Data Structures

BS (CS) _Fall_2024

Lab_07 Manual

Learning Objectives:

1. Doubly Linked List
2. Circular Linked List

Doubly Linked List

Linked List:

A Linked List is a data structure which consists of connected nodes. Each node is connected or linked to the next node, which is why it is called a linked list. Linked Lists can be of two types: Singly Linked List or Doubly Linked List.

As we previously discuss about the single linked list now taking look into a doubly linked list. Conversely, a doubly linked list consists of nodes which have data, a pointer to the next node, but also a pointer to the previous node. Hence, we can **travel** in two directions in a doubly linked list, both **forward and backward**.

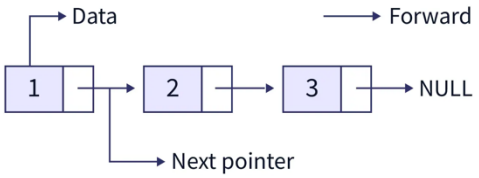


Figure 1: Single Linked List

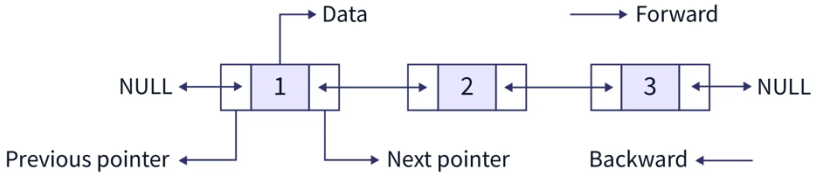


Figure 2: Doubly Linked List

Representation of Doubly Linked List:

A doubly linked lists consists of **connected nodes**. Each node contains three things:

1. **Data:** Information that the linked list contains.
2. **"*"prev:** Pointer to the previous node.
3. **"*"next:** Pointer to the next node.

Hence, a typical node in a doubly linked list will look like:

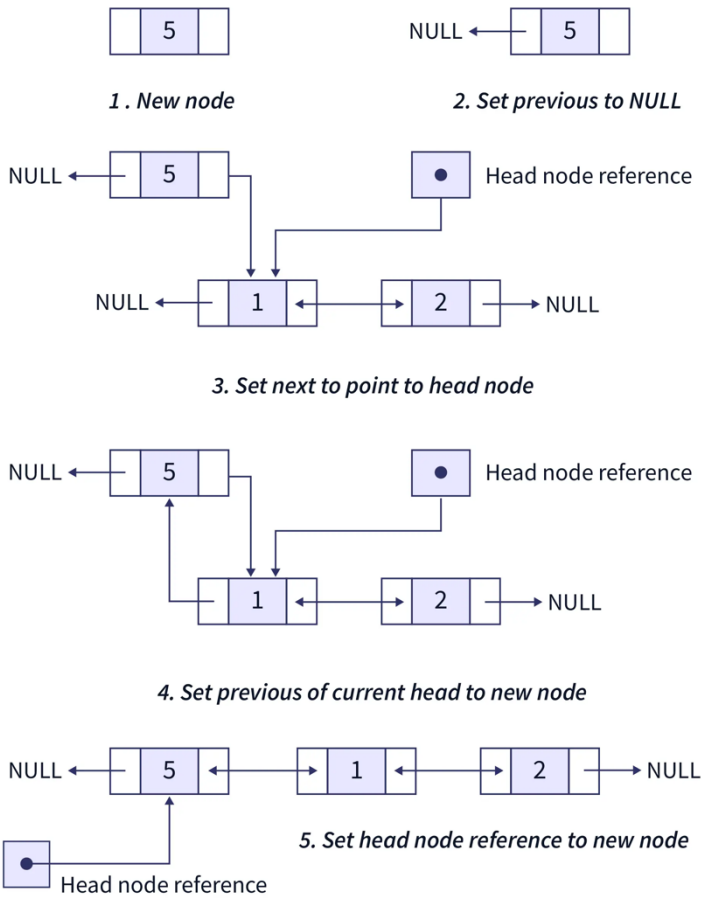
To create a doubly linked list in C++, we first need to represent the nodes. We use structures in C++ to represent one node of a **doubly linked list**, so we can create multiple such nodes and link them together using pointers to create a doubly linked list in C++.
The structure representing a node of the doubly linked list is defined as:

```
list<dataType> Name_of_list[N];
// structure to define a node of the doubly linked list in C++
struct node{

    // part which will store data
    int data;
    // pointer to the previous node
    struct node *prev;
    // pointer to the next node
    struct node *next;

};
```

Insertion in doubly linked list:

Insertion in a doubly linked list in C++ means create a new node with some data and add it to the linked list. Now, there can be **three ways** of inserting a node in a doubly linked list in C++. These are:

1. Insertion of node at the front of the list
2. Insertion of node after a given node of the list
3. Insertion of node at the end of the list.



Syntax

```
// function to insert a new node with given data in the front of the doubly linked list
void insert_at_front(node** head, int data){

    // create a new node with given data
    node* new_node = new node();
    new_node->data = data;

    // assign previous pointer to NULL
    new_node->prev = NULL;

    // assign next pointer to the current head node
    new_node->next = (*head);

    // if the list is not empty, set the current head's previous pointer to new node
    if((*head) != NULL){
        (*head)->prev = new_node;
    }

    // point the head to the new node
    (*head) = new_node;

    return;

}
```

Forward Traversal

In forward traversal, we just need to follow the next pointers of the nodes of the doubly linked list until we reach the end, which is denoted by the NULL pointer. This can be easily achieved using a while loop in C++.
The diagram below illustrates the concept.

Follow the next pointer for forward traversal

Backward Traversal

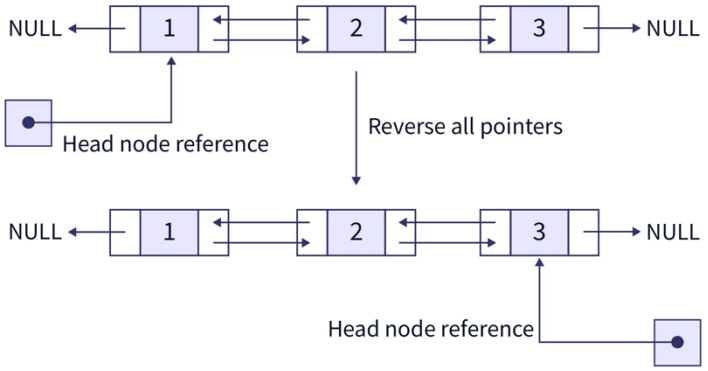
Backward traversal is same as forward traversal, the only difference is we need to use the previous pointers of the nodes of a doubly linked list until we reach the start of the doubly linked list, which is again denoted by the NULL pointer. This can also be achieved with a simple while loop in C++.

Follow the previous pointer for backward traversal

Syntax

Reverse a DLL

Reversing a doubly linked list involves reversing or swapping the next and previous pointers of all the nodes. So basically to reverse a doubly linked list we need to traverse the linked list and just swap the next and previous pointers of all the nodes.



Syntax

```
// to reverse the list we just swap the next and prev of all the nodes
    while (current != NULL) {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }
```

Advantage:

- A doubly linked list has both next as well as previous pointer unlike singly linked list which has only the next pointer.
- The doubly linked list can be traversed in both the forward as well in the backward direction as opposed to singly linked list which we can travel only in the forward direction.
- We can delete a node in a doubly linked list by just having the pointer to the node to be deleted. In a singly linked list we need the node before the node to be deleted.

Circular Linked List

A circular linked list is a type of linked list in which the **first** and the **last** nodes are also connected to each other to form a circle. Every node of the circular linked list consists of data value and the pointer to the next node and as it is circular, any node can be considered as the head node.

Syntax

```
// creating a structure using the struct keyword
struct Node
```

```
{
    // declaring a variable to store the data
    int data; // it could be of any data type (declaring integer here)

    // declaring a pointer to store the address of the next node
    struct Node* next; // here struct is the keyword to indicate the pointer of Node type
};
```

There are various operations we can perform on a circular linked list like **insertion**, **deletion**, and **traversal**. To keep the address of the linked list we will create a global pointer, let's say **'head'**, and initially, it will contain **'NULL'**.

```
// creating a pointer to keep track of head
Node* head = NULL;
```

Insertion

Insertion in an empty list syntax.

```
// checking condition if the list is empty or not
if(head == NULL)
{
    // assigning head the reference of the new node
    head = new_node;
    // to make it circular, assign new_node reference to its next
    head->next = head;
}
```

There are three possible positions to insert a new node in the circular linked list:

1. Insertion at the beginning of the list
2. Insertion at the end of the list.
3. Insertion in between the nodes.

Deletion

Linked lists or circular linked lists are created using dynamic memory allocation and allocated memory only deallocates when the program ends or the user manually deletes it using the delete function of C++. If the user forgets to manually deallocates the memory then it may arise the problem of a memory leak where the program terminates due to lack of memory. To prevent this problem use the delete() function to deallocate out-of-use memory blocks.

```
// deallocating memory
    delete(temp);
```

There are three different types of possible locations from where the user may delete the node:

1. **Deletion at the beginning of the list**

   For the deletion of the node at the beginning, need to do three operations:
   - Assigning reference of head to the next pointer of the head itself.
   - As the second node is going to be the head of the circular list, update the address of the next node.
   - Finally, delete the previous head pointer.
2. **Deletion at the end of the list.**

   For the deletion of the node at the end, we have to do two operations:
   - Assigning reference of head to the second last node because this is going to be the last node after deletion.

- Finally, delete the previous last pointer.

3. **Deletion in between the nodes.**
For deleting a node between two nodes, we are provided with two nodes let's say first and second, we have to do two operations to delete a node between them:
   - Deallocating the memory of the node next to the first node.
   - Assigning reference of the second node to the next pointer of the first node, as the middle node is deleted.

Traversal

Traversal for a circular linked list means visiting over the nodes and for this, we only need a single pointer which is the head or any node of the circular linked list.
As the whole list is connected to each other in the cycle which means we can visit every node and can get back to the starting point.

```
// creating temporary node
temp = head;

while(temp->next != head)
{
    temp = temp->next;
}

// after traversing the loop we will get back to head
```

Advantages

There are many advantages of the circular linked list:

- In a circular linked list, the last node is connected to the first one which makes circular operations like scheduling easy.
- Every node can be considered as a head node or first node which makes traversal over the list easy.
- There is no need to initialize the next pointer with a NULL value as there will always be an option for the next node.
- It reduces the time complexity for many operations in some advanced data structures.
- Scheduling algorithms like Round Robing can be set up without worrying about NULL pointer reference.