4.1. Rewrite the `main` function in Listing 4 to build and display a linked list as follows \{12, 5, 79, 82, 21, 43, 31, 35, 57\}.

```c
void main() {
    List aList;
    aList.addFirst(12);
    aList.addFirst(5);
    aList.addFirst(79);
    aList.addFirst(82);
    aList.addFirst(21);
    aList.addFirst(43);
    aList.addFirst(31);
    aList.addFirst(35);
    aList.addFirst(57);
    aList.display();
}
```

```c
void main() {
    List aList;
    aList.addFirst(57);
    aList.addFirst(35);
    aList.addFirst(31);
    aList.addFirst(43);
    aList.addFirst(21);
    aList.addFirst(82);
    aList.addFirst(79);
    aList.addFirst(5);
    aList.addFirst(12);
    aList.display();
}
```

Listing 5. Left (print in the reverse order) – Right (print in the same order)

4.2. Rewrite the `main` function in Exercise 4.1 to do the following tasks:

- use the `buildLinkedList` function to create a list based on input from user.
- display the list
- free the memory allocated to the list

```c
void main() {
    List *aList;
    aList = buildLinkedList();
    aList->display();
    delete aList;
}
```

Listing 6. The `buildLinkedList` return a pointer

This operation releases the memory holding by the pointer `aList`, which points to a List object.

Note that, before doing the release, the destructor of the class List is called. As given in Listing 3, the destructor of the class List clears all elements of the linked list.

Further note that, if the class List has no destructor, or the destructor does not clear the elements of the linked list, those elements are still in the memory and cannot be accessed any more.
4.3. Write for the class List in Listing 3 an additional method int addFirstIfPrime(int n) which adds \( n \) to the list if \( n \) is a prime integer. In that case the returned result is 1, otherwise 0.

*Note that, the following method will cause an error if the input newdata is negative (by the \texttt{sqrt} function)*

```cpp
int addFirstIfPrime(int newdata) {
    //check Prime
    bool isPrime = true;
    for (int i=2; i<sqrt(newdata) && isPrime; i++)
        if ((newdata % i) != 0)
            isPrime = false;
    if (isPrime) {
        Node* pTemp = new Node;
        pTemp->data = newdata;
        pTemp->next = pHead;
        pHead = pTemp;
        count++;
        return 1;
    }
    return 0;
}
```

**Listing 7. Left (Not use insertFirst method) – Right (Use insertFirst method)**

4.4. Write for the class List in Listing 3 an additional method int addLast(int n) which adds \( n \) to the last position of the list.

```cpp
int addLast(int newdata) {
    //if the list is empty, insert first
    if (head == NULL) {
        insertFirst(newdata);
    } else {
        //position to the last
        Node* last = head;
        while (last->link != NULL)
            last = last->link;
        //connect to the last
        Node* pTemp = new Node;
        pTemp->data = newdata;
        pTemp->link = NULL;
    }
    return 1;
}
```
//connect to the last
pTemp->link = NULL;
last->link = pTemp;
}
count++;
return 1;
}

Listing 8. Left (Not use insertFirst method) – Right (Use insertFirst method)

4.5. Implement the following method

\[
\text{int findMax(List* pList)}
\]

For that specification, this method receives an input pointer pList of the List object and will return the maximum value from the elements of the input list. Note that, the object on which the method is called has not been touched. For example, the result printed to the screen of the following program is 1:

```c
void main() {
    List * aList, *bList;
    aList->insertFirst(2);  //aList contains the element 2
    bList->insertFirst(1);  //bList contains the element 1
    cout << aList->findMax(bList);   //the maximum value of the input object bList will be returned
}
```

```c
int findMax(List *pList) {
    //max is the data of the first element
    int max = pList->head->data;   //head pointer is of the input object
    //the list is not empty
    Node* tmp = pList->head->link;   //head pointer is of the input object
    while (tmp != NULL) {
        if (tmp->data > max)
            max = tmp->data;
        tmp = tmp->link;
    }
    return max;
}
```

Listing 6. The findMax as a method can use the head pointer of the input List object

**Discussion**

Further note that, if we would like to find the maximum on the List object on which the method is called, the specification should be somewhat \text{int findMax1()}. In this case, we need to implement the method so that it does not receive any input, and will search on the called object for the result. For example, the result printed to the screen of the following program is 2:

```c
void main() {
    List * aList, *bList;
    aList->insertFirst(2);  //aList contains the element 2
    bList->insertFirst(1);  //bList contains the element 1
    cout << aList->findMax1();   //the maximum value of the input object bList will be returned
}
```
void main() {
    List * aList, *bList;
    aList->insertFirst(2); //aList contains the element 2
    bList->insertFirst(1); //bList constains the element 1
    cout « aList->findMax1(); //the maximum value of the called object aList will be returned
}

int findMax1() {
    //max is the data of the first element
    int max = head->data; //head pointer is of the called object
    //the list is not empty
    Node* tmp = head->link; //head pointer is of the called object
    while (tmp != NULL) {
        if (tmp->data > max)
            max = tmp->data;
        tmp = tmp->link;
    }
    return max;
}

Listing 7. The findMax1 searches on the linked list of the called object