

THE UNIVERSITY  
OF LAHORE  
**ISLAMABAD  
CAMPUS**

**Data Structures and Algorithms  
(CS09203)**

**Lab Report**

Name: Rehman ullah baig

Reg No: CSU-F16-118

Lab Report: 4

Submitted To: Sir, Usman Ahmed

## Experiment # 04

### Link-list Basic Insertion and Traversal

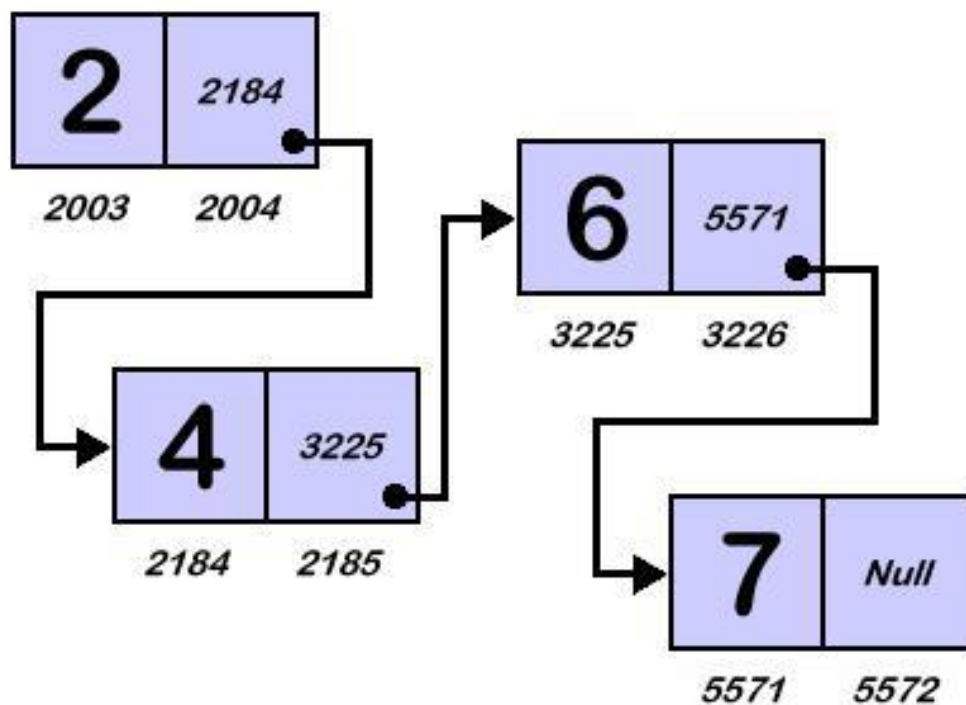
#### Objective:-

The objective of this session is to understand the various operations on linked list in C++.

#### Software Tools:-

1. Dev C++

#### Theory:-



You have already seen how data is organized and processed sequentially using an array, called a sequential list. You have performed several operations on sequential lists, such as sorting, inserting, deleting, and searching. You also found that if data is not sorted, searching for an item in the list can be very time consuming, especially with large lists. Once the data is sorted, you can use a binary search and improve the search algorithm. However, in this case, insertion and deletion become time consuming, especially with large lists because these operations require data movement. Also, because the array size must be fixed during execution, new items can be added only if there is room. Thus, there are limitations when you organize data in an array.

This session helps you to overcome some of these problems. Last session showed how memory (variables) can be dynamically allocated and deallocated using pointers. This session uses pointers to organize and process data in lists, called linked lists. Recall that when data is stored in an array, memory for the components of the array is contiguous—that is, the blocks are allocated one after the other. However, as we will see, the components (called nodes) of a linked list need not be contiguous.

#### LINKED LIST:-

A linked list is a collection of components, called nodes. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (that is, data) and one to store the address, called the link, of the next node in the list. The address of the first node in the list is stored in a separate location, called the head or first. Figure 1 is a pictorial representation of a node.

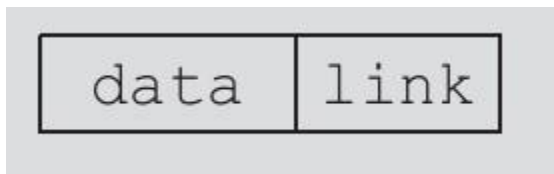


Figure 1

**Linked list:** A list of items, called **nodes**, in which the order of the nodes is determined by the address, called the **link**, stored in each node.

The list in Figure 2 is an example of a linked list.

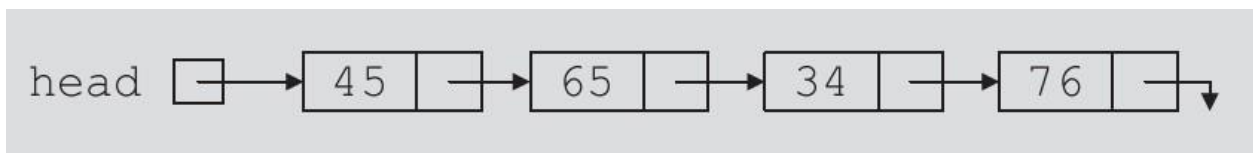


Figure 2

The arrow in each node indicates that the address of the node to which it is pointing is stored in that node. The down arrow in the last node indicates that this link field is **NULL**.

For a better understanding of this notation, suppose that the first node is at memory location 1200, and the second node is at memory location 1575, see Figure 3.

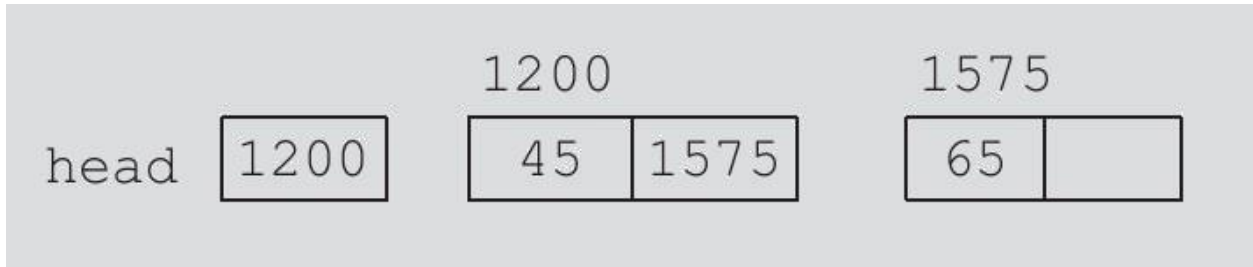


Figure 3

The value of the head is 1200, the data part of the first node is 45, and the link component of the first node contains 1575, the address of the second node. We will use the arrow notation whenever we draw the figure of a linked list.

For simplicity and for the ease of understanding and clarity, Figures 3 through 5 use decimal integers as the values of memory addresses. However, in computer memory the memory addresses are in binary.

Because each node of a linked list has two components, we need to declare each node as a **class** or **struct**. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is **int**.)

```
struct nodeType
{
    int info;
    nodeType* link;
};
```

The variable declaration is as follows:

***nodeType\* head;***

### Linked Lists: Some Properties

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 4.

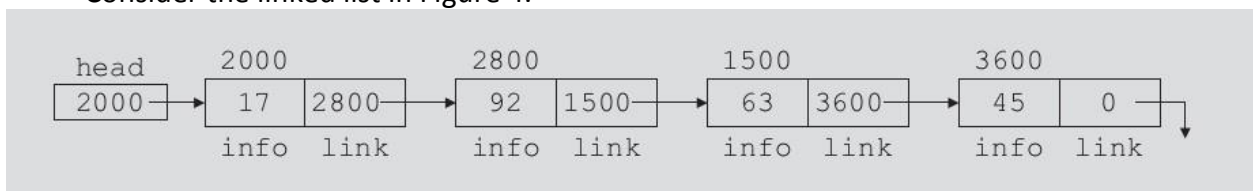


Figure 4

This linked list has four nodes. The address of the first node is stored in the pointer head. Each node has two components: info, to store the info, and link, to store the address of the next node. For simplicity, we assume that info is of type int. Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table 1 shows the values of head and some other nodes in the list shown in Figure 4.

Table 1: Values of head and some of the nodes of the linked list in Figure 4

	Value	Explanation
head	2000	
head->info	17	Because head is 2000 and the info of the node at location 2000 is 17
head->link	2800	
head->link->info	92	Because head->link is 2800 and the info of the node at location 2800 is 92

Suppose that current is a pointer of the same type as the pointer head. Then the statement

```
current = head;
```

Copies the value of head into current. Now consider the following statement:

```
current = current->link;
```

This statement copies the value of **current->link**, which is **2800**, into **current**. Therefore, after this statement executes, **current** points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 5.

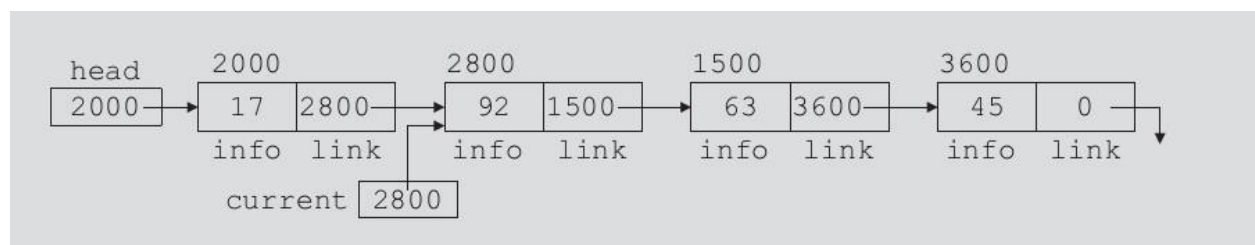


Figure 5: List after the statement `current = current->link;` executes

Table 2 shows the values of current, head, and some other nodes in Figure 5.

Table 2: Values of current, head, and some of the nodes of the linked list in Figure5

	Value
current	2800
current->info	92
current->link	1500
current->link->info	63
head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
current->link->link->link	0 (that is, NULL)
current->link->link->link->info	Does not exist (run-time error)

From now on, when working with linked lists, we will use only the arrow notation.

#### **TRAVERSING A LINKED LIST:-**

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, and delete an item from the list. These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

#### **Algorithm:-**

**(Traversing a Linked List)** Let LIST be a linked list in memory. This algorithm traverses LIST applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR=START [Initialize pointer PTR]
2. Repeat Step 3 and 4 while PTR  $\neq$  NULL.
3. Apply PROCESS TO INFO[PTR].
4. Set PTR=LINK[PTR] [PTR now points to the next node] [End of Step 2 Loop]
5. Exit.

Suppose that the pointer **head** points to the first node in the list, and the link of the last node is **NULL**. We cannot use the pointer **head** to traverse the list because if we use the **head** to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move **head** to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical because it would require additional computer time and memory space to maintain the list).

Therefore, we always want **head** to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that **current** is a pointer of the same type as **head**. The following code traverses the list:

```
    current = head;
    while(current != NULL)
    {
        cout << current->info << " ";
        current = current->link;
    }
```

## Conclusion

---

---

---

---

---

-----  
-----  
(Concerned Teacher/Lab Engineer)

## Insertion and Traversing in Linked List

### Code:

```
#include<iostream>
using namespace std;
class Node
{
private:
    int Data;
    Node *Next;
public:
    void insert(int x);
    void print();
};
Node *Head;
void Node::insert(int x)
{
    Node *temp = new Node();
    temp->Data = x;
    temp->Next = Head;
    Head = temp;
}

void Node::print()
{
    Node *temp = Head;
    cout<<"\n\nList is: ";
    while(temp != NULL)
    {
        cout<<temp->Data<<" ";
        temp = temp->Next;
    }
    cout<<"\n\n";
}

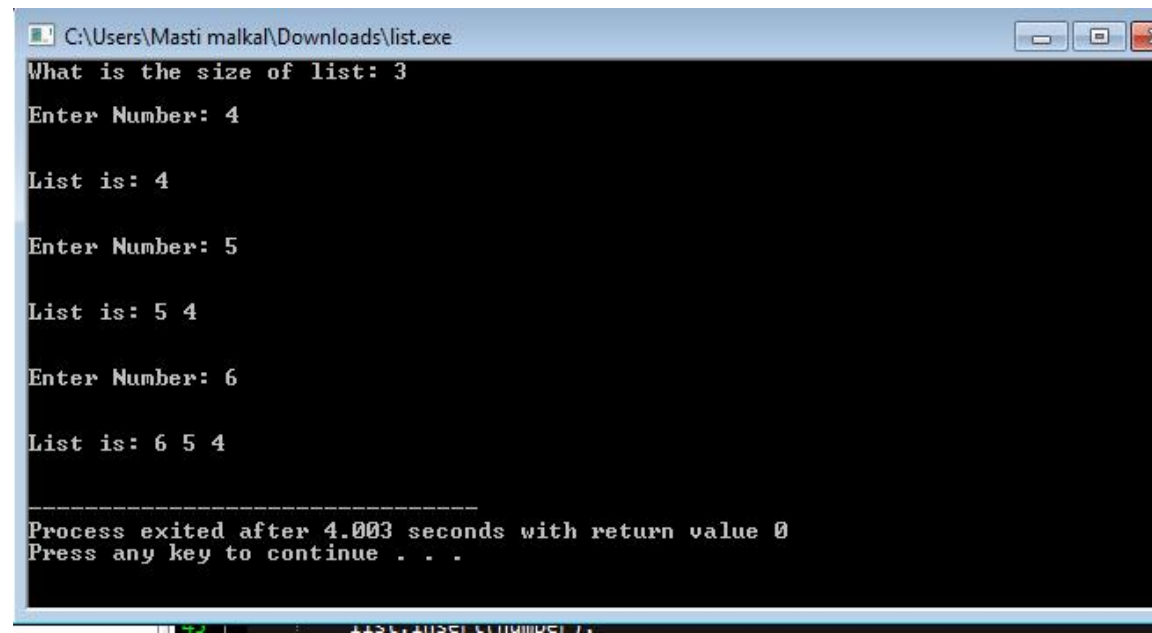
main()
{
    Node list;
    Head = NULL;
    int size, number;
    cout<<"What is the size of list: ";
    cin>>size;
    for(int i = 1; i<=size; i++)
    {
        cout<<"\nEnter Number: ";
```



```
        cin>>number;
        list.insert(number);
        list.print();
    }

    return 0;
}
```

### **Output:**



```
C:\Users\Masti malkal\Downloads\list.exe
What is the size of list: 3
Enter Number: 4
List is: 4
Enter Number: 5
List is: 5 4
Enter Number: 6
List is: 6 5 4
-----
Process exited after 4.003 seconds with return value 0
Press any key to continue . . .
```