Data Structures
BS-III Fall 2019
LAB NO: 04_1

Name	
Student ID	
Date	

Objectives:

After completing this Lab students will able to

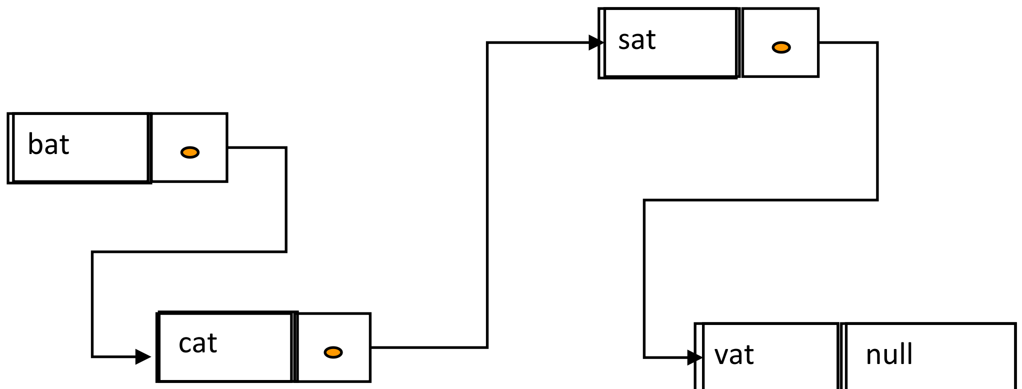
1. Understand the concept and usage of Linked Lists in programming.
2. Implementation of Dictionary Operations on Linked List.

Objective 1: *Understanding Linked Lists.*

A linked list is an ordered and simple linear data structure which can change during execution.

- Successive elements are connected by pointers.
- Last element points to NULL.
- It can grow or shrink in size during execution of a program.
- It can be made just as long as required.
- It does not waste memory space.
- The nodes do not reside in sequential locations.
- The locations of the nodes may change on different runs.

A linked list is a complex data structure, especially useful in systems or applications programming. A linked list is comprised of a series of nodes, each node containing a data element, and a pointer to the next node, eg,



A structure which contains a data element and a pointer to the next node is created by,

```
    struct Node {
            int   value;
            Node  *next;
      };
```

Array versus Linked Lists:

- Arrays are suitable for:
    - Inserting/deleting an element at the end.
    - Randomly accessing any element.
    - Searching the list for a particular value.
- Linked lists are suitable for:
    - Inserting an element randomly.
    - Deleting an element randomly.
    - Applications where sequential access is required.
    - In situations where the number of elements cannot be predicted beforehand.

Objective – 2: ***Implementing of Linked List.***

Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting node in the list
- Appending the list
- Searching within list
- Deleting an item from the list
- Counting nodes in a list

Creating a List

To start with, we have to create a node (the first node), and make head point to it.

Traversing a List

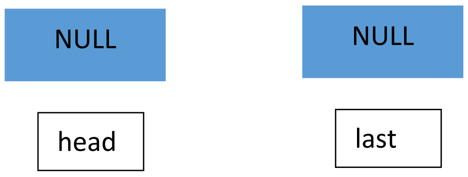
Once the linked list has been constructed and *head* points to the first node of the list,

- Follow the pointers.
- Display the contents of the nodes as they are traversed.
- Stop when the *next*pointer points to NULL.

Insertion at the last: One of the simplest insertion algorithm of linked list is to always insert the new node at the end of the linked list. For such insertion, one of the fastest approach is to not only keep a pointer at the first node, but also keep a pointer at the last (Node *last) node. There are two cases for the insertion

Inserting at the last of an empty list:

Initially, when linked list is empty,



If we want to insert a new node with the data = 5 into the empty list

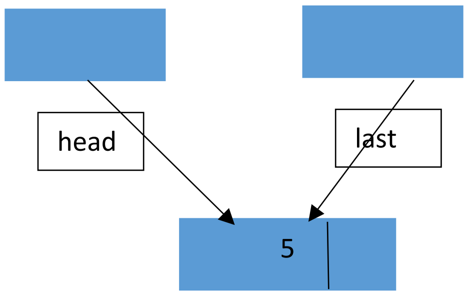
```
Node* newNode = new Node;
newNode->data = data;
```

The instructions above will create a new node and will store the data into it.

Then we need both the head and last to point towards this node

```
if (!head)
                {
                        head = newNode;
                        last = newNode;
                }
```

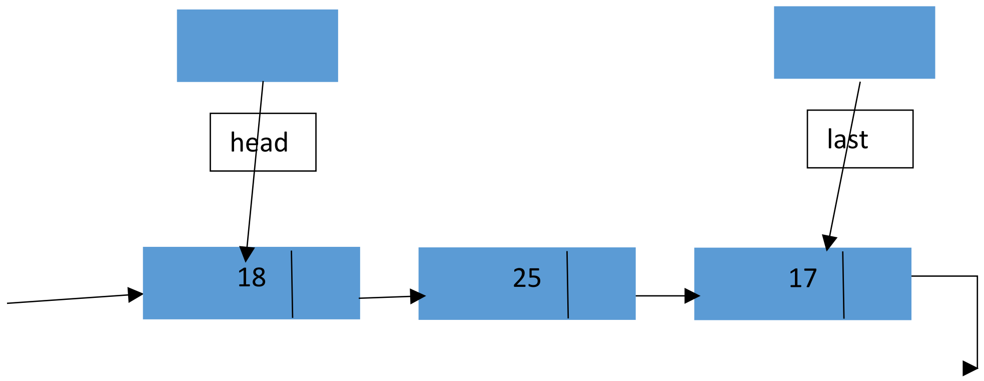
It will result in the following linked list



Data Structures
BS-III Fall 2019
LAB NO: 04_1

Inserting at the last of a non-empty list:

Suppose that someone wants to insert a node at the last of a non-empty list. The pointer named last will help us to link the new node at the last of the linked list. Let's say that the linked list before insertion is the following



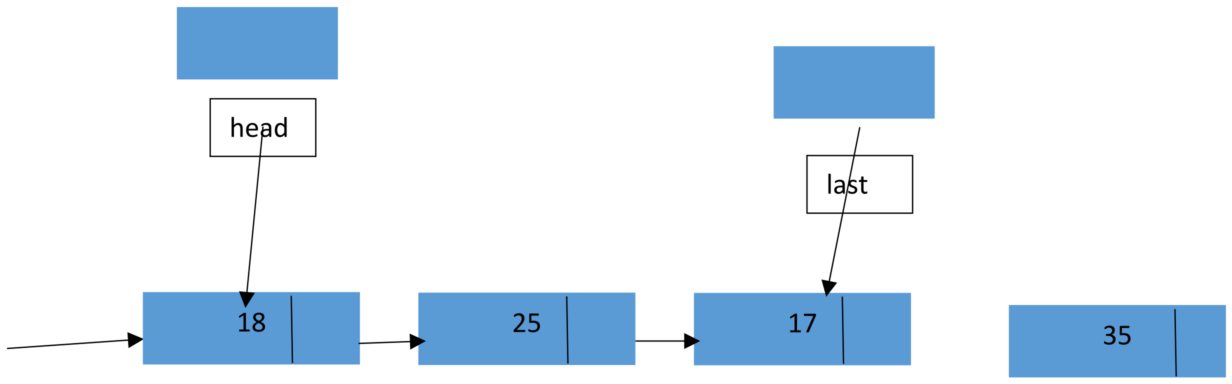
If we want to insert the node with the data value of 35 in the linked list, the following code will perform the insertion for the second case

```
Node* newNode = new Node;
newNode->data = data;
```

The instructions above will create a new node and will store the data into it.

And the following instruction will link the previous last node with the new last node

```
last->next = newNode;
```



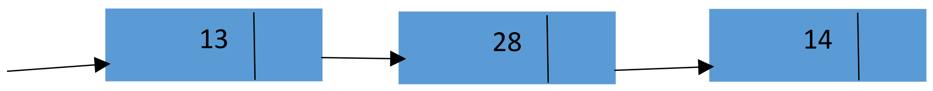
Last, but not least, we have to move last pointer to the new last node using the following instruction
last = newNode;

Task1: The following code contain two classes named Node and List. The list function implements a function named insertAtLast that implements the algorithm of insertion discussed above.

```
struct Node
{
        int data;
        Node* next;
        Node()
        {
                next = NULL;
        }
};
class List
{
        Node* head, *last;
public:
        List() { head = NULL; last = NULL; }
        void insertAtLast(int data)
        {

                Node* newNode = new Node;
                newNode->data = data;
                if (!head)
                {
                        head = newNode;
                        last = newNode;
                }
                else
                {
                        last->next = newNode;
                        last = newNode;
                }
        }
void printAll();
};
```

Remember that the class List has prototype of a **non-recursive** function named "printAll", however, the body is missing. Implement the body of this function in such as way that it prints the data part of all nodes in forward order. For example, in case of the following linked list



The output should be

13

28

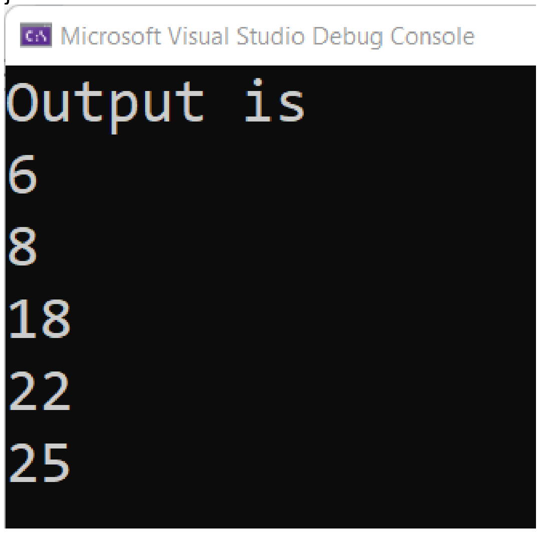
14

In order to verify the implementation, I am providing the test cases along with their expected outputs

Test case 1:

```
void main()
{
	List obj;
	obj.insertAtLast(6);
	obj.insertAtLast(8);
	obj.insertAtLast(18);
	obj.insertAtLast(22);
	obj.insertAtLast(25);
	cout << "Output is" << endl;
	obj.printAll();

}
```

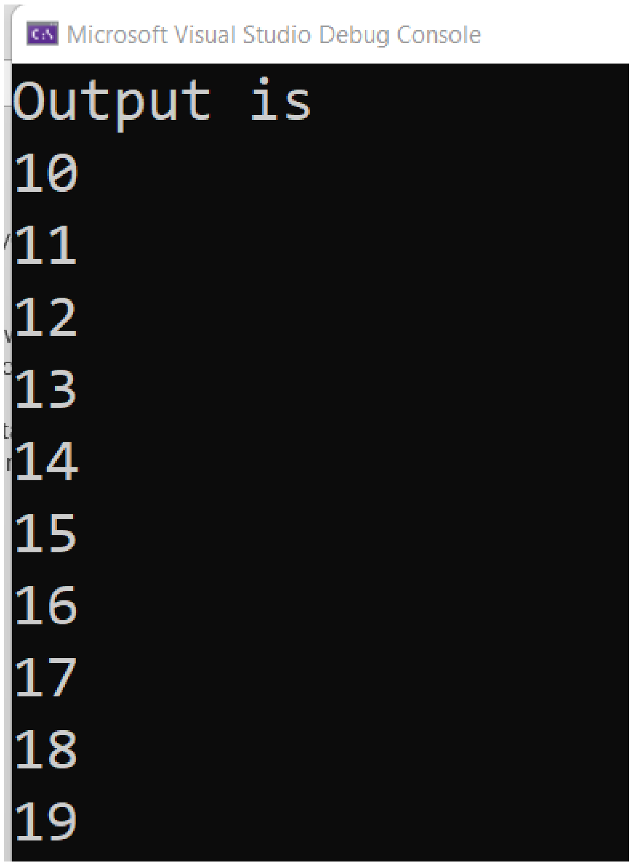


```
Output is
6
8
18
22
25
```

Test case 2:

```
void main()
{
	List obj;
	for (int i = 0; i < 10; i++)
	{
		obj.insertAtLast(i+10);
	}
	cout << "Output is" << endl;
```

```
        obj.printAll();
}
```



Microsoft Visual Studio Debug Console

```
Output is
10
11
12
13
14
15
16
17
18
19
```

Task 2. The following recursive function print the data of all the nodes in forward order.

```
void printRec(Node *curr)
        {
                if (curr)
                {
                        cout << curr->data << endl;
                        printRec(curr->next);
                }
        }
```

This function should be called from inside of the List class as it required the value of head (private member of the List class) pointer in the first call. Therefore, I called the function using the following set of instructions

```
void printAll()
        {
                    printRec(head);
        }
        void printRec(Node *curr)
        {
                    if (curr)
                    {
                                cout << curr->data << endl;
                                printRec(curr->next);
                    }
        }
```

Modify this function so that it prints the list in reverse order. You should keep the function recursive.

Task 3: Non-credit task

Implement a function named "insertAtFront" that always insert the new node to the front of the linked list.

Note: You do not require last pointer in such function.