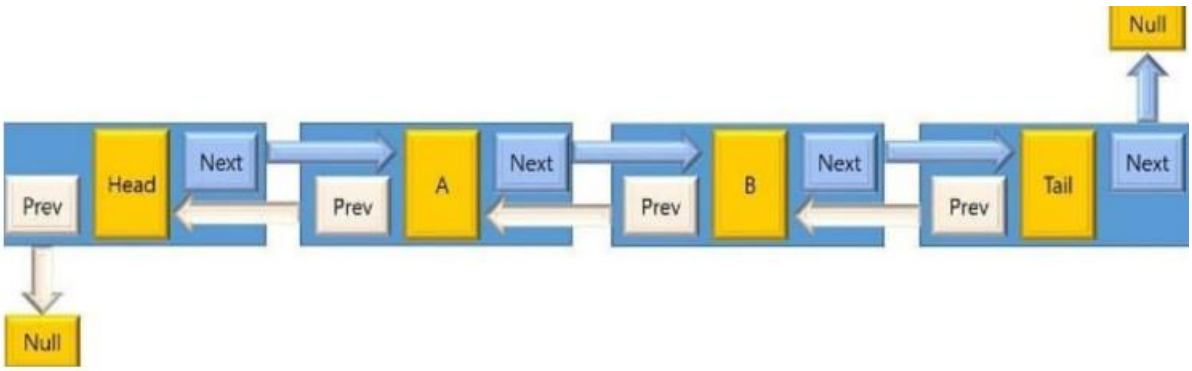
Objectives:

- Efficient traversal in both directions (forward and backward).
- Efficient insertion and deletion of nodes at any position in the list.
- Flexibility in managing nodes in the list, such as easily swapping two adjacent nodes or reversing the order of the nodes in the list
- Efficient implementation of other data structures, such as stacks, queues, and hash tables, which can be built on top of a doubly linked list.

Unlike a singly linked list, which only allows traversal in one direction (forward), a doubly linked list allows traversal in both directions (forward and backward), making it easier to insert or delete nodes at any position in the list.

Doubly Link List:

A doubly linked list is a type of data structure in which each node contains a data element and two pointers, one pointing to the previous node and the other pointing to the next node. This allows for efficient traversal in both forward and backward directions, making it useful for applications such as implementing a music playlist. However, it requires more memory than a singly linked list due to the additional pointer for each node.



The head node contains a pointer to the first node in the list. Since a doubly linked list can be traversed in both directions, the head node also contains a null pointer for the previous node.

The last node, is the final node in the list and contains a null pointer for the next node. It also contains a pointer to the second to last node in the list, allowing for traversal in reverse order.

By maintaining pointers to both the head and last nodes, we can easily insert or delete nodes at both the beginning and end of the list.

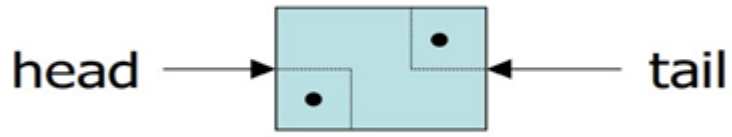
We can implement this organization by storing the information and handle the next and previous element with the other data as follows:-

```
class Doublelistnode
{
        Node *prev;          // Pointer to previous node
        Node  *next;         // Pointer to next node
        int      data;
};
```

Basic Operations that can be performed on Linked List:

- Traversal:
  To traverse all the nodes one after another
- Append a new node:
  To add a new node at the end
- Prepend a new node:
  To add a new node at the beginning
- Insertion:
  To add a node at the given position
- Deletion:
  To delete a node from the list
- Updating:
  To update a node in the list
- Searching:
  To search an element by value
- Sorting:
  To arrange nodes in a linked list in a specific order
- Calculate length:
  To count number of nodes in the list

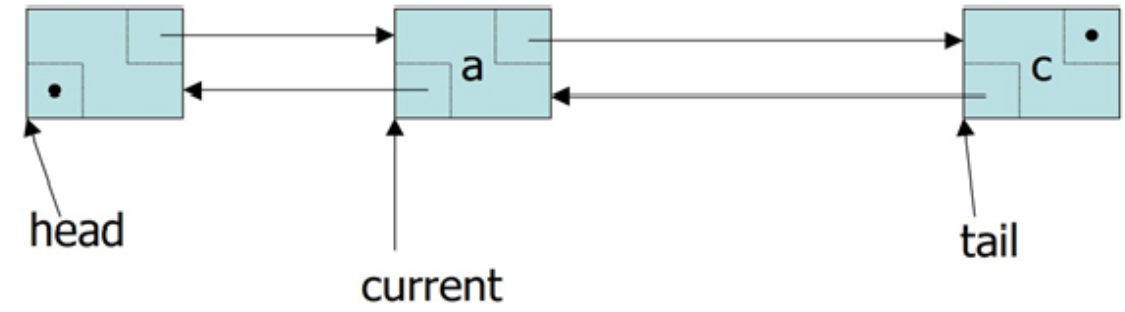
Adding first Node

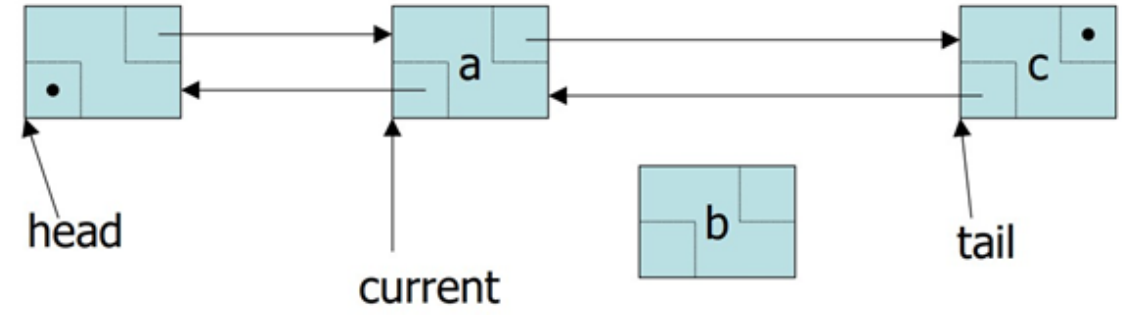


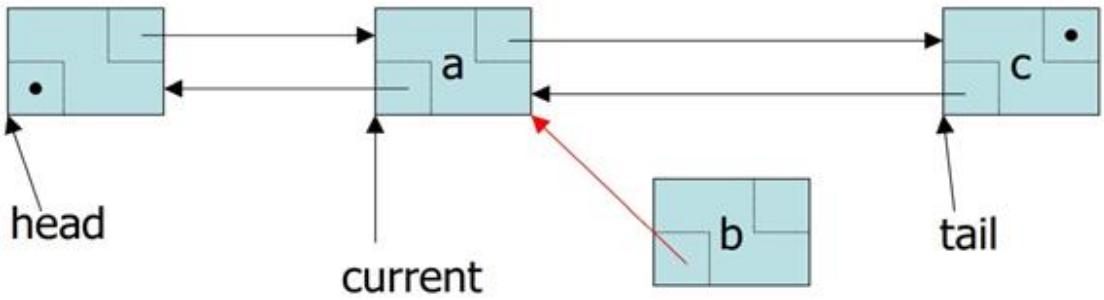

```
head = new Doublelistnode;
head->next = null;
head->prev = null;
tail = head;
```

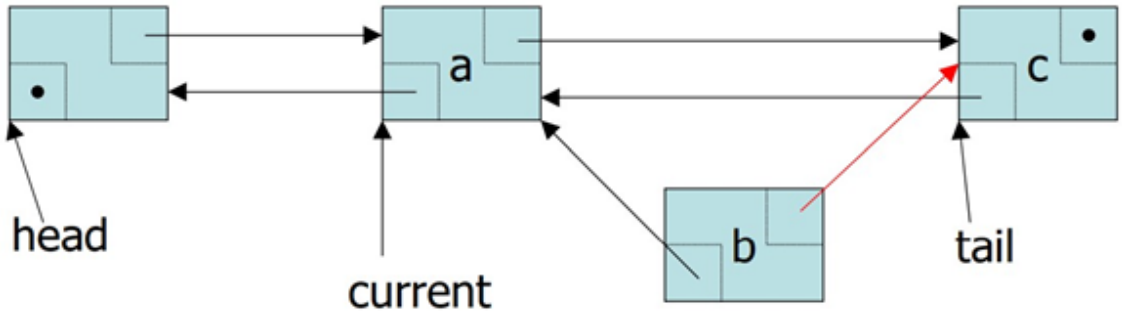
Inserting a Node in a doubly-linked List

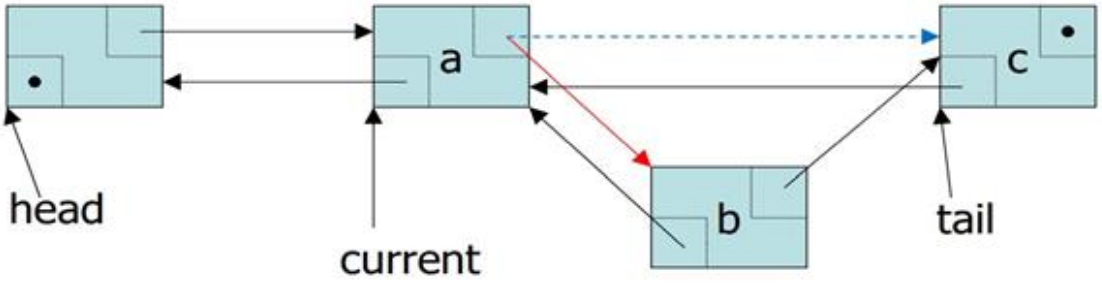
Add a new item after the linked list node pointed by current.

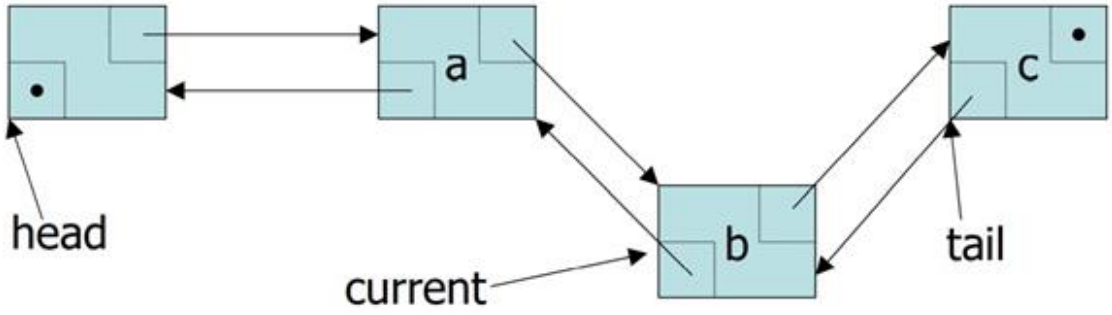








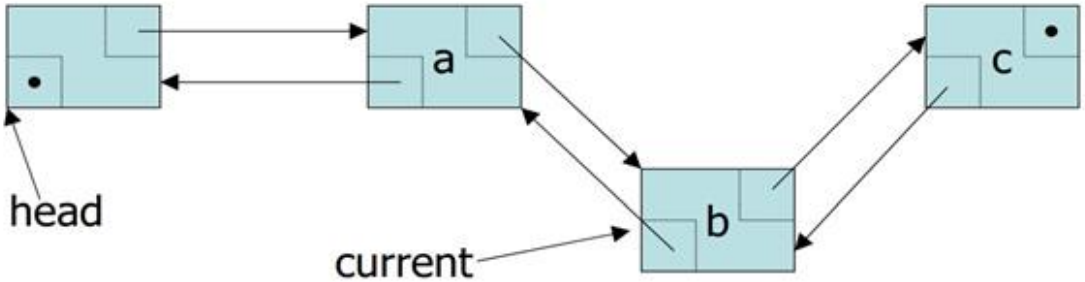


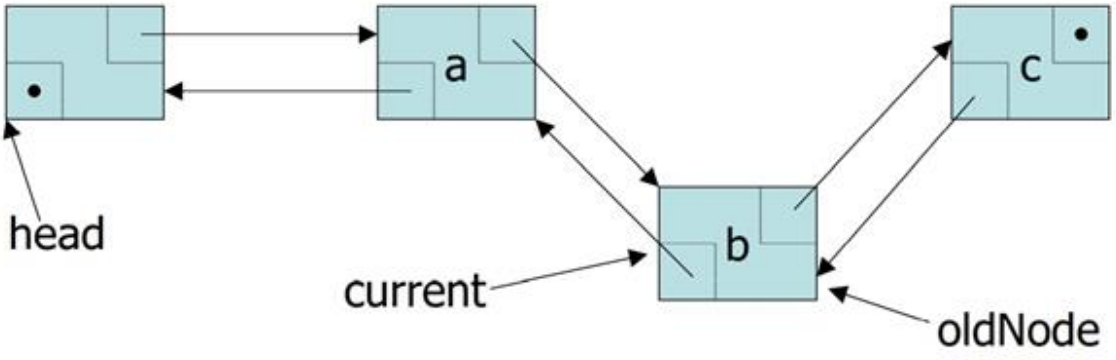


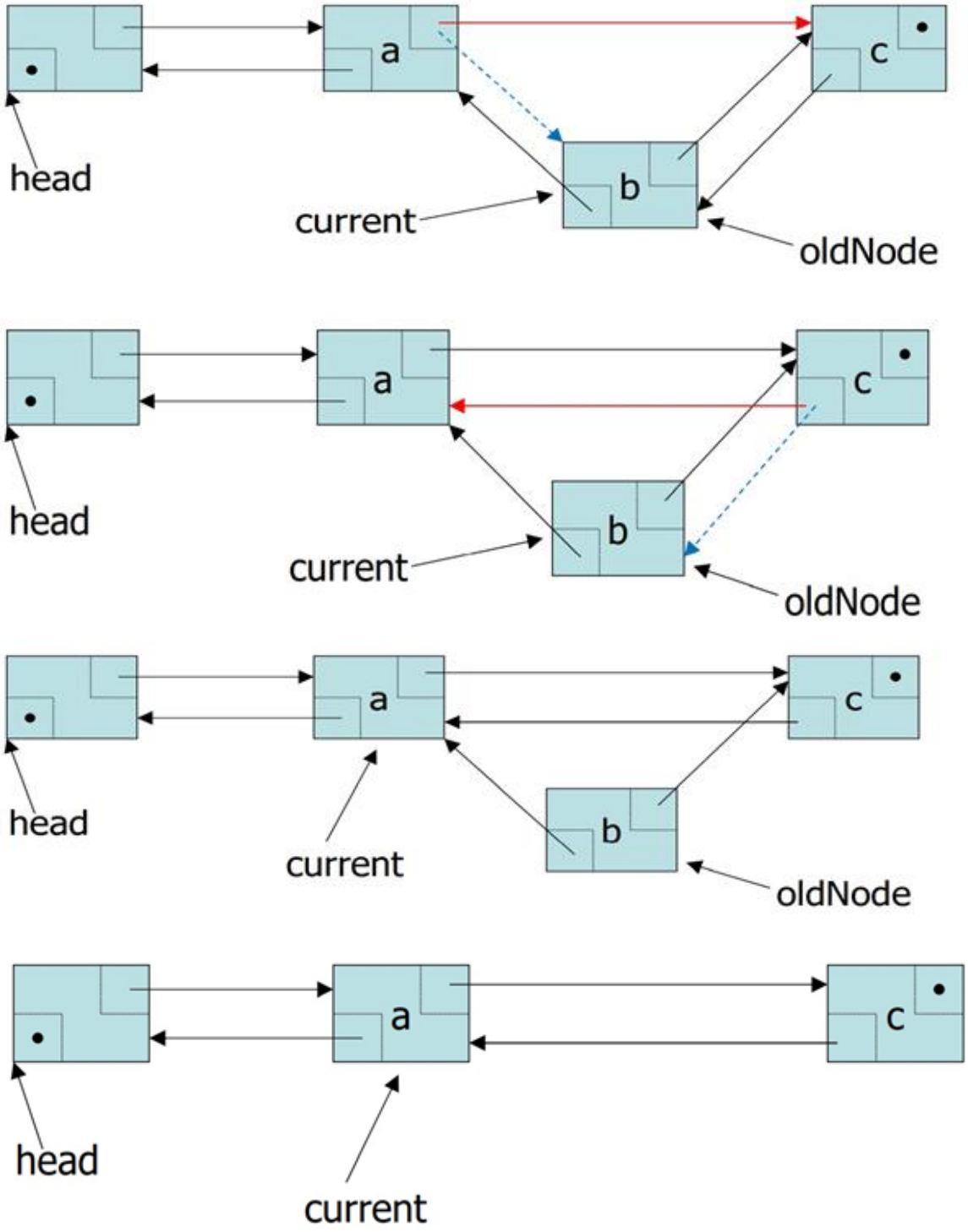

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

Deleting a Node from Doubly-linked List

Suppose Current points to the node to be removed from the list








```
oldNode = current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Time Complexity of doubly linked list

Function	Time Complexity
Traverse all elements	O(n)
Insert at front	O(1)
Insert at end	O(1)
Insert after a node (if position of the node is unknown and need to be traverse)	O(n)
Delete at front	O(1)
Delete at end	O(1)
Delete after a node (if position of the node is unknown and need to be traverse)	O(n)

Sample Task

A program that uses class Doublelinkedlist to create a double linked list of integers and then display it on console.

Implementation:

```
#include <iostream>

using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node* prev;
};

class Doublelinkedlist {
public:
    Node* head;
    Node* tail;
    Doublelinkedlist() {
        head = NULL;
        tail = NULL;
    }
    void addNode(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = NULL;
        if (head == NULL) {
            newNode->prev = NULL;
            head = newNode;
            tail = newNode;
        }
        else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
    Node* getHead() {
        return head;
    }
    Node* getTail() {
        return tail;
    }
};

int main() {
    Doublelinkedlist myList;
    myList.addNode(33);
    myList.addNode(56);
    myList.addNode(21);
    myList.addNode(44);
    Node* currentNode = myList.getHead();
    while (currentNode != NULL) {
        cout << currentNode->data << " ";
        currentNode = currentNode->next;
    }
    return 0;
}
```

Task

Create your own class of ***MyDoublyLinkedList*** with getter and setter methods for its nodes (Data, Next And Prev) and implement the following functions.

- Function called InsertAtBegin to add node at the begging of list.
- Function called InsertAtEnd to add node at the last of list.
- Function called InsertAt to insert new value anywhere in list.
- Function called DeleteFromBegin to delete node from the begging of list.
- Function called DeleteFromEnd to delete node from the last of list.
- Function called DeleteMax to delete maximum value in list.
- Function called DeleteList to delete the complete list.
- Function called Search to search any node from the list.
- Function called Display to display list on console.
- Function called ReverseDisplay to display the complete list in reverse order.

Note: ***Your program should be menu based and should ask the user what action he wishes to perform***.

First, work with a linked list that contains a single attribute, such as a name or an ID number. Perform each of the above-mentioned operations on this list. Next, update the list to include more attributes, such as an email address and a phone number, and perform the same operations on the updated list.