# Department of Computing

## **Name : Mahum Samar**

## **CMS ID : 290647**

**CS250: Data Structure and Algorithms**

## **Class: BSCS 9B**

# **ASSIGNMENT # 01**

# **Code**

#include <iostream>

using namespace std;

class ListNode{

public:

int data;

ListNode \*next;

};

class SinglyLinkedList{

public:

ListNode \*headNode; // special variable which stores address of the head node.

ListNode \*lastNode; // special variable which stores address of the last node.

ListNode \*preLoc; //to be used by Search(value) method to store address of logical predecessor of value in a list.

ListNode \*loc; //to be used by Search(value) method to store address of the node containing the searched value in a list. If it is not found it contains NULL.

//constructor for creating the empty list

SinglyLinkedList(){

headNode=NULL; //headNode to point to the node at the start of the list

preLoc=NULL; //preLoc fro the predecessor node of the loc

loc=NULL; //loc to assign tho the node which is required to be found

}

//method for checking whether the singly linked list is empty or not.

bool isEmpty()

{

return headNode == NULL;

}

//method for inserting the value at the headNode

void insertAtFront(int value)

{

ListNode \*newNode = new ListNode();

newNode -> data = value;

if(isEmpty())

{

//executes if the list is empty

headNode = newNode;

lastNode = newNode;

}

else

{

//executes if the list is not empty

newNode -> next = headNode;

headNode = newNode;

}

printList(); //prints the list after insertion at front

}

//method for inserting the value at the lastNode

void insertAtEnd(int value)

{

ListNode \*newNode = new ListNode();

newNode -> data = value;

if(isEmpty())

{

//executes if the list is empty

headNode = newNode;

lastNode = newNode;

}

else

{

//executes if the list is not empty

lastNode -> next = newNode;

lastNode = newNode;

}

printList(); //prints the list after insertion at end

}

//method for printing the list on the screen

void printList()

{

//method to print the list by using the temporary pointer.

ListNode \*tempNode = headNode;

if(!isEmpty())

{

while(tempNode != NULL)

{

cout << tempNode -> data << endl;

tempNode = tempNode -> next;

}

}

}

//method for searching the node which contains the value entered by the user.

void searchNode(int value)

{

preLoc = NULL;

loc = headNode;

if (isEmpty())

{

//returns if the list is empty.

return;

}

while(loc != NULL && loc -> data < value )

{

//this loop executes until loc is not null and logical position of the value is also not found.

preLoc = loc;

loc = loc -> next;

}

if(loc != NULL && loc -> data != value)

{

//if the value is not found the loc is assigned NULL.

loc = NULL;

}

}

void printReverse(ListNode \*headNode)

{

//Question # 1 method

//method to print the linked list in the reverse order by using the recursion.

if(isEmpty())

{ //if the list is empty.

cout<< "list is empty." << endl;

}

cout<< "Printing list in reverse order." << endl;

if (headNode == NULL)

{

//when the pointer crosses the headNode by going through recursion

return;

}

//recursion call

printReverse(headNode -> next);

//printing the data

cout << headNode -> data << endl;

}

void ReverseOrder()

{

//Question # 3 method

//method for reversing the list.

if(isEmpty())

{

//if the list is empty.

cout<< "list is empty." << endl;

return;

}

cout<< "printing list in reverse order." << endl;

ListNode \*head = headNode; //pointer variable to store the address of the headNode.

loc = headNode; //loc initialized with he headNode of the list.

preLoc = NULL;

ListNode \*temp = NULL; //temp pointer variable to help in reversing the list by keeping the address of the current node.

while(loc != NULL)

{

// reverses the addresses in the list.

temp = loc -> next;

loc -> next = preLoc;

preLoc = loc;

loc = temp;

}

headNode = preLoc; //headNode transfered to the lastNode

lastNode = head; //the riginal headNode becomes the lastNode.

printList(); //displaying the result on the screen.

}

void DeleteOddValues()

{

//Question # 2 method

//method to delete the odd values in the linked list.

if( isEmpty ())

{

//if the list is empty.

cout<< "list is empty." << endl;

return;

}

preLoc = NULL;

loc = headNode;

//loop to delete all the odd values in the list.

while(loc != NULL)

{

//if condition executed if the value in the loc node is odd.

if(((loc -> data)%2) == 1)

{

//if odd value is at the headNode

if(preLoc == NULL)

{

if(loc -> next == NULL)

{

//if single node

delete headNode;

}

else

{

//if headNode is not the single node in the list.

preLoc = loc;

loc = loc -> next;

headNode = loc;

delete preLoc;

}

}

else //if the odd value is after the headNode.

if (preLoc != NULL)

{

if (loc -> next != NULL)

{

//if in middle

preLoc -> next = loc -> next;

delete loc;

loc = preLoc -> next;

}

else

{

//if lastNode

lastNode = preLoc;

delete loc;

loc = lastNode;

loc -> next = NULL;

}

}

}

else

{

//if the odd value not found the preLoc and loc are incremented.

preLoc = loc;

loc = loc -> next;

}

}

printList();

}

void GroupingEvenOdd()

{

//Question # 4 method

//method for grouping the even and odd nodes. Even nodes at front and then odd nodes at the end.

if( isEmpty ())

{

//if the list is empty.

cout<< "list is empty." << endl;

return;

}

//creating the new singlyLinkedList to store the odd values and then attaching it to the remaining list of the even valued nodes.

SinglyLinkedList \*sll = new SinglyLinkedList();

preLoc = NULL;

loc = headNode;

sll -> loc = sll -> headNode; //initializing the loc of sll with the sll headNode

while(loc != NULL)

{

//loop to separate the even and odd valued nodes.

//if the value is odd

if(((loc -> data)%2) == 1)

{

if(preLoc == NULL)

{

if(loc -> next == NULL)

{

//if list has only one node.

cout << "only one node in the list." << endl;

break;

}

else

{

//if at headNode and not the only node.

//that odd valued node is moved to the sll and removed from the original list.

if(sll -> headNode == NULL)

{

//if nothing in the list

sll -> headNode = loc;

sll -> lastNode = loc;

sll -> loc = sll -> headNode;

}

else

{

//if list already contains some values attaching the odd value at the end.

sll -> loc -> next = loc;

sll -> loc = sll -> loc -> next;

sll -> lastNode = loc;

}

preLoc = loc;

loc = loc -> next;

headNode = loc;

}

}

else //if the odd value is after the headNode

if (preLoc != NULL)

{

if (loc -> next != NULL)

{

if(sll -> headNode == NULL)

{

sll -> headNode = loc;

sll -> lastNode = loc;

sll -> loc = sll -> lastNode;

}

else

{

sll -> loc -> next = loc;

sll -> loc = sll -> loc -> next;

sll -> lastNode = loc;

}

// if in middle

preLoc -> next = loc -> next;

loc = preLoc -> next;

}

else

{

//if the lastNode

if(sll -> headNode == NULL)

{

sll -> headNode = loc;

sll -> lastNode = loc;

sll -> loc = sll -> lastNode;

}

else

{

sll -> loc -> next = loc;

sll -> loc = sll -> loc -> next;

sll -> lastNode = loc;

}

lastNode = preLoc;

loc = lastNode;

loc -> next = NULL;

}

}

}

else

{

//if its even valued node, preloc and loc are just incremented

preLoc = loc;

loc = loc -> next;

}

}

if(sll -> headNode -> next == NULL)

{

//if the odd values are not found in the list

cout<< "noo odd values." << endl;

}

else

{

//attaching the odd valued list at the end of the even valued list.

lastNode -> next = sll -> headNode;

sll -> lastNode = lastNode;

}

printList();

}

void SwappingNodes()

{

//we have assumed that that list is already sorted.

int num1, num2;

cout << "Enter the first Number: " << endl;

cin >> num1 ;

cout << "Enter the second Number: " << endl;

cin >> num2 ;

int temporary = 0;

if(num1 > num2)

{

//if user enter the 2nd number which is smaller than 1st than the numbers are swapped.

temporary = num1;

num1 = num2;

num2 = temporary;

}

searchNode(num1);

//storing preLoc and loc positions in other pointer variable, returned by the searchNode() method

ListNode \*preloc1 = preLoc;

ListNode \*loc1 = loc;

searchNode(num2);

ListNode \*preloc2 = preLoc;

ListNode \*loc2 = loc;

if((loc2 == NULL) || (loc1 == NULL))

{

cout << "The numbers are not present in the list so cannnot be swapped." << endl;

return;

}

ListNode \*temp = NULL;

if (preloc1 != NULL && loc2 != lastNode)

{

if(loc1 -> next != loc2)

{

//case 1 not adjescent, not 1st and last node

temp = loc2 -> next;

loc2 -> next = loc1 -> next;

loc1 -> next = temp;

if (preloc1 != NULL)

{

preloc1 -> next = loc2;

preloc2 -> next = loc1;

}

}

else

{

//case 3 adjescent, but not 1st and last node

temp = loc2 -> next;

loc1 -> next = temp;

loc2 -> next = loc1;

preloc1 -> next = loc2;

}

}

else

if (preloc1 == NULL)

{

if(loc2 != loc1 -> next)

{

//case 2 if not adjescent, and one is headNode

temp = loc2 -> next;

loc2 -> next = loc1 -> next;

loc1 -> next = temp;

preloc2 -> next = loc1;

headNode = loc2;

}

else

{

//case 4 if adjecent

temp = loc2 -> next;

loc1 -> next = temp;

loc2 -> next = loc1;

headNode = loc2;

}

}

else

if(loc2 -> next == NULL && loc2 != loc1 -> next)

{

//case 2 if are not adjescent, and one is lastNode

loc2 -> next = loc1 -> next;

preloc1 -> next = loc2;

preloc2 -> next = loc1;

lastNode = loc1;

loc1 -> next = NULL;

}

else

if (loc1 == headNode && loc2 == lastNode && loc1 -> next == loc2)

{

//case 5 if there are only 2 nodes.

loc2 -> next = loc1;

headNode = loc2;

lastNode = loc1;

}

cout << "Now the list is: " << endl;

printList();

}

};

Main method for the questions

# **Question #1**

## Code:

int main()

{

//creating the object of the SinglyLinkedList class

SinglyLinkedList \*singlyLinkedList = new SinglyLinkedList();

//checking if the list is empty or not

if (singlyLinkedList -> isEmpty())

{

cout << "List is Empty." << endl;

}

else

{

cout << "List is not Empty." << endl;

}

cout << "Insertion at front." << endl;

cout << "Insertion of 4." << endl;

singlyLinkedList -> insertAtFront(4);

cout << "Insertion of 3." << endl;

singlyLinkedList -> insertAtFront(3);

cout << "Insertion of 1." << endl;

singlyLinkedList -> insertAtFront(1);

cout << "Insertion in sorted list ." << endl;

singlyLinkedList -> insertSorted(0);

cout << "Insertion at End." << endl;

singlyLinkedList -> insertAtEnd(5);

cout<< "Printing list in reverse order." << endl;

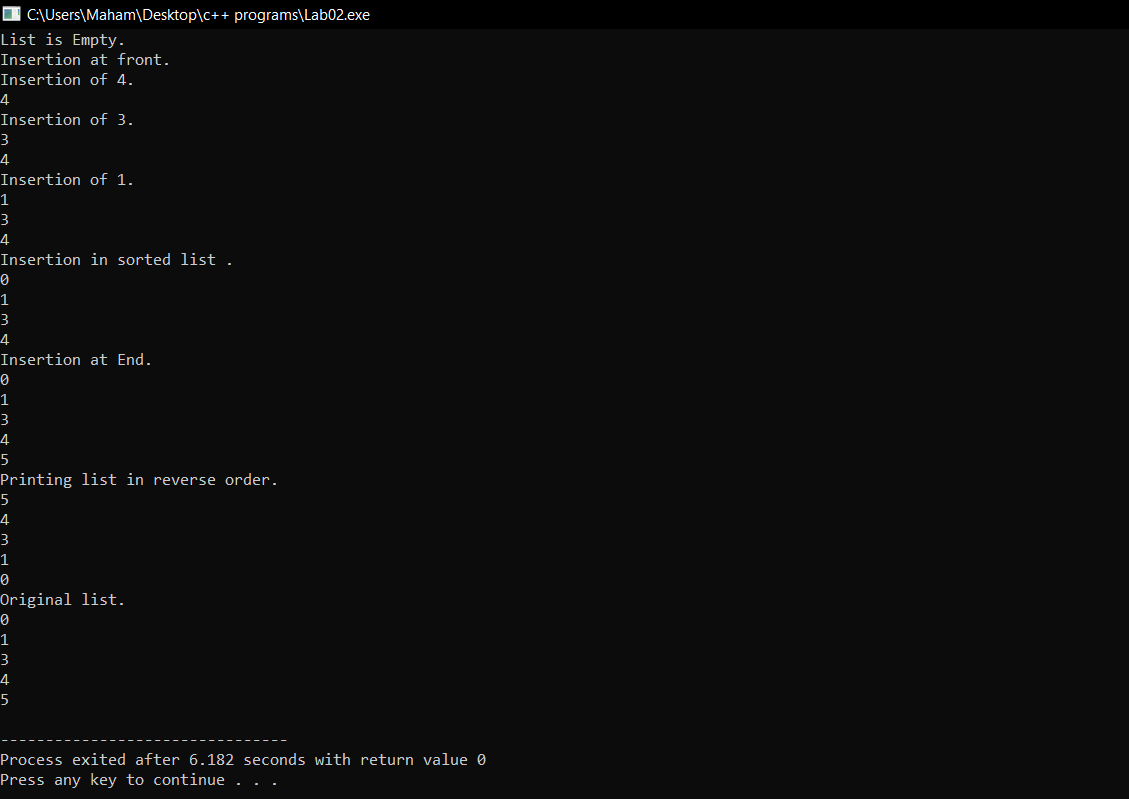
singlyLinkedList -> printReverse(singlyLinkedList -> headNode);

cout<< "Original list." << endl;

singlyLinkedList -> printList();

}

## Output:



# **Question # 2**

## Code:

int main()

{

//creating the object of the SinglyLinkedList class

SinglyLinkedList \*singlyLinkedList = new SinglyLinkedList();

//checking if the list is empty or not

if (singlyLinkedList -> isEmpty())

{

cout << "List is Empty." << endl;

}

else

{

cout << "List is not Empty." << endl;

}

cout << "Insertion at front." << endl;

cout << "Insertion of 4." << endl;

singlyLinkedList -> insertAtFront(4);

cout << "Insertion of 3." << endl;

singlyLinkedList -> insertAtFront(3);

cout << "Insertion of 1." << endl;

singlyLinkedList -> insertAtFront(1);

cout << "Insertion in sorted list ." << endl;

singlyLinkedList -> insertSorted(0);

cout << "Insertion at End." << endl;

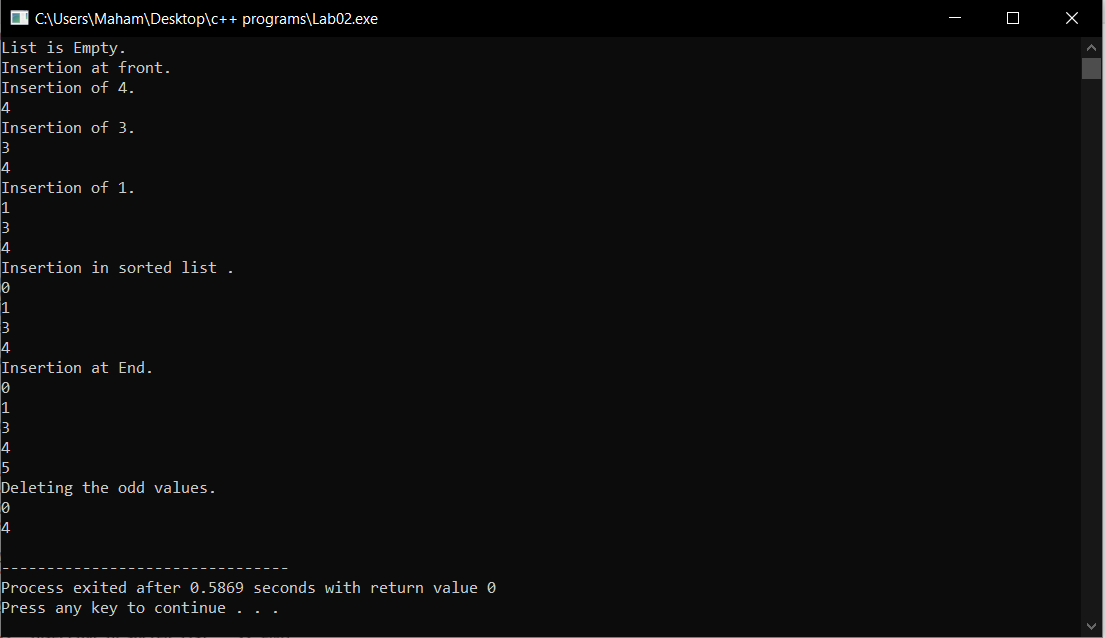
singlyLinkedList -> insertAtEnd(5);

cout << "Deleting the odd values." << endl;

singlyLinkedList -> DeleteOddValues();

}

## Output:



# **Question # 3**

## Code:

int main()

{

//creating the object of the SinglyLinkedList class

SinglyLinkedList \*singlyLinkedList = new SinglyLinkedList();

//checking if the list is empty or not

if (singlyLinkedList -> isEmpty())

{

cout << "List is Empty." << endl;

}

else

{

cout << "List is not Empty." << endl;

}

cout << "Insertion at front." << endl;

cout << "Insertion of 4." << endl;

singlyLinkedList -> insertAtFront(4);

cout << "Insertion of 3." << endl;

singlyLinkedList -> insertAtFront(3);

cout << "Insertion of 1." << endl;

singlyLinkedList -> insertAtFront(1);

cout << "Insertion in sorted list ." << endl;

singlyLinkedList -> insertSorted(0);

cout << "Insertion at End." << endl;

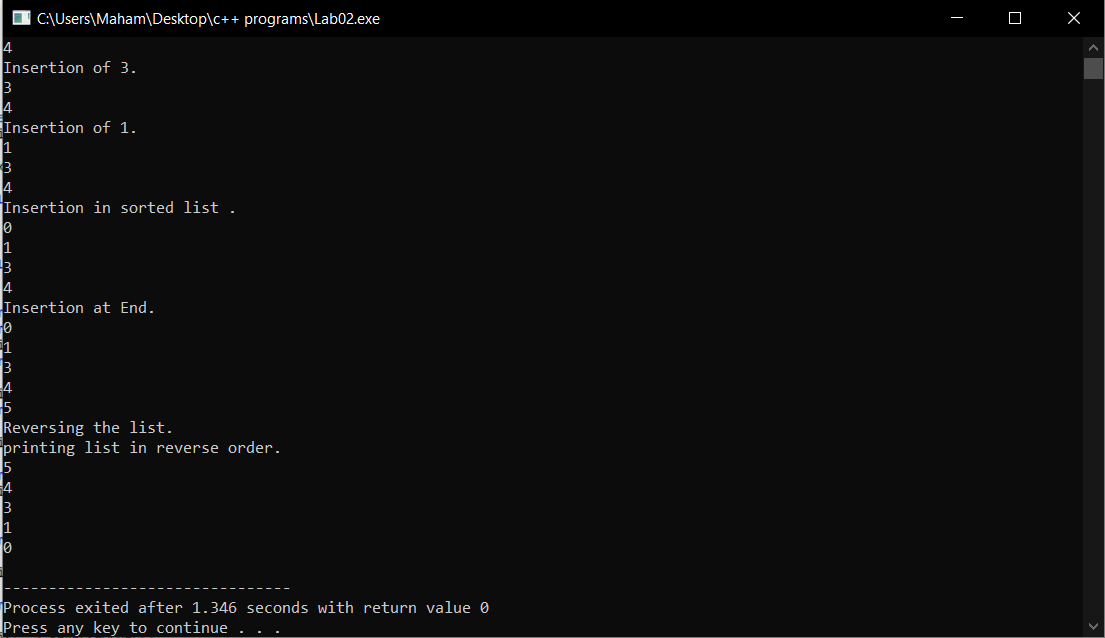
singlyLinkedList -> insertAtEnd(5);

cout << "Reversing the list." << endl;

singlyLinkedList -> ReverseOrder();

}

## Output:



# **Question # 4**

## Code:

int main()

{

//creating the object of the SinglyLinkedList class

SinglyLinkedList \*singlyLinkedList = new SinglyLinkedList();

//checking if the list is empty or not

if (singlyLinkedList -> isEmpty())

{

cout << "List is Empty." << endl;

}

else

{

cout << "List is not Empty." << endl;

}

cout << "Insertion at front." << endl;

cout << "Insertion of 4." << endl;

singlyLinkedList -> insertAtFront(4);

cout << "Insertion of 3." << endl;

singlyLinkedList -> insertAtFront(3);

cout << "Insertion of 1." << endl;

singlyLinkedList -> insertAtFront(1);

cout << "Insertion in sorted list ." << endl;

singlyLinkedList -> insertSorted(0);

cout << "Insertion at End." << endl;

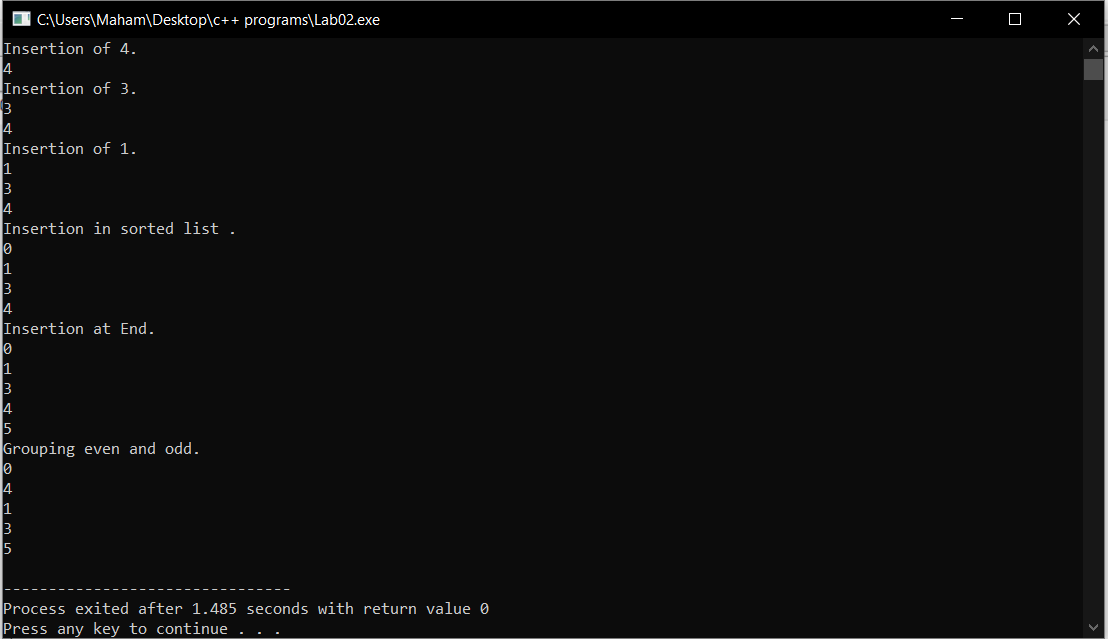
singlyLinkedList -> insertAtEnd(5);

cout << "Grouping even and odd." << endl;

singlyLinkedList -> GroupingEvenOdd();

}

## Output:



# **Question # 5**

## Code:

int main()

{

//creating the object of the SinglyLinkedList class

SinglyLinkedList \*singlyLinkedList = new SinglyLinkedList();

//checking if the list is empty or not

if (singlyLinkedList -> isEmpty())

{

cout << "List is Empty." << endl;

}

else

{

cout << "List is not Empty." << endl;

}

cout << "Insertion at front." << endl;

cout << "Insertion of 4." << endl;

singlyLinkedList -> insertAtFront(4);

cout << "Insertion of 3." << endl;

singlyLinkedList -> insertAtFront(3);

cout << "Insertion of 1." << endl;

singlyLinkedList -> insertAtFront(1);

cout << "Insertion in sorted list ." << endl;

singlyLinkedList -> insertSorted(0);

cout << "Insertion at End." << endl;

singlyLinkedList -> insertAtEnd(5);

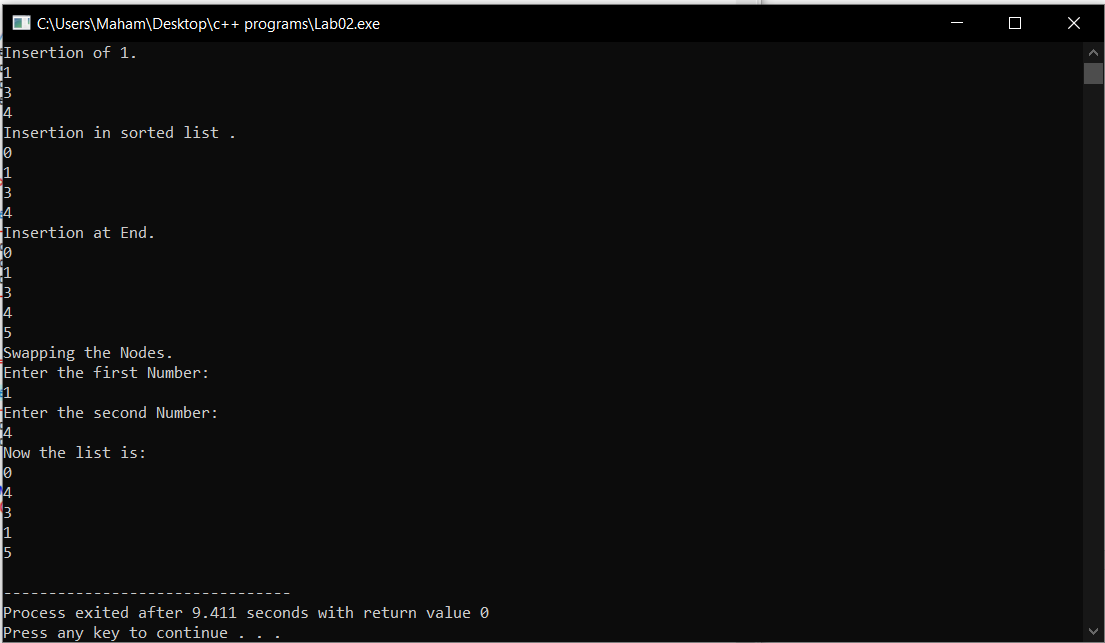
cout << "Swapping the Nodes." << endl;

singlyLinkedList -> SwappingNodes();

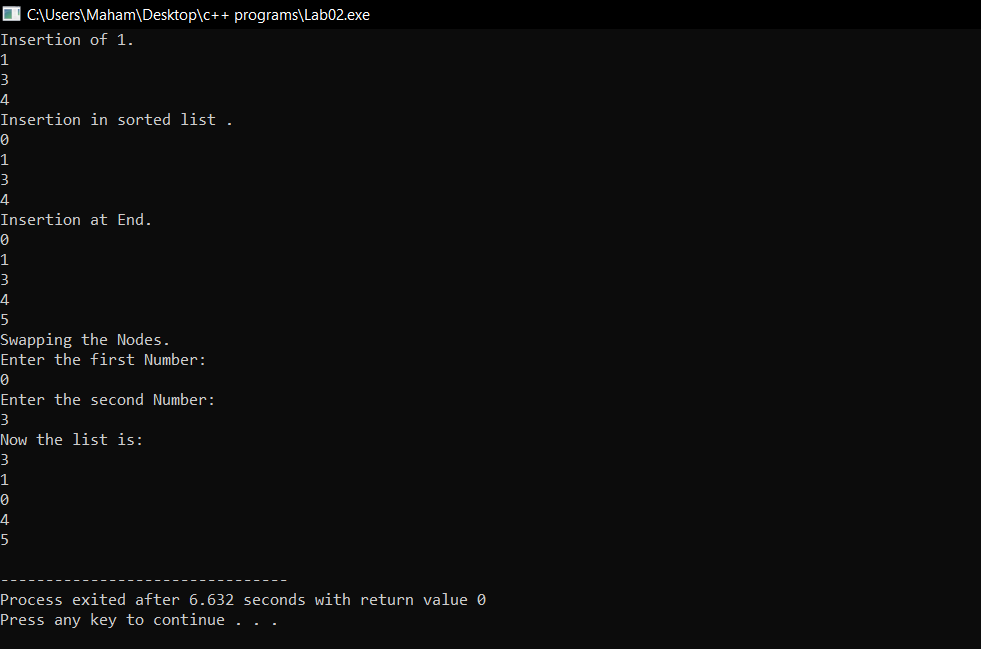
}

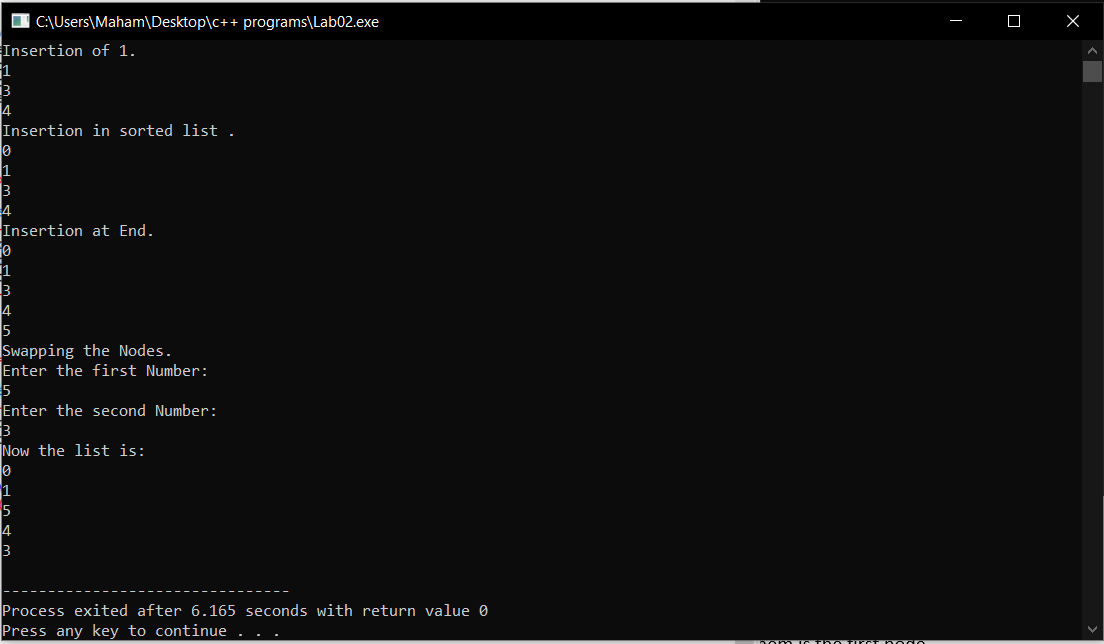
## Output:

* + The two nodes are not adjacent and none of them is the first or last node.

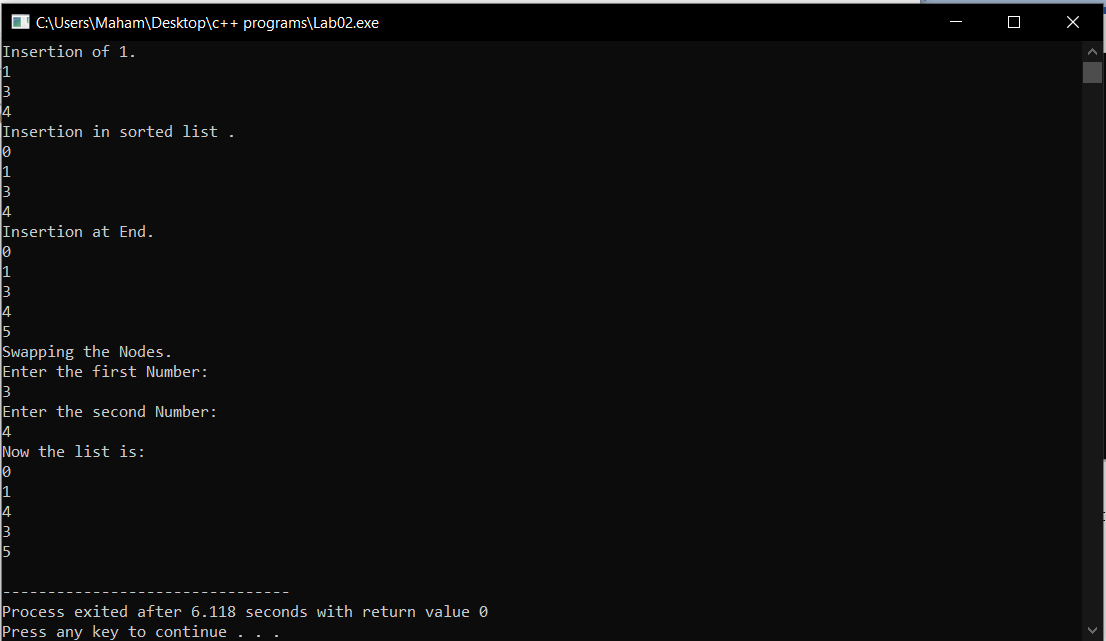


* + The two nodes are not adjacent to each other and one of them is first or last node.

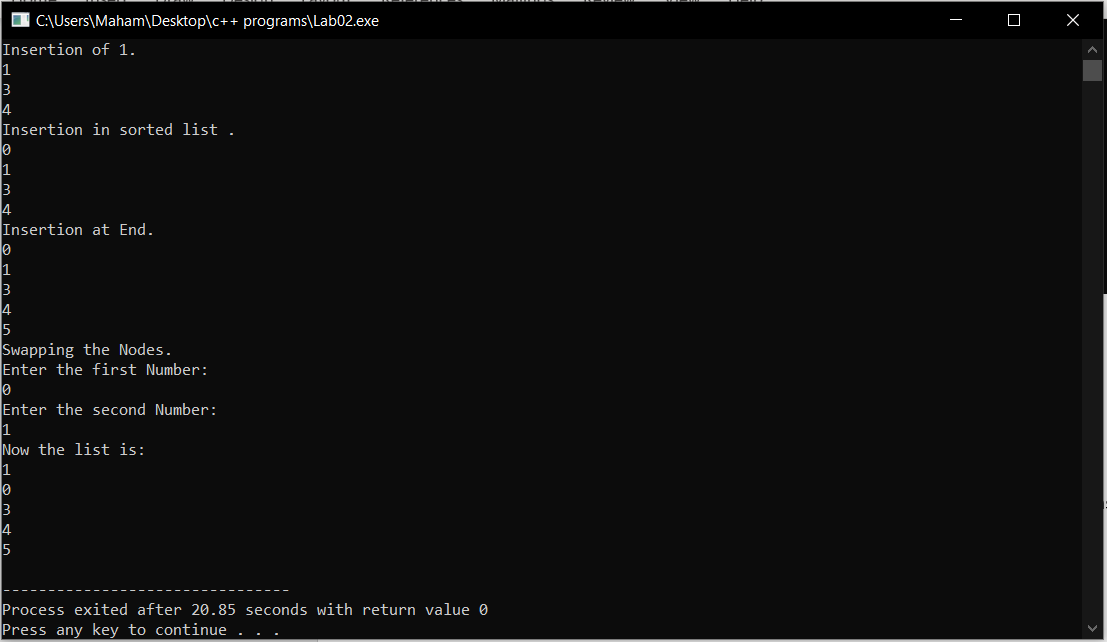




* + Both nodes are adjacent and none of them is the first or last.



* + Both nodes are adjacent and one of them is the first node.



* + Both are adjacent, one is first and the other is last. This is the case in which the length of a list is two.

## code;

int main()

{

//creating the object of the SinglyLinkedList class

SinglyLinkedList \*singlyLinkedList = new SinglyLinkedList();

//checking if the list is empty or not

if (singlyLinkedList -> isEmpty())

{

cout << "List is Empty." << endl;

}

else

{

cout << "List is not Empty." << endl;

}

cout << "Insertion at front." << endl;

cout << "Insertion of 4." << endl;

singlyLinkedList -> insertAtFront(4);

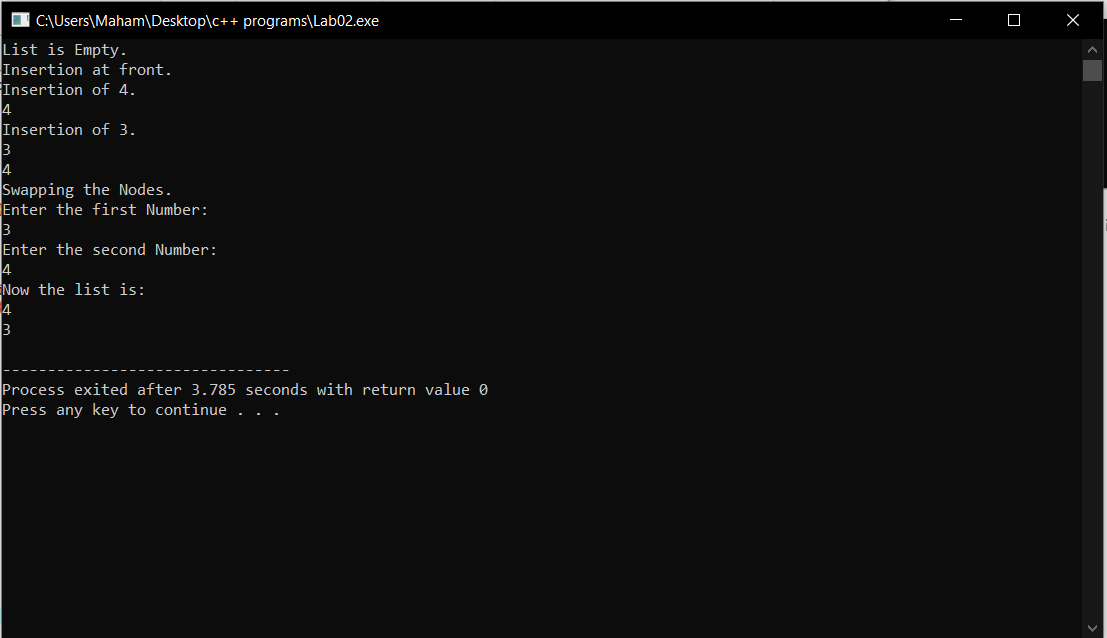
cout << "Insertion of 3." << endl;

singlyLinkedList -> insertAtFront(3);

cout << "Swapping the Nodes." << endl;

singlyLinkedList -> SwappingNodes();

}



If the values are not found:

