Lab Report 05

Submitted By

Muhammad Awais / 2883

Submitted To

Ma’am Zanaib Malik

Department of Computer Science and Engineering

National University of modern language, Islamabad

List of Task

1. Remove Given Element from the list (Singly linked list)
2. Start Double linked list
3. Made a class DNode
4. Build parent child relation to Node(Parent) with DNode(Child).
5. Add to Head
6. Add to tail
7. Forward Traverse
8. Reverse Traverse (Task)
9. Add Before Given Element
10. Remove from Tail
11. Searching (Task)

1. Description of Void Remove Given Element

```
 /*write a function name remove given element that will take an element as a
parameter. The function should find the given element and delete it if element is
present otherwise display an error. */
   void removeGivenElement(t element);
```

1. Void function means **nothing will return.**
2. Take an element as a parameter.
3. Check 5-possible mappings.
4. If list is empty show an error message.
5. `Only Head modify -> more than one element and element on head`

   In this condition we **call remove from Head function** which remove element from head.
6. `Only tail modify -> more than one element and element on tail`

   In this condition we **call remove from Tail function** which remove element from tail.
7. `Head and Tail both modify -> only one element in list that is the given element`

   We just delete head or tail and assign them zero so list will empty
8. `Nor Head Not Tail Modify-> Yes if remove from in between`

In this condition declared a node variable ptr start from head and use a while loop having condition ptr!= tail and element != ptr next address and information / data of ptr next.

If loop terminate by ptr!=tail means element is not found in the list.

We stop the loop before the given element. Why? To save the information and address to maintain the connection in list.

Otherwise element is found So perform the deletion step.

- Make a node variable and store the node the next node of given element. Why we do this? Because after deleting the given node we lost the further connection So maintain the connection we store the node of next of element in a temporary node.
- Then delete the given element.
- Set next of ptr to temporary node which make a connection in list.

Source Code

```
Main Class
#include <iostream>
#include "SLL.h"
using namespace std;

int main(int argc, char **argv)
{
    SLL<string> list1;
    list1.addToHead("Awais");
    list1.addToHead("Qarni");
    list1.addToTail("C++");
    list1.addToTail("Code");
    list1.addAfterGivenElement("C#", "Qarni");
    list1.addAfterGivenElement("C","C++");
    list1.addBeforeGivenElement("Awais", " Haider");
    list1.addBeforeGivenElement("C++", "python");
    list1.traverse();

    list1.removeGivenElement("C++");
    list1.removeGivenElement("Awais");
    list1.removeGivenElement("C#");
    list1.removeGivenElement("DSA");
    list1.removeGivenElement("Qarni");
    list1.removeGivenElement("Code");
    list1.traverse();
```

```
 return 0;
}

template <class t>
void SLL<t>::removeGivenElement(t element)
{
    /* Check 5 Possible Mapping
 1- Error -> if empty && element not found in list
 2- Only Head modify -> more than one element and element on head
 3- Only Tail modify -> more than one element and element on tail
 4- Head and Tail both modify -> only one element in list that is the given
element

5- Nor Head Not Tail Modify-> Yes if remove from in between
 */

    if (head == 0 && tail == 0)
    {
        cerr << "List is empty \n";
    }
    else if (element == head->getData()) // more than one element and element  on
head
    {
        removeFromHead();
    }
    else if (element == tail->getData()) // more than one element and element on
tail
    {
        removeFromTail();
    }
    else if (head == tail) // only one element in list which is given element
    {
        delete head;
        head = tail = 0;
    }
    else // Not head Nor tail
    {
        Node<t> *ptr = head;

        while (ptr != tail && element != ptr->getNext()->getData())
        {
            ptr = ptr->getNext();
        }

        if (ptr == tail) // 1st condition
```

```
        {
            cerr << "Element is not Found in list \n";
        }
        else // 2nd condition
        {
            Node<t> *temp = ptr->getNext()->getNext();
            delete ptr->getNext();
            ptr->setNext(temp);
        }
    }

} // End Remove From Given Element
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\BSCS 3rd Semester\DSA\My_code\Singly linked list> cd "e:\BSCS 3rd Semester\DSA\My_code\Singly linked list\" ; if ($?) {
\main }
(0x911378) |Qarni|0x910f10|
(0x910f10) |C#|0x910f60|
(0x910f60) | Haider|0x911350|
(0x911350) |Awais|0x910f88|
(0x910f88) |python|0x9113e8|
(0x9113e8) |C++|0x910f38|
(0x910f38) |C|0x911410|
(0x911410) |Code|0|
Element is not Found in list
(0x910f60) | Haider|0x910f88|
(0x910f88) |python|0x910f38|
(0x910f38) |C|0|
PS E:\BSCS 3rd Semester\DSA\My_code\Singly linked list>
```

2. Description of DNode Class

This code defines a template class DNode that represents a doubly linked list node, where each node has a pointer to its previous and next nodes. The class is inherited from the Node class, which represents a single linked list node. The DNode class has a private member variable prev that points to the previous node in the list. The constructor of the class takes three arguments: the previous node pointer, the node information, and the next node pointer. The constructor also calls the parent class constructor by passing the node information and next node pointer as arguments to it. The class has three member functions. setprev() sets the prev pointer to the given pointer. getprev() returns the prev pointer. displayDNode() displays the node information and the previous node pointer.

The code also includes a header file Node.h, which contains the definition of the parent class Node.

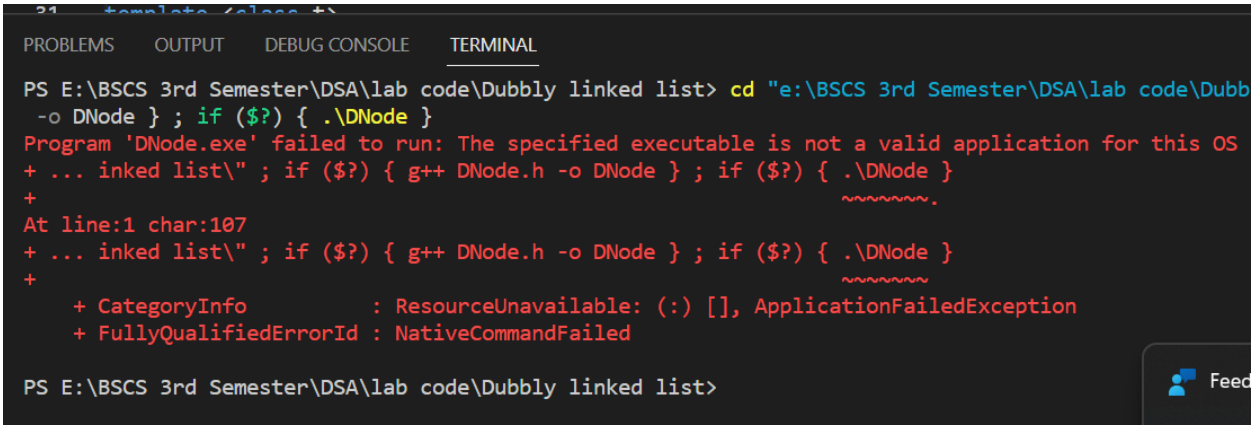
Overall, the code provides a basic implementation of a doubly linked list node using templates and inheritance.

Source Code

```
#include<iostream>
#include"E:\BSCS 3rd Semester\DSA\lab code\L3P3 Singular link List\Node.h"
using namespace std;
template<class t>
class DNode:public Node<t>   // Extend  parent class
{
private:
    DNode<t> *prev;

public:
    DNode(DNode<t> *prev, t info, DNode<t> *next):Node<t>(info,next) // inhe
parent funcnalities
    {
        this->prev = prev;
    }

    void setprev(DNode<t> *prev);

    DNode<t> *getprev();
    void displayDNode();

};// End of DNode class

// ************************Function Definitions**********************

template <class t>
void DNode<t>::setprev(DNode<t> *prev)
{
    this->prev = prev;
} // end

template <class t>
DNode<t> *DNode<t>::getprev()
{
    return this->prev;
} // end
```

```
template<class t>
void DNode<t>::displayDNode()
{
    Node<t>::displayNode();
    cout<<prev<<")|";
}// End of Display
```

Output



3. Description of Void Add to Head

function that adds a new node with a given element to the head of a doubly linked list. The function takes a template parameter 't' to indicate the data type of the element being added. Here's a description of what the function does:

1. The function first creates a new node 'n' with the given element.
2. If the head and tail pointers of the linked list are both null, then the list is empty and the new node becomes both the head and the tail of the list.
3. If the list is not empty, then the new node is added to the head of the list. This involves the following steps:
   a) Set the next pointer of the new node 'n' to point to the current head of the list.
   b) Set the previous pointer of the current head of the list to point to the new node 'n'.
   c) Update the head pointer of the list to point to the new node 'n'.
4. The function does not modify the tail pointer since the new node is added to the head of the list.
5. If the list is already empty, then neither the head nor the tail pointers are modified.
6. There is no typecasting required since the template parameter 't' already indicates the data type of the element being added.

Source Code

```
#include <iostream>
```

```cpp
#include "DLL.h"
using namespace std;

int main(int argc, char **argv)
{
    DLL<int> l1;
    l1.addToHead(1);
    l1.addToHead(2);
    // l1.addToTail(4);
    // l1.addToTail(9); // Part of Assignment
    // l1.addToHead(0);
    // l1.addBeforeGivenE(7, 9);
    // l1.addAfterGivenE(1, 10); // Part Of Assignment
    // l1.removeFromHead(); // Part Of Assignment
    // l1.removeFromTail();
   // l1.removeGivenElement(7); // Part of Assignment
    l1.forwardTraverse();
    cout << endl;
    //cout<<l1.searchElement(11);  // Part of Assignment
    // l1.reverseTraverse(); // Part Of Assignment

    return 0;
}
template <class t>
void DLL<t>::addToHead(t element)
{
    /*5 possible scenarios
   1-> Error -> No
   2-> Modify head only -> if list one or more having element
   3-> Modify tail only -> No
   4-> Modify head and tail-> if list is empty
   5-> Neither head nor tail modify -> No
    */

    DNode<t> *n = new DNode<t>(0, element, 0);
    if (head == 0 && tail == 0)
    {
        head = tail = n;
    }

    else
    {
        n->setNext(head); //  why not typecasting
        head->setprev(n);
        head = n;
```

```
    }

} // end of Add To Head
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\BSCS 3rd Semester\DSA\lab code\Dubbly linked list> cd "e:\BSCS 3rd Semester\DSA\la
p -o main } ; if ($?) { .\main }
(0x1101488) |2|0x1101470|0)|
(0x1101470) |1|0|0x1101488)|

PS E:\BSCS 3rd Semester\DSA\lab code\Dubbly linked list>
```

4. Description of Add to Tail

class template DLL (Doubly Linked List) that adds a new element to the tail (end) of the list.

The function starts by creating a new node using the template class t. The node is initialized with a value of 'element' and default values of 0 for the previous and next pointers.

The function then checks if the list is empty by checking if both the head and tail pointers are null. If the list is empty, the head and tail pointers are set to the new node. This indicates that the list now contains a single node with the new element as its value.

If the list is not empty, the new node is added to the end of the list by setting the next pointer of the current tail node to point to the new node and setting the previous pointer of the new node to point to the current tail node. Then, the tail pointer is updated to point to the new node, indicating that the new node is now the last element of the list.

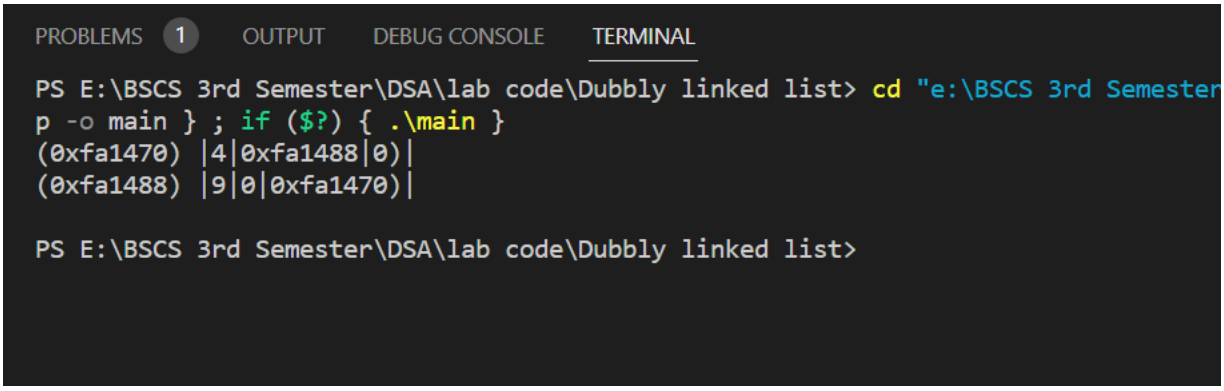
Overall, this function adds a new node containing the specified element to the tail of the doubly linked list and updates the head and tail pointers as necessary, based on whether the list is empty or not.

Source Code

```
#include <iostream>
#include "DLL.h"
using namespace std;
```

```
int main(int argc, char **argv)
{
    DLL<int> l1;

     l1.addToTail(4);
    l1.addToTail(9);
    l1.forwardTraverse();
    cout << endl;

    return 0;
}


template <class t>
void DLL<t>::addToTail(t element)
{
    /*5 possible scenarios
1-> Error -> No
2-> Modify head only -> No
3-> Modify tail only -> if list is having one or more element
4-> Modify head and tail-> if list is empty
5-> Neither head nor tail modify -> No
 */

    DNode<t> *n = new DNode<t>(0, element, 0);
    if (head == 0 && tail == 0)
    {
        head = tail = n;
    }

    else
    {
        tail->setNext(n);
        n->setprev(tail);
        tail = n;
    }

} // End of Add To Tail
```

Output



5. Description of Forward Traverse

- This code represents a member function **forwardTraverse()** defined in a templated doubly linked list class (**DLL**) that can hold elements of any type **t**.
- The function performs a traversal of the linked list in a forward direction, starting from the head node. It does this by initializing a pointer **ptr** to the head node of the list, and then iterating through each node in the list using a while loop.
- During each iteration, the function displays the contents of the current node by calling the **displayDNode()** function of the **DNode** class. After displaying the node's contents, it moves the pointer to the next node in the list by explicitly typecasting the **getNext()** function of the **DNode** class to a pointer of type **DNode<t>***.
- This typecasting is necessary because the **getNext()** function of the **DNode** class returns a **Node*** type, which is a base class pointer. Since the **DLL** class is templated, the compiler needs to know the exact type of the pointer to set it to the next node in the list. Therefore, the explicit typecasting ensures that the pointer is correctly set to the next node in the list.
- Overall, the **forwardTraverse()** function allows the user to traverse a doubly linked list in a forward direction and display the contents of each node.

Source Code

```
template <class t>
void DLL<t>::forwardTraverse()
{
    DNode<t> *ptr = head;
    while (ptr != NULL)
    {
```

```
        ptr->displayDNode();
        cout << endl;
        ptr = (DNode<t> *)ptr->getNext(); // typcasting explicit
    }
} // End of Forward Traverse
```

6. Description of Forward Traverse

- Function **reverseTraverse()** which is a member function of a doubly linked list class template **DLL** with a template parameter **t**. This function traverses the list in reverse order, starting from the tail node and printing each node's data by calling its **displayDNode()** member function.
- The function starts by initializing a pointer **ptr** to the tail node of the list. Then it enters a loop that continues until **ptr** becomes NULL, which indicates that it has reached the beginning of the list.
- In each iteration of the loop, the function first calls the **displayDNode()** member function of the current node pointed by **ptr** to print its data. Then it prints a newline character for formatting purposes.
- After that, the function updates the **ptr** pointer to point to the previous node of the current one by calling its **getprev()** member function. This is how the function traverses the list in reverse order.
- Finally, when the loop exits, the function returns and the reverse traversal of the list is complete.
- Overall, the **reverseTraverse()** function provides a useful way to iterate over a doubly linked list in reverse order, allowing for efficient access to the list's elements from back to front.

Source Code

```
#include <iostream>
#include "DLL.h"
using namespace std;

int main(int argc, char **argv)
{
  DLL<int> l1;
  // l1.addToHead(1);
  // l1.addToHead(2);
  l1.addToTail(4);
  l1.addToTail(9);
  l1.addToTail(19);
  l1.addToHead(0);
```

```
  l1.addAfterGivenE(4, 33);
  l1.addBeforeGivenE(7, 9);

  l1.forwardTraverse();
  cout << endl;
  cout<<"Below is reverse Traverse \n";
  l1.reverseTraverse();

return 0;
}
template <class t>
void DLL<t>::reverseTraverse()
{
    DNode<t> *ptr = tail;
    while (ptr != NULL)
    {
        ptr->displayDNode();
        cout << endl;
        ptr = ptr->getprev();
    }

} // End of Reverse traverse
```

Output

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

p -o main } ; if ($?) { .\main }
(0x8913f0) |0|0x891470|0)|
(0x891470) |4|0x891408|0x8913f0)|
(0x891408) |33|0x890f98|0x891470)|
(0x890f98) |7|0x891488|0x891408)|
(0x891488) |9|0x8913d8|0x890f98)|
(0x8913d8) |19|0|0x891488)|

Below is reverse Traverse
(0x8913d8) |19|0|0x891488)|
(0x891488) |9|0x8913d8|0x890f98)|
(0x890f98) |7|0x891488|0x891408)|
(0x891408) |33|0x890f98|0x891470)|
(0x891470) |4|0x891408|0x8913f0)|
(0x8913f0) |0|0x891470|0)|
PS E:\BSCS 3rd Semester\DSA\lab code\Dubbly linked list>
```

7. Description of Add Before Given Element

- The function **addBeforeGivenE** takes two arguments, a new element **newE** and an existing element **existing** to which the new element is to be added before.
- The function first checks whether the list is empty or not. If the list is empty, it prints an error message indicating that the list is empty.
- If the list is not empty, the function checks whether the existing element is the head node of the list. If it is, the function adds the new element before the head node using the **addToHead** function, which is not shown in the given code.
- If the existing element is not the head node, the function iterates through the list to find the node that contains the existing element. The iteration is done using a while loop that starts from the head node and stops when it reaches the tail node or finds the node containing the existing element.
- During the iteration, the pointer **ptr** is of type **DNode<t>**, which is a pointer to a node of the doubly linked list class template. When **ptr** moves to the next node using the **getNext()** function, the pointer has to be typecasted to **DNode<t>*** to explicitly convert it to a pointer of the correct type, as the **getNext()** function returns a pointer to the base class **Node**.
- Once the node containing the existing element is found, the function creates a new node **n** with the new element **newE**, and inserts it before the node containing the existing element. The **setNext()** and **setprev()** functions are used to set the next and previous pointers of the new node **n**, as well as the pointers of the adjacent nodes.
- If the existing element is not found in the list, the function prints an error message indicating that the existing element is not present in the list.
- In summary, the **addBeforeGivenE** function adds a new element before a given existing element in a doubly linked list.

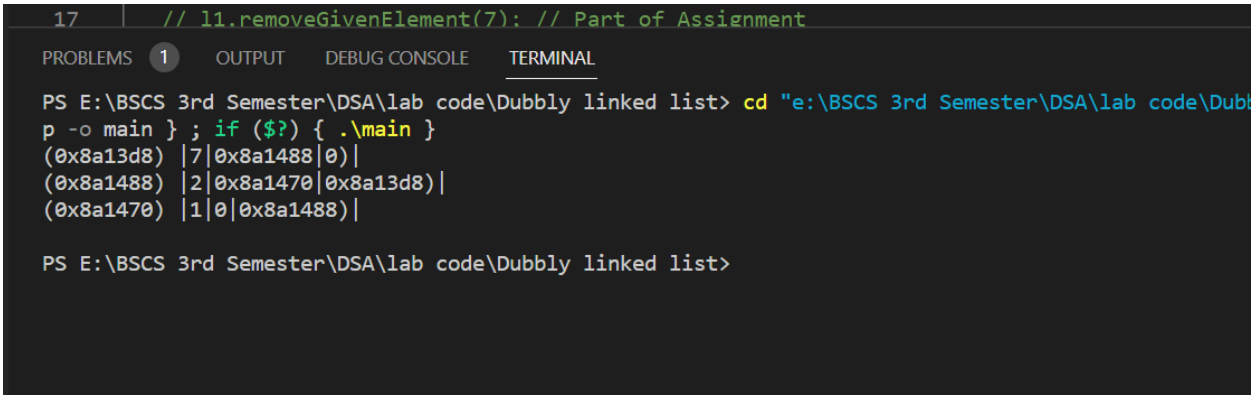
Source Code

```
#include <iostream>
#include "DLL.h"
using namespace std;

int main(int argc, char **argv)
{
    DLL<int> l1;
    l1.addToHead(1);
    l1.addToHead(2);
    l1.addBeforeGivenE(7, 2);
    l1.forwardTraverse();
    cout << endl;
```

```
    return 0;
}


template <class t>
void DLL<t>::addBeforeGivenE(t newE, t existingE)
{
    /*5 possible scenarios
1-> Error -> yes if empty
2-> Modify head only -> if element is on head
3-> Modify tail only -> No
4-> Modify head and tail-> No
5-> Neither head nor tail modify -> if element is in between of list
*/

    if (head == 0 && tail == 0)
    {
        cerr << "List is empty \n";
    }

    else if (existingE == head->getInfo())
    {
        addToHead(newE);
    }

    else
    {
        DNode<t> *ptr = head;
        while (ptr != tail && existingE != ptr->getNext()->getInfo()) // why not
typcasting
        {
            ptr = (DNode<t> *)(ptr->getNext()); // why here is typcasting
        }

        if (ptr == tail) // 1st condition
        {
            cerr << "Existing element Not found \n";
        }

        else // Found
        {
            DNode<t> *n = new DNode<t>(0, newE, 0);
            n->setNext(ptr->getNext());
            n->setprev(ptr);
            ptr->setNext(n);
```

```
            ((DNode<t> *)n->getNext())->setprev(n);
        }
    }
} // End of Add Before Given Element
```

Output



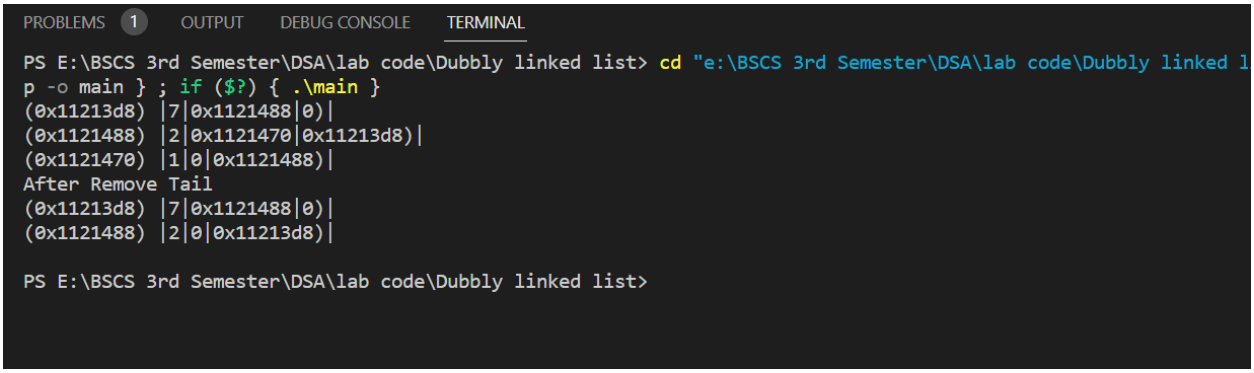
8. Description of Remove Tail

- The "Remove from Tail" function is a member function of a template doubly linked list class (DLL). This function is responsible for deleting the node at the tail end of the linked list.
- The function first checks if the linked list is empty by checking if both the head and tail pointers are pointing to null. If the linked list is empty, the function displays an error message indicating that the list is empty and there is nothing to delete.
- If the linked list is not empty, the function checks if there is only one node in the linked list by comparing the head and tail pointers. If there is only one node, then the function deletes that node and sets both the head and tail pointers to null, indicating that the linked list is now empty.
- If the linked list has more than one node, the function creates a temporary pointer named "temp" and sets it to point to the node before the tail node. The function then sets the next pointer of the "temp" node to null, indicating that it is now the new tail of the linked list. Finally, the function deletes the original tail node and updates the tail pointer to point to the new tail node.
- In summary, the "Remove from Tail" function deletes the node at the tail end of the linked list and updates the necessary pointers to maintain the integrity of the doubly linked list.

Source Code

```cpp
#include <iostream>
#include "DLL.h"
using namespace std;

int main(int argc, char **argv)
{
    DLL<int> l1;
    l1.addToHead(1);
    l1.addToHead(2);

    l1.addBeforeGivenE(7, 2);
      l1.forwardTraverse();
      cout<<"After Remove Tail \n";
     l1.removeFromTail();
    l1.forwardTraverse();
    cout << endl;
    return 0;
}


template <class t>
void DLL<t>::removeFromTail()
{
    /*5 possible scenarios
1-> Error -> yes if empty
2-> List have node (may be one or more)->Delete Tail
 */
    if (head == 0 && tail == 0)
    {
        cerr << "List is empty So, nothing will delete \n";
    }
    else if (head == tail)
    {
        delete tail;
        head = tail = 0;
    }
    else
    {
        DNode<t> *temp = tail->getprev();
        temp->setNext(0);
        delete tail;
        tail = temp;
    }

} // End of Remove from Tail
```

Output



9. Description of Searching Function

- Implementation a **search function** for a doubly linked list in C++.
- The function takes a **parameter 'element' of type 't**', which is the type of data stored in the nodes of the list. It returns a **boolean value, true if the 'element' is found in the list and false otherwise.**
- The function initializes a pointer **'ptr' to** point to the head of the list. It then traverses the list until it reaches the tail or until it finds a node whose 'info' field matches the 'element'. The 'info' field of a node contains the data stored in that node.
- If the **'element' is not found in the list**, i.e., the traversal reaches the end of the list, the function **returns false. Otherwise, it returns true,** indicating that the 'element' is present in the list.
- Note that the traversal is performed using the **'getNext()'** function of the **DNode class, which returns a pointer to the next node in the list.** Since this is a doubly linked list, each node also has a **'getPrev()'** function, which returns a pointer to the previous node in the list.
- Overall, this function provides a basic implementation of a search function for a doubly linked list in C++.

Source Code

```
#include <iostream>
#include "DLL.h"
using namespace std;

int main(int argc, char **argv)
{
  DLL<int> l1;
  // l1.addToHead(1);
  // l1.addToHead(2);
  l1.addToTail(4);
  l1.addToTail(9);
  l1.addToTail(19);
  l1.addToHead(0);
  l1.addAfterGivenE(4, 33);
  l1.addBeforeGivenE(7, 9);

  l1.forwardTraverse();


   cout<<l1.searchElement(19);
   cout<<endl;
   cout<<l1.searchElement(99);

  return 0;
}


template<class t>
bool DLL<t>::searchElement(t element)
{
    DNode<t> *ptr=head;
    while (ptr!=tail && element!=ptr->getNext()->getInfo())
    {
        ptr=(DNode<t>*)ptr->getNext();
    }

    if (ptr==tail)
    {
        return false;
    }

   else
   {
```

```
53.         return true;
54.     }
55.
56.}// End of Search Element Class
```

Output

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS E:\BSCS 3rd Semester\DSA\lab code\Dubbly linked list> cd "e:\BSCS 3rd Semester\DSA\la
p -o main } ; if ($?) { .\main }
(0x10213f0) |0|0x1021470|0)|
(0x1021470) |4|0x1021408|0x10213f0)|
(0x1021408) |33|0x1020f98|0x1021470)|
(0x1020f98) |7|0x1021488|0x1021408)|
(0x1021488) |9|0x10213d8|0x1020f98)|
(0x10213d8) |19|0|0x1021488)|
1
0
PS E:\BSCS 3rd Semester\DSA\lab code\Dubbly linked list>
```