University of Anbar
College of Computer Science
and Information Technology
Computer Network Systems
Department

Data Structures

Lecture Twelve

Second Stage

First Course - 2024-2025

Muhmmad shihab muayad Shihab

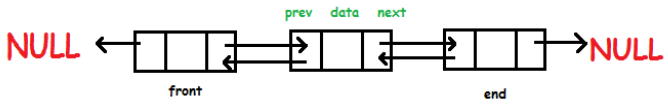
Master of Computer Engineering

muhmmad.shihab@uoanbar.edu.iq

Data Structures

Double Linked List

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL.



The two links help us to traverse the list in both backward and forward direction. But storing an extra link requires some extra space.

Implementation of Doubly Linked List

First we define the node.

```
struct node
{
        int data;       // Data
        node *prev;          // A reference to the previous node
        node *next;  // A reference to the next node
};
```

Now we define our class Doubly Linked List. It has the following methods:

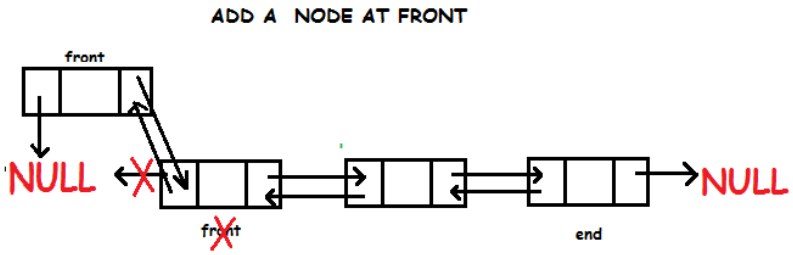
- add_front: Adds a new node in the beginning of list
- add_after: Adds a new node after another node
- add_before: Adds a new node before another node
- add_end: Adds a new node in the end of list
- delete: Removes the node
- forward_traverse: Traverse the list in forward direction
- backward_traverse: Traverse the list in backward direction

```
class Doubly_Linked_List
{
        node *front;          // points to first node of list
        node *end;            // points to first las of list
        public:
```

```
        Doubly_Linked_List()
        {
                front = NULL;
                end = NULL;
        }
        void add_front(int );
        void add_after(node* , int );
        void add_before(node* , int );
        void add_end(int );
        void delete_node(node*);
        void forward_traverse();
        void backward_traverse();
};
```

Insert Data in the beginning

1. The prev pointer of first node will always be NULL and next will point to front.
2. If the node is inserted is the first node of the list then we make front and end point to this node.
3. Else we only make front point to this node.



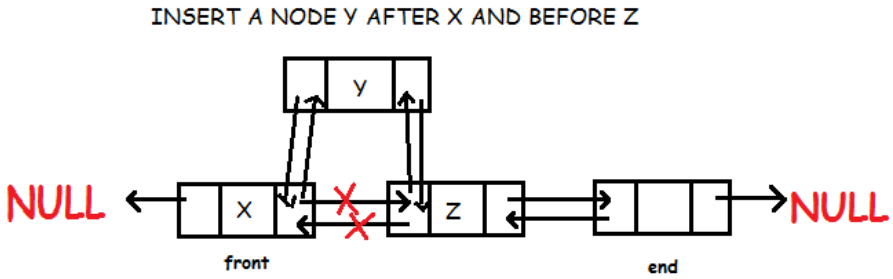
1. WE MAKE THE CURRENT FRONT NODE'S PREV POINT TO NEW NODE.
2. MAKE NEXT OF NEW NODE POINT TO CURRENT FRONT AND PREV OF NEW NODE POINT TO NULL.
3. WE CHANGE FRONT NODE TO THE NEW NODE.

```
void Doubly_Linked_List :: add_front(int d)
{
        // Creating new node
        node *temp;
        temp = new node();
        temp->data = d;
        temp->prev = NULL;
```

```
        temp->next = front;

        // List is empty
        if(front == NULL)
                end = temp;

        else
                front->prev = temp;

        front = temp;
}
```

Insert Data before a Node

Let's say we are inserting node X before Y. Then X's next pointer will point to Y and X's prev pointer will point the node Y's prev pointer is pointing. And Y's prev pointer will now point to X. We need to make sure that if Y is the first node of list then after adding X we make front point to X.



1. WE MAKE Y'S NEXT NODE POINT TO Z AND PREV NODE POINT TO X.
2. THEN MAKE X'S NEXT NODE POINT TO Y AND Z'S PREV NODE POINT TO Y.

```
void Doubly_Linked_List :: add_before(node *n, int d)
{
        node *temp;
        temp = new node();
        temp->data = d;
        temp->next = n;
        temp->prev = n->prev;
        n->prev = temp;

        //if node is to be inserted before first node
```

```
        if(n->prev == NULL)
                front = temp;
}
```

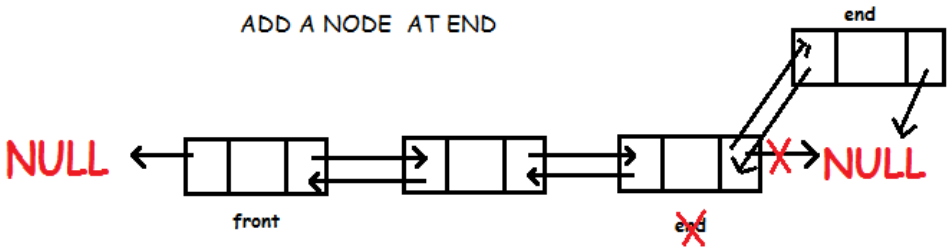
Insert Data after a Node

Let's say we are inserting node Y after X. Then Y's prev pointer will point to X and Y's next pointer will point the node X's next pointer is pointing. And X's next pointer will now point to Y. We need to make sure that if X is the last node of list then after adding Y we make end point to Y.

```
void Doubly_Linked_List :: add_after(node *n, int d)
{
        node *temp;
        temp = new node();
        temp->data = d;
        temp->prev = n;
        temp->next = n->next;
        n->next = temp;

        //if node is to be inserted after last node
        if(n->next == NULL)
                end = temp;
}
```

Insert Data in the end

1. The next pointer of last node will always be NULL and prev will point to end.
2. If the node is inserted is the first node of the list then we make front and end point to this node.
3. Else we only make end point to this node.

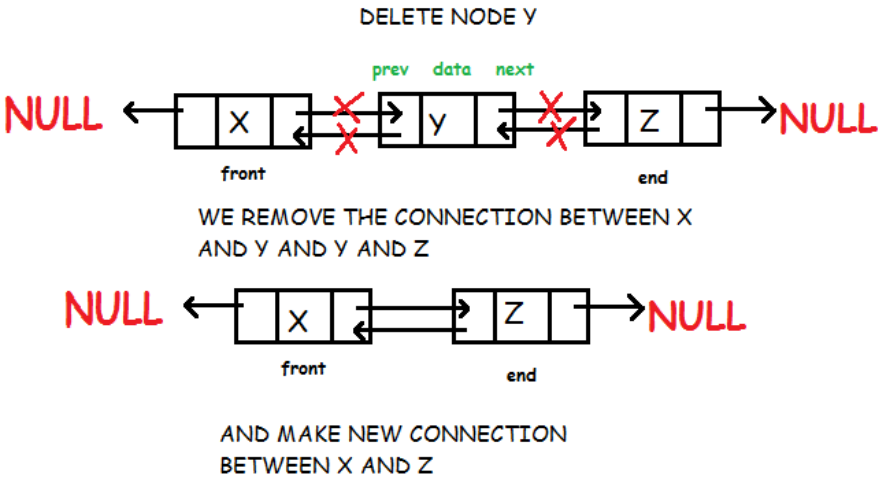


1. WE MAKE THE CURRENT END NODE'S NEXT POINT TO THE NEW NODE
2. THEN WE MAKE NEW NODE'S PREV POINT TO CURRENT END NODE AND NEXT POINT TO NULL.
3. LASTLY WE CHANGE END TO NEW NODE

```
void Doubly_Linked_List :: add_end(int d)
{
        // create new node
        node *temp;
        temp = new node();
        temp->data = d;
        temp->prev = end;
        temp->next = NULL;

        // if list is empty
        if(end == NULL)
                front = temp;
        else
                end->next = temp;
        end = temp;
}
```

Remove a Node

Removal of a node is quite easy in Doubly linked list but requires special handling if the node to be deleted is first or last element of the list. Unlike singly linked list where we require the previous node, here only the node to be deleted is needed. We simply make the next of the previous node point to next of current node (node to be deleted) and prev of next node point to prev of current node. Look code for more details.




```
void Doubly_Linked_List :: delete_node(node *n)
{
        // if node to be deleted is first node of list
        if(n->prev == NULL)
        {
                front = n->next; //the next node will be front of list
                front->prev = NULL;
        }
        // if node to be deleted is last node of list
        else if(n->next == NULL)
        {
                end = n->prev;   // the previous node will be last of
list
                end->next = NULL;
        }
        else
        {
                //previous node's next will point to current node's next
                n->prev->next = n->next;
                //next node's prev will point to current node's prev
                n->next->prev = n->prev;
        }
        //delete node
        delete(n);
}
```

Forward Traversal

Start with the front node and visit all the nodes untill the node becomes NULL.

```
void Doubly_Linked_List :: forward_traverse()
{
        node *trav;
        trav = front;
        while(trav != NULL)
        {
                cout<<trav->data<<endl;
                trav = trav->next;
        }
}
```

Backward Traversal

Start with the end node and visit all the nodes until the node becomes NULL.

```
void Doubly_Linked_List :: backward_traverse()
{
        node *trav;
        trav = end;
        while(trav != NULL)
        {
                cout<<trav->data<<endl;
                trav = trav->prev;
        }
}
```

References:

- Frank Carrano, D.J. Henry: Data Abstraction and Solving with C++, 2012, 6th edition, Pearson Education, Inc.
- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++, 2014, 4th edition, Pearson Education, Inc.