Name	Shubhan Singh
UID no.	2022300118
Experiment No.	4

Program 1	
PROBLEM STATEMENT :	**Implement ADT for storing a DLL and performing given operations on it. Use dummy nodes - header and trailer, in your implementation. The header points to the first node in the DLL and the trailer points to the last node in the DL**
Program:	see below

```
/*
 * File: dll_operations2.c
 * Author: Siddhartha Chandra
 * Email: siddhartha_chandra@spit.ac.in
 * Created: September 24, 2023
 * Description: This implements ADT for storing a DLL and
performing listed operations on it.
 * Use dummy nodes - header and trailer, in your
implementation
 */


#include<string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>


typedef struct Node {
    int val;
    struct Node* prev;
    struct Node* next;
} Node;


Node* create_node(int val){
    Node* header=malloc(sizeof(Node));
    Node* trailer=malloc(sizeof(Node));
    Node* first=malloc(sizeof(Node));
    header->val=trailer->val=0;
    header->next=first;
    first->prev=header;
    header->prev=trailer->next=NULL;
    first->next=trailer;
    trailer->prev=first;
    first->val=val;
    return header;
}

// pos == 0 indicates start of the DLL
// pos == -1 indicates end of the DLL
// pos == n indicates intermediate node pos
// val --> val that needs to be inserted
// is_after = true => insert after 'pos'
// is_after = false => insert after 'pos'
void insert_at_pos(Node* header, Node* trailer, int val,
int pos, bool is_after) {
    Node* iter;
    if(pos==-1){
        iter=trailer->prev;
        is_after=true;
    }
    else if(pos==0){
        iter=header->next;
        is_after=false;
    }
    else{
        iter=header->next;
        for(int i=0;i<pos;i++){
            if(iter->next==trailer){
                break;
            }
            iter=iter->next;
        }
```

```c
    }
    Node* new=malloc(sizeof(Node));
    new->val=val;
    if(is_after){
        new->prev=iter;
        new->next=iter->next;
        iter->next=new;
        new->next->prev=new;
    }
    else{
        new->next=iter;
        new->prev=iter->prev;
        iter->prev=new;
        new->prev->next=new;
    }
}

// pos == 0 indicates start of the DLL
// pos == -1 indicates end of the DLL
// pos == n indicates intermediate node pos
void delete_at_pos(Node* header, Node* trailer, int pos)
{
    Node* iter;
    if(pos==-1)
    iter=trailer->prev;
    else{
        iter=header->next;
        for(int i=0;i<pos;i++){
            if(iter->next==trailer){
                break;
            }
            iter=iter->next;
        }
    }
    iter->prev->next=iter->next;
    iter->next->prev=iter->prev;
    free(iter);
}
```

```
// delete every alternate node starting from the first
node in the list.
// ex: delete_alternates([header, 1, 2, 3, 4, trailer])
=> [header, 2, 4, trailer]
void delete_alternates(Node* header){
    Node* iter=header->next;
    int i=0;
    while(true){
        if(i%2==0){
            iter->prev->next=iter->next;
            iter->next->prev=iter->prev;
            Node* temp=iter;
            iter=iter->next;
            free(temp);
        }
        else{
            iter=iter->next;
        }
        if(iter->next==NULL){
            break;
        }
        i++;

    }
}

// merge 2 sorted DLL's. Ensure that the order property
is preserved
// ex: merge_sorted_dlls([1,3,5], [2,4,6]) => [1, 2, 3,
4, 5, 6]
void merge_sorted_dlls(Node* header1, Node* header2){
    Node* liter=header1->next;
    Node* riter=header2->next;
    Node* temp;
    while(true){
        if(riter->next==NULL){
```

```c
            free(riter);
            break;
        }
        if(liter->val >= riter->val){
            temp=riter->next;
            riter->next=liter;
            riter->prev=liter->prev;
            liter->prev->next=riter;
            liter->prev=riter;
            riter=temp;
        }
        else{
            liter=liter->next;
        }
        if(liter->next==NULL){
            riter->prev=liter->prev;
            liter->prev->next=riter;
            free(liter);
            break;
        }
    }
}


void display(Node *header, Node* trailer){
    Node* iter=header->next;
    while(true){
        if(iter==trailer){
            break;
        }
        if(iter!=header->next){
            printf("<-->");
        }
        printf("%d",iter->val);
        iter=iter->next;
    }
    printf("\n");
}
```

```
void freelist(Node* header,Node* trailer){
    Node* temp;
    Node* iter=header->next;
    while(iter!=trailer){
        temp=iter;
        iter=iter->next;
        free(temp);
    }
    free(header);
    free(trailer);
}

void operate(Node* header,Node* trailer, int operator){
    if(operator==0){
        return;
    }
    else if(operator==1){
        int no;
        printf("Enter number of elements to insert: ");
        scanf("%d",&no);
        int temp,pos,after;
        bool isafter;
        while((getchar())!='\n');
        for(int i=0;i<no;i++){
            printf("Enter value, position, whether to
insert before or after that position(0 for before, 1 for
after):\n");
            scanf("%d%d%d",&temp,&pos,&after);
            isafter=after==0?false:true;

insert_at_pos(header,trailer,temp,pos,isafter);
        }
        return;
    }
    else if(operator==2){
        printf("Enter position to delete:\n");
        int pos;
        scanf("%d",&pos);
        delete_at_pos(header,trailer,pos);
    }
    else if(operator==3){
        delete_alternates(header);
    }
    else if(operator==4){
        Node* header1;
        Node* header2;
        int temp;
        int size1,size2;
        printf("Enter the size of 2 sorted DLLs\n");
        scanf("%d%d",&size1,&size2);
        printf("Enter the elements of the first DLL\n");
        scanf("%d",&temp);
        header1=create_node(temp);
        Node* trailer1=header1->next->next;
        for(int i=0;i<size1-1;i++){
            scanf("%d",&temp);
            insert_at_pos(header1,trailer1,temp,-1,true);
        }
        printf("Enter the elements of the second DLL\n");
        scanf("%d",&temp);
        header2=create_node(temp);
        Node* trailer2=header2->next->next;
        for(int i=0;i<size2-1;i++){
            scanf("%d",&temp);
            insert_at_pos(header2,trailer2,temp,-1,true);
        }
        merge_sorted_dlls(header1,header2);
        trailer1=header1->next;
        while(trailer1->next!=NULL){
            trailer1=trailer1->next;
        }
        printf("The merged DLL is:\n");
        display(header1,trailer1);
    }
    else if(operator==5){
```

```c
        display(header,trailer);
    }
    else if(operator==6){
        freelist(header,trailer);
    }
    else{
        printf("Invalid input\n");
    }
}

int main(){
    int val;
    printf("Enter the first value of the list:\n");
    scanf("%d",&val);
    Node* header=create_node(val);
    Node* trailer=header->next->next;
    int input=-1;
    while(input){
        printf("Enter\n1 to insert at position\n2 to delete at position\n3 to delete alternate elements\n4 to merge dorted DLLs\n5 to display DLL\n6 to free the DLL\n0 to end program\n");
        scanf("%d",&input);
        operate(header,trailer,input);
        if(input==6){input=0;}
    }
}
```

Algorithm:

Here are algorithms for each of the methods in the given doubly linked list implementation in C language:

1. **create_node(int val)**:
    - Description: Creates a new doubly linked list node with the given value and returns a pointer to it.
    - Algorithm:
        1. Allocate memory for three nodes: **header**, **trailer**, and **first**.
        2. Initialize **header** and **trailer** values to 0.
        3. Set **header**'s next to **first** and **trailer**'s next to NULL.
        4. Set **first**'s prev to **header** and next to **trailer**.

        5. Set **header**'s prev and **trailer**'s prev to NULL.
        6. Set **first**'s value to the given **val**.
        7. Return **header**.

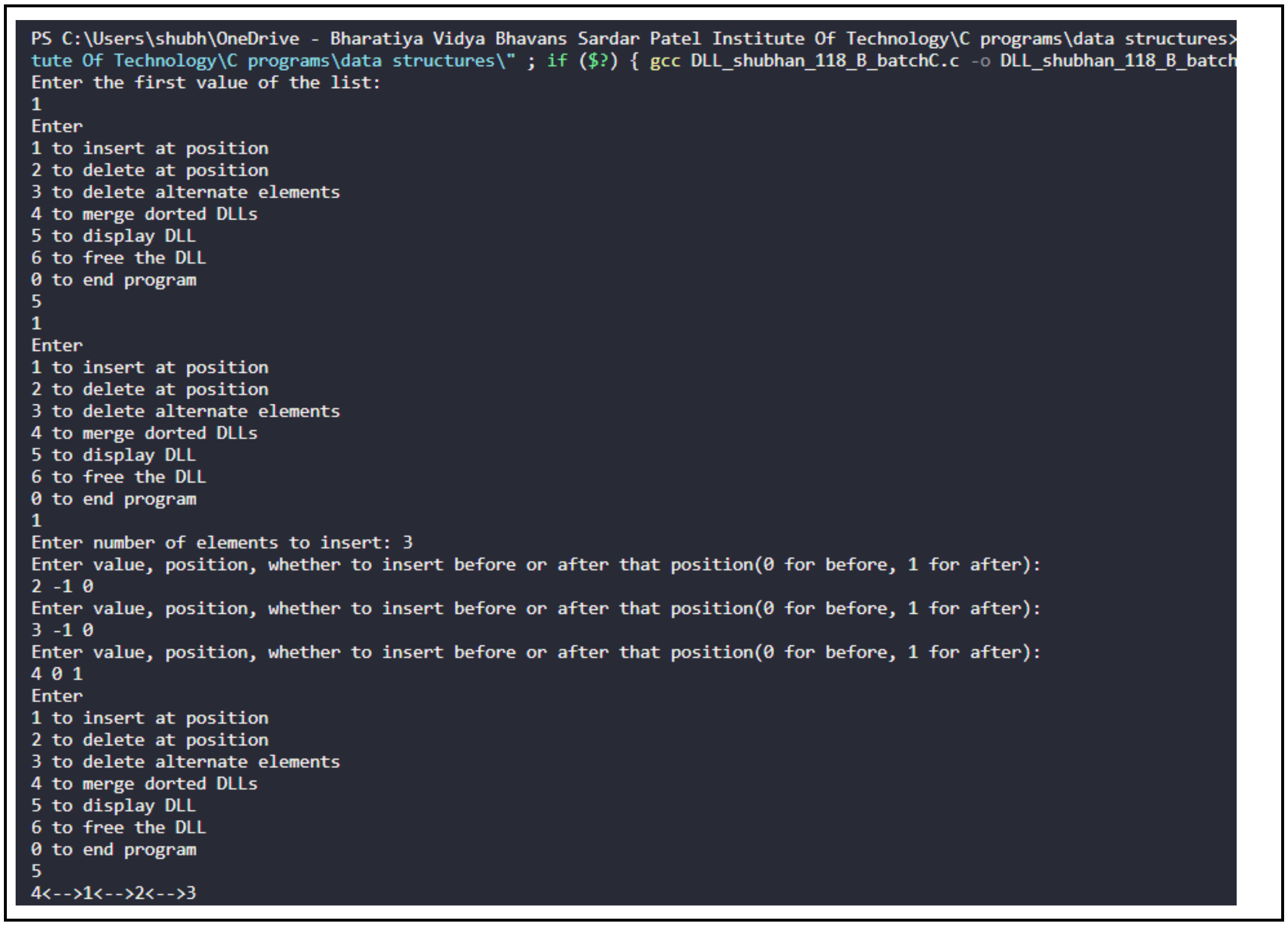
2. **insert_at_pos(Node* header, Node* trailer, int val, int pos, bool is_after)**:
    - Description: Inserts a new node with the given value at the specified position in the doubly linked list.
    - Algorithm:
        1. Initialize a pointer **iter** to **header->next**.
        2. If **pos** is -1, set **iter** to **trailer->prev** and set **is_after** to true.
        3. If **pos** is 0, set **iter** to **header->next** and set **is_after** to false.
        4. Otherwise, traverse the list to find the **pos**-th node.
        5. Allocate memory for a new node **new** and set its value to **val**.
        6. If **is_after** is true, insert **new** after **iter**.
            - Set **new**'s prev to **iter**.
            - Set **new**'s next to **iter->next**.
            - Update **iter->next** to point to **new**.
            - Update **new->next**'s prev to point to **new**.
        7. If **is_after** is false, insert **new** before **iter**.
            - Set **new**'s next to **iter**.
            - Set **new**'s prev to **iter->prev**.
            - Update **iter->prev** to point to **new**.
            - Update **new->prev**'s next to point to **new**.
3. **delete_at_pos(Node* header, Node* trailer, int pos)**:
    - Description: Deletes the node at the specified position in the doubly linked list.
    - Algorithm:
        1. Initialize a pointer **iter** to **header->next**.
        2. If **pos** is -1, set **iter** to **trailer->prev**.
        3. Otherwise, traverse the list to find the **pos**-th node.
        4. Update the previous node's next pointer to skip **iter**.
        5. Update the next node's prev pointer to skip **iter**.
        6. Free the memory occupied by **iter**.
4. **delete_alternates(Node* header)**:
    - Description: Deletes every alternate node starting from the first node in the list.
    - Algorithm:

        1. Initialize a pointer **iter** to **header->next**.
        2. Initialize a variable **i** to 0.
        3. While **iter** is not NULL:
            - If **i** is even (0-based), delete **iter**:
                - Update the previous node's next pointer to skip **iter**.
                - Update the next node's prev pointer to skip **iter**.
                - Free the memory occupied by **iter**.
            - Increment **i** by 1.
            - Move **iter** to the next node.
        4. End the loop when **iter** is NULL.

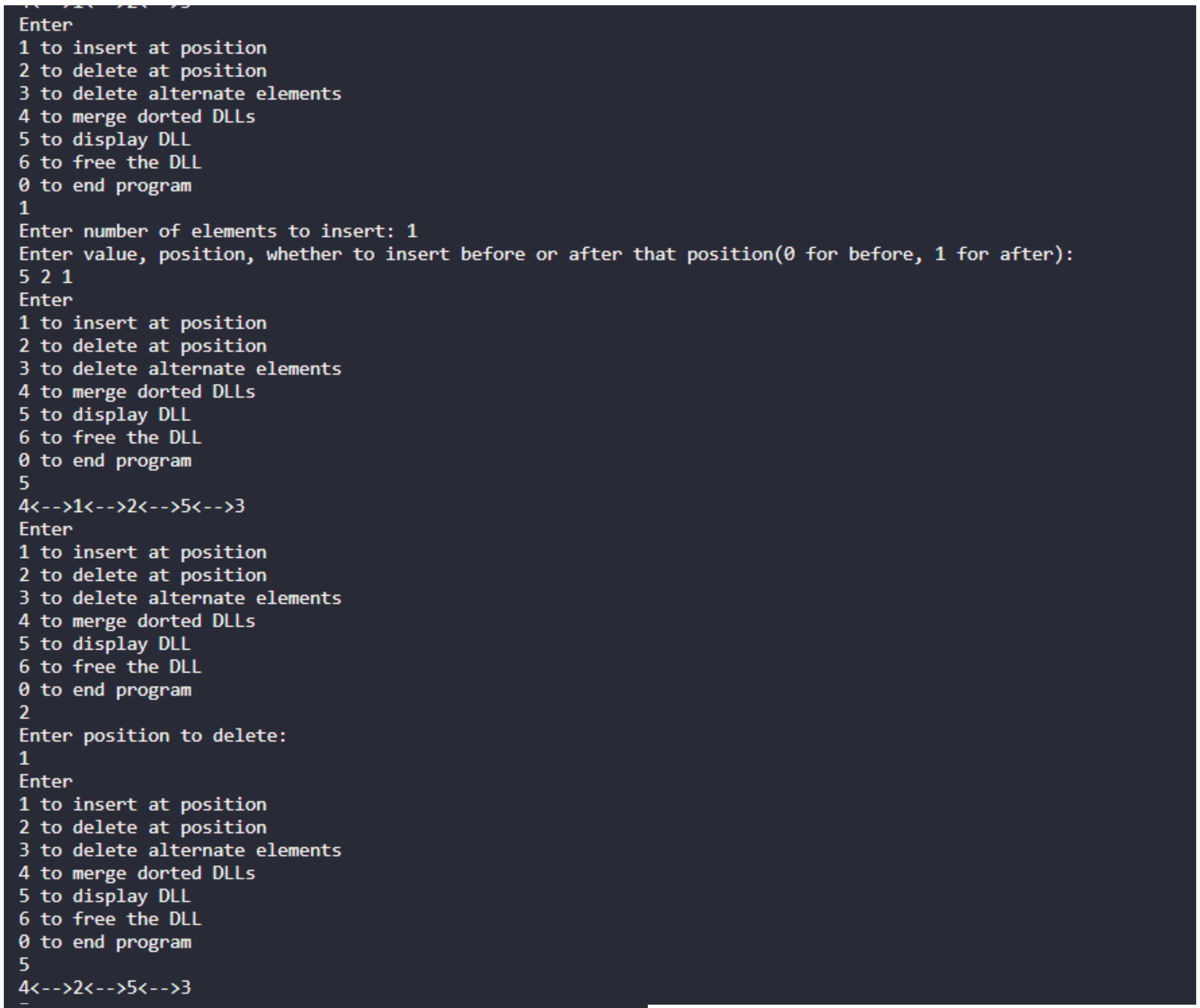
5. **merge_sorted_dlls(Node* header1, Node* header2)**:
    - Description: Merges two sorted doubly linked lists into one while preserving the order.
    - Algorithm:
        1. Initialize pointers **liter** and **riter** to the first nodes of **header1** and **header2**.
        2. Initialize a temporary pointer **temp**.
        3. While **riter** is not the trailer node of **header2**:
            - If the value at **riter** is greater than or equal to the value at **liter**:
                - Set **temp** to **riter->next**.
                - Update **riter**'s next and prev pointers to insert it after **liter**.
                - Update **liter**'s next and prev pointers accordingly.
                - Move **riter** to **temp**.
            - Otherwise, move **liter** to the next node in **header1**.
        4. If **liter** is not the trailer node of **header1**, insert the remaining elements of **header1** after **riter**.
        5. Update **header1** and **header2** as needed to reflect the merged list.

These algorithms implement the basic operations on a doubly linked list

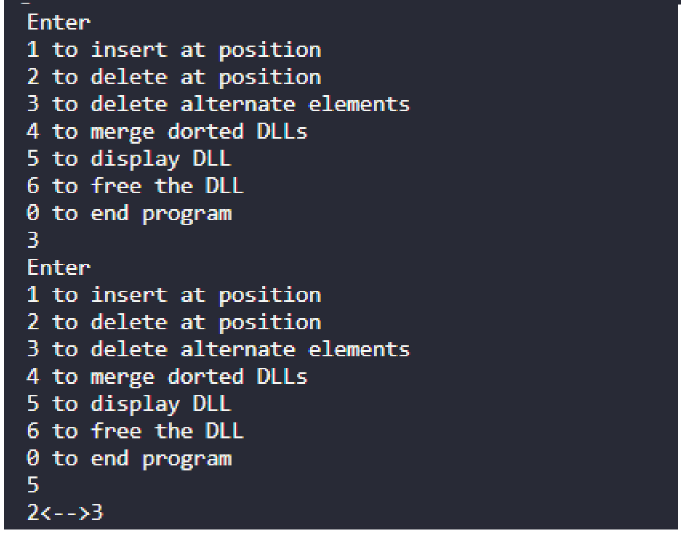
RESULT:



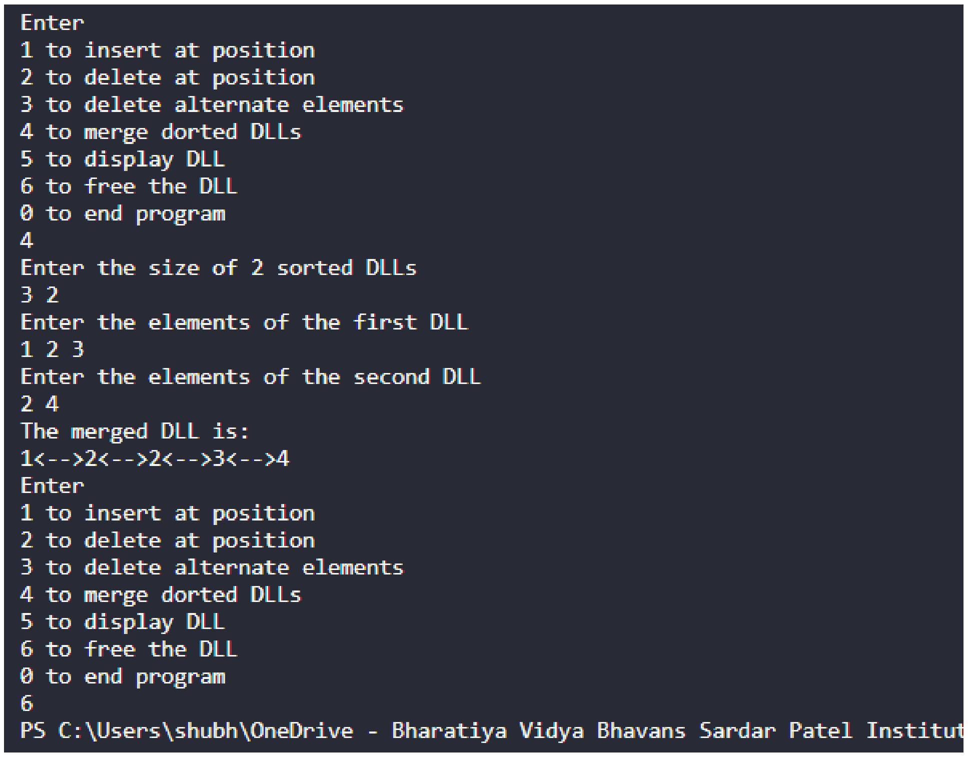
```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\C programs\data structures>
tute Of Technology\C programs\data structures\" ; if ($?) { gcc DLL_shubhan_118_B_batchC.c -o DLL_shubhan_118_B_batch
Enter the first value of the list:
1
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
5
1
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
1
Enter number of elements to insert: 3
Enter value, position, whether to insert before or after that position(0 for before, 1 for after):
2 -1 0
Enter value, position, whether to insert before or after that position(0 for before, 1 for after):
3 -1 0
Enter value, position, whether to insert before or after that position(0 for before, 1 for after):
4 0 1
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
5
4<-->1<-->2<-->3
```



```
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
1
Enter number of elements to insert: 1
Enter value, position, whether to insert before or after that position(0 for before, 1 for after):
5 2 1
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
5
4<-->1<-->2<-->5<-->3
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
2
Enter position to delete:
1
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
5
4<-->2<-->5<-->3
```



```
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
3
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
5
2<-->3
```



```
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
4
Enter the size of 2 sorted DLLs
3 2
Enter the elements of the first DLL
1 2 3
Enter the elements of the second DLL
2 4
The merged DLL is:
1<-->2<-->2<-->3<-->4
Enter
1 to insert at position
2 to delete at position
3 to delete alternate elements
4 to merge dorted DLLs
5 to display DLL
6 to free the DLL
0 to end program
6
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institut
```

Theory:	A doubly linked list is a fundamental data structure used in computer science and programming for efficiently organizing and managing collections of data elements. It is an extension of the singly linked list, where each node in the list contains not only a reference to the next node but also a reference to the previous node. This bidirectional linkage provides several advantages and flexibility, making doubly linked lists a valuable tool in various applications. Key characteristics and features of doubly linked lists: 1. **Bidirectional Traversal**: Unlike singly linked lists, where you can only traverse forward through the list, doubly linked lists allow for traversal in both directions. This makes it easier to navigate the list from the beginning to the end and vice versa. 2. **Insertion and Deletion**: Doubly linked lists support efficient insertion and deletion operations at both the beginning and end of the list, as well as at any arbitrary position within the list. These operations typically require updating only a few pointers, resulting in a constant-time or O(1) complexity for many cases. 3. **Memory Overhead**: The primary disadvantage of doubly linked lists compared to singly linked lists is the increased memory overhead due to the extra pointers for the previous nodes. Each node contains two pointers (next and prev) instead of just one (next) in singly linked lists. 4. **Reversal**: Reversing a doubly linked list is a straightforward process.

	You can simply swap the prev and next pointers for each node, effectively reversing the order of elements in the list. 5. **Applications**: • Doubly linked lists are commonly used in various data structures like stacks, queues, and dequeues (double-ended queues). • They are used in some memory management algorithms, such as memory allocation and deallocation. • Doubly linked lists are used in text editors and applications that require efficient cursor movement and text manipulation. 6. **Complexity**: • Accessing elements by index in a doubly linked list generally requires linear time O(n) because you may need to traverse the list from either end. • Insertion and deletion at arbitrary positions may require O(n) time in the worst case. In summary, doubly linked lists offer bidirectional traversal and efficient insertion/deletion at both ends of the list, making them a versatile choice for various data structures and applications. However, they come with increased memory overhead compared to singly linked lists, and accessing elements by index can be less efficient due to the need for traversal. The choice of which linked list type to use (singly or doubly) depends on the specific requirements of your application and the trade-offs between memory usage and the types of operations you need to perform.
Conclusion:	**1)**Bidirectional traversal: Unlike singly linked lists, which allow you to traverse the list only in one direction (forward), doubly linked lists allow for traversal in both directions (forward and backward). **2)**Efficient insertion and deletion: Adding or removing elements at the beginning or end of a doubly linked list is efficient because you have direct access to both the previous and next nodes.