# Department of Computing

## **Name : Mahum Samar**

## **CMS ID : 290647**

**CS250: Data Structure and Algorithms**

## **Class: BSCS 9B**

# Lab 3: doubly Linked Lists

# Course Instructor: Dr. Yasir Faheem

## Code:

# include <iostream>

using namespace std;

class ListNode{

public:

int data;

ListNode \*next;

ListNode \*prev;

};

class LinkedList{

public:

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

LinkedList(){

//constructor for doubly linked list.

headNode=NULL;

lastNode=NULL;

preloc=NULL;

loc=NULL;

}

bool isEmpty()

{

//method to check if the list is empty.

return headNode == NULL;

}

void InsertValueAtFront(int value)

{

//method to insert the value at the front of the list

ListNode \*newNode = new ListNode();

newNode->data=value;

if(isEmpty())

{

//if the list is empty

headNode = newNode;

lastNode = newNode;

}

else

{

//if the list is not empty

newNode -> next = headNode;

headNode -> prev = newNode;

headNode = newNode;

}

PrintList(); //displays the list on the screen

}

void InsertValueAtEnd(int value)

{

//method to insert the value at the end of the list

ListNode \*newNode = new ListNode();

newNode->data=value;

if(isEmpty())

{

//if the list is empty

headNode = newNode;

lastNode = newNode;

}

else

{

//if the list is not empty

lastNode -> next = newNode;

newNode -> prev = lastNode;

lastNode = newNode;

}

PrintList();

}

void SearchNode(int value)

{

//method to search the value in the list

loc = headNode;

preloc = NULL;

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

{

preloc = loc;

loc = loc -> next;

}

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

{

//if the value is not found the loc's value is set to NULL.

loc = NULL;

}

}

void InsertSorted(int value)

{

//we assumed that the list is in ascending order and duplication of values is not allowed.

//method to insert the value at its logical position in the list in ascending order.

ListNode \*newNode = new ListNode();

newNode->data=value;

if(isEmpty())

{

//if the list is empty

headNode = newNode;

lastNode = newNode;

}

SearchNode(value);

if(loc != NULL)

{

//if the alue already exists.

cout << "Value already exists." << endl;

return;

}

else

{

if(preloc == NULL)

{

//if value is to be inserted at front.

InsertValueAtFront(value);

}

else

if(preloc == lastNode)

{

//if value is to be inserted at end

InsertValueAtEnd(value);

}

else

{

//if value is to be inserted in the middle.

newNode->next= preloc->next;

newNode->prev= preloc;

preloc->next->prev= newNode;

preloc->next = newNode;

}

}

PrintList();

}

void DeleteFront()

{

//method to delete the headNode.

if(isEmpty())

{

cout << "List is already empty." << endl;

return;

}

else

{

//deletes the headNode

ListNode \*temp = headNode;

headNode = headNode -> next;

headNode -> prev = NULL;

delete temp;

}

PrintList();

}

void DeleteEnd()

{

//method to delete the lastNode.

if(isEmpty())

{

cout << "List is already empty." << endl;

return;

}

else

{

ListNode \*temp = lastNode;

lastNode = lastNode -> prev;

lastNode -> next = NULL;

delete temp;

}

PrintList();

}

void DeleteNode(int value)

{

if(isEmpty())

{

//if list is empty

cout << "List is already empty." << endl;

return;

}

SearchNode(value);

if(loc == NULL)

{

//if value is not found.

cout << "Value does not exists." << endl;

return;

}

else

{

if(preloc == NULL)

{

//is the headNode

if(loc == lastNode)

{

//only one node in list

headNode = NULL;

lastNode = NULL;

}

else

{

//if more then one nodes in the list

ListNode \*temp = headNode;

headNode = headNode -> next;

headNode -> prev = NULL;

delete temp;

}

}

else

if(loc == lastNode)

{

//if lastNode

ListNode \*temp = lastNode;

lastNode = lastNode -> prev;

lastNode -> next = NULL;

delete temp;

}

else

{

//if in the middle of the list, neither headNode nor lastNode.

preloc -> next = loc -> next;

loc -> next -> prev = preloc;

delete loc;

}

}

PrintList();

}

void PrintList()

{

//method ot display the list on the screen.

if(!isEmpty())

{

ListNode \*temp = headNode;

while(temp != NULL)

{

cout << temp -> data << endl;

temp = temp -> next;

}

}

else

{

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

}

}

void DestroyLinkedList()

{

//this method destroys the list both logically and physically.

if(!isEmpty())

{

ListNode \*temp = headNode;

while(temp!= NULL)

{

headNode = headNode -> next;

delete temp;

temp = headNode;

}

cout << "Calling print function." << endl;

PrintList();

}

else

{

cout << "The list is already Empty." << endl;

}

}

/\* Function to reverse a Doubly Linked List \*/

void reverse()

{

ListNode \*temp = NULL;

ListNode \*current = headNode;

/\* swap next and prev for all nodes of

doubly linked list \*/

while (current != NULL)

{

temp = current -> prev;

current -> prev = current -> next;

current -> next = temp;

current = current->prev;

}

/\* Before changing the head, check for the cases like empty

list and list with only one node \*/

if(temp != NULL )

headNode = temp->prev;

PrintList();

}

/\* Function to swap nodes x and y in linked list by

changing links \*/

void swapNodes(int x, int y)

{

// Nothing to do if x and y are same

if (x == y)

{

return;

}

// Search for x (keep track of prelocX and locX

ListNode \*prelocX = NULL, \*locX = headNode;

while (locX && locX->data != x)

{

prelocX = locX;

locX = locX->next;

}

ListNode \*prelocY = NULL, \*locY = headNode;

while(locY && locY -> data != y)

{

prelocY = locY;

locY = locY -> next;

}

// If either x or y is not present, nothing to do

if (locX == NULL || locY == NULL)

{

return;

}

// If x is not head of linked list

if (prelocX != NULL)

{

prelocX->next = locY;

}

else // Else make y as new head

{

headNode = locY;

}

// If y is not head of linked list

if (prelocY != NULL)

{

prelocY->next = locX;

}

else // Else make x as new head

{

headNode = locX;

}

// code to Swap next pointers of x and y

ListNode \*temp = locY -> next;

locY -> next = locX -> next;

locX -> next = temp;

cout << "List after swapping the nodes: " << endl;

PrintList();

}

// Rearranges given linked list such that all odd

// positioned nodes are before even positioned.

// Returns new head of linked List.

void rearrangeEvenOdd()

{

// Corner case

if (headNode == NULL)

{

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

return;

}

// Initialize first nodes of even and

// odd lists

ListNode \*oddNode = headNode;

ListNode \*evenNode = headNode -> next;

// node to store the address of the headNode of the evenNode list

//so that it can be attached at the end of the oddNode list

ListNode \*evenHeadNode = evenNode;

while (1)

{

// If there are no more nodes, then connect

// first node of evenNode list to the last node

// of oddNode list

if (!oddNode || !evenNode || !(evenNode->next))

{

oddNode->next = evenHeadNode;

break;

}

// list update for oddNode nodes

oddNode -> next = evenNode -> next;

oddNode = evenNode -> next;

// If there are NO more evenNode nodes after

// current oddNode.

if (oddNode->next == NULL)

{

evenNode->next = NULL;

oddNode->next = evenHeadNode;

break;

}

// list update for evenNode nodes

evenNode -> next = oddNode -> next;

evenNode = oddNode -> next;

}

PrintList();

}

};

# Main function for each method

InsertValueAtFront( )

int main()

{

LinkedList \*linkedList = new LinkedList();

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

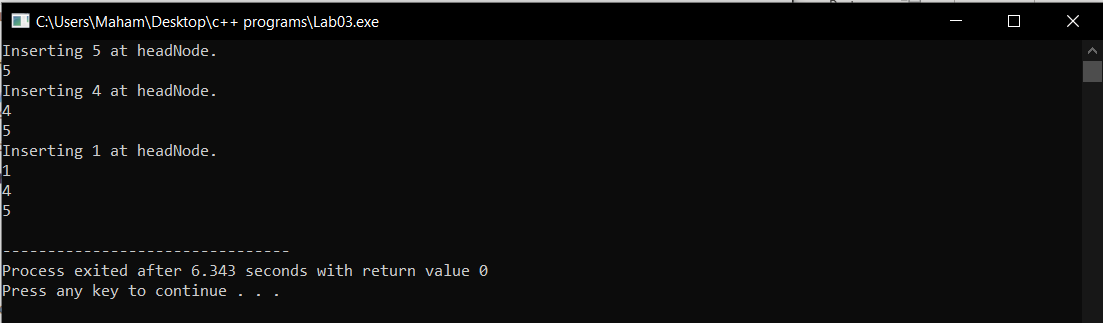
linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

}

### Output:



InsertValueAtEnd( )

int main()

{

LinkedList \*linkedList = new LinkedList();

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

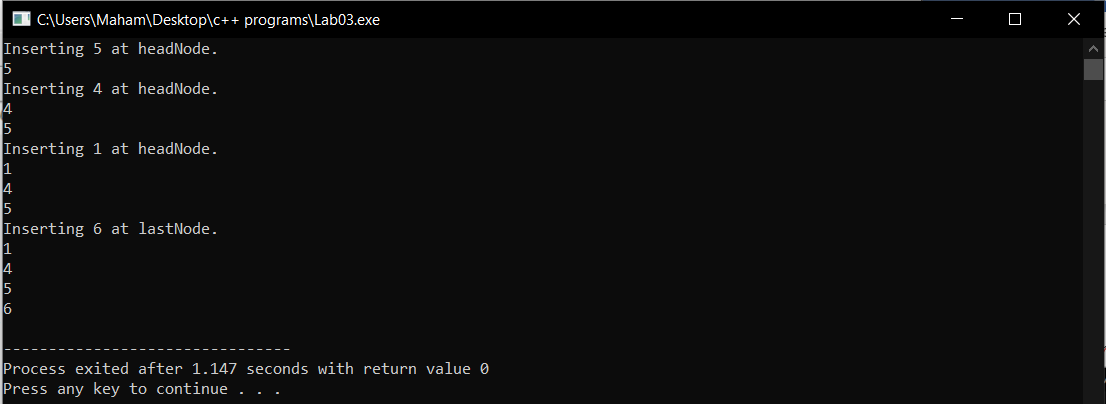
linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

}

### Output:



InsertSorted( )

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

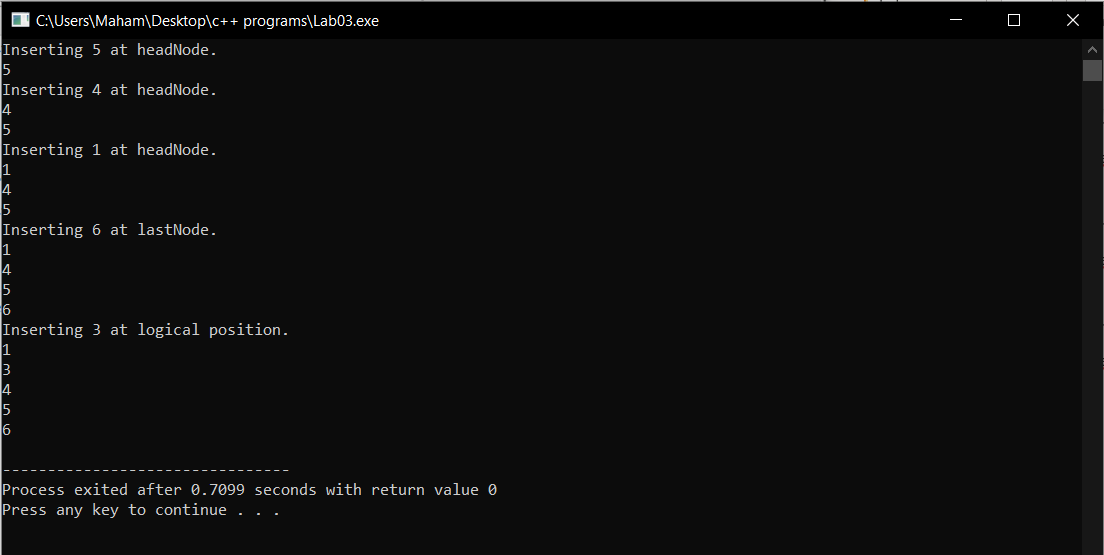
linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

linkedList -> InsertSorted(3);

}

### Output:



DeleteFront( )

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

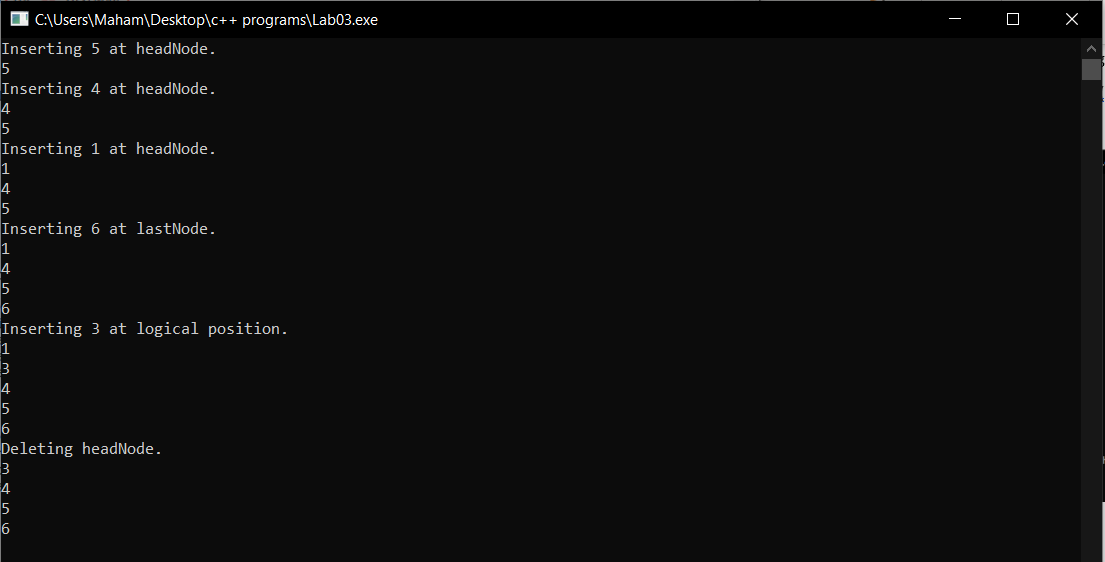
linkedList -> InsertSorted(3);

cout << "Deleting headNode." << endl;

linkedList -> DeleteFront();

}

### Output:



DeleteEnd()

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

linkedList -> InsertSorted(3);

cout << "Deleting headNode." << endl;

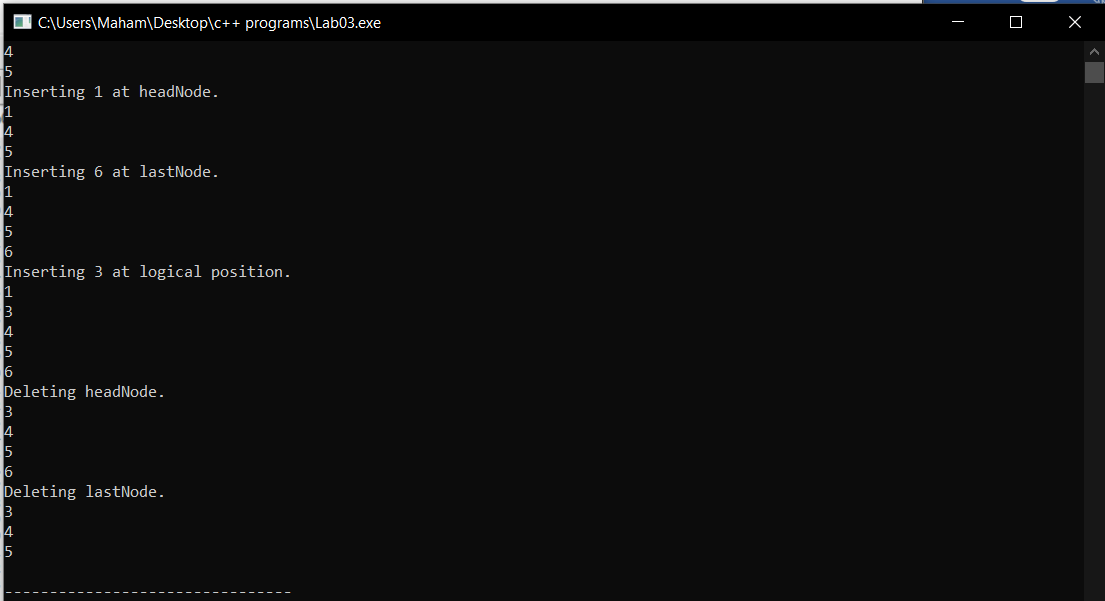
linkedList -> DeleteFront();

cout << "Deleting lastNode." << endl;

linkedList -> DeleteEnd();

}

### Output:



DeleteNode()

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

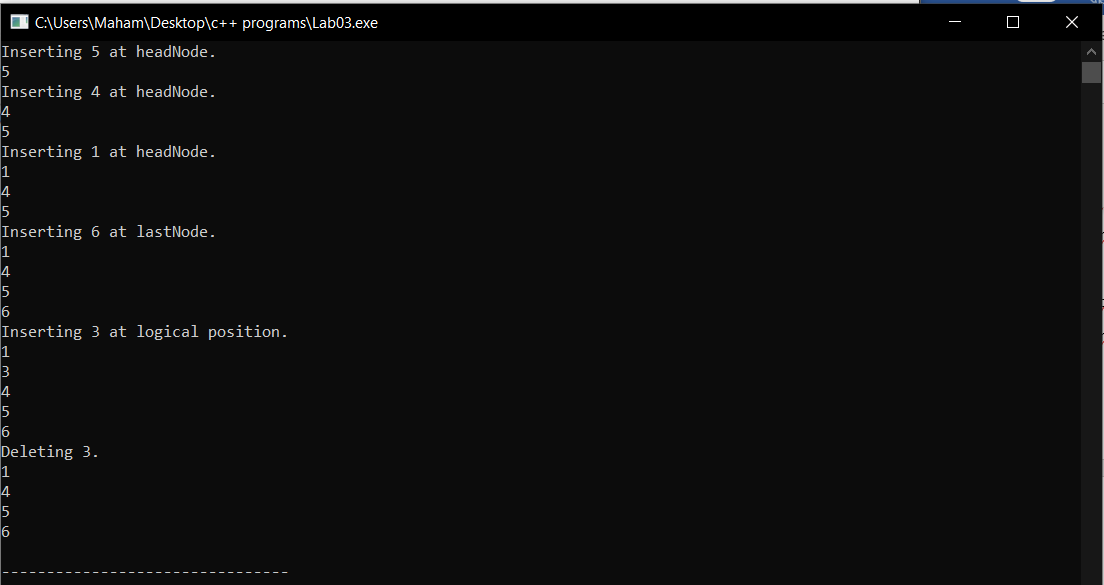
linkedList -> InsertSorted(3);

cout << "Deleting 3." << endl;

linkedList -> DeleteNode(3);

}

### Output:



DestroyLinkedList( )

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

linkedList -> InsertSorted(3);

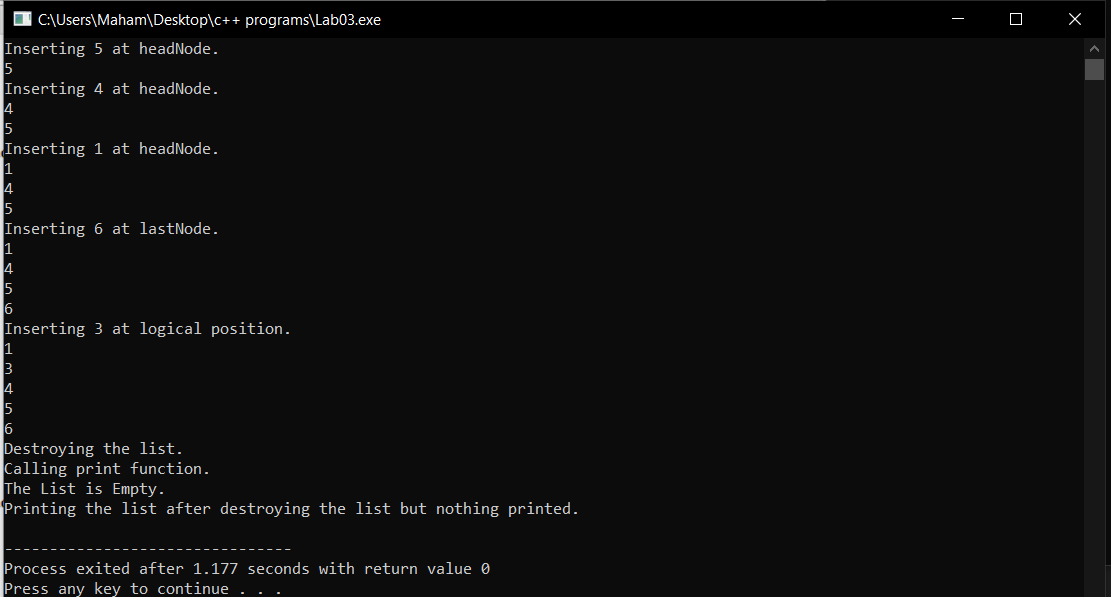
cout << "Destroying the list." << endl;

linkedList -> DestroyLinkedList();

cout << "Printing the list after destroying the list but nothing printed." << endl;

}

### Output:



Reverse( )

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

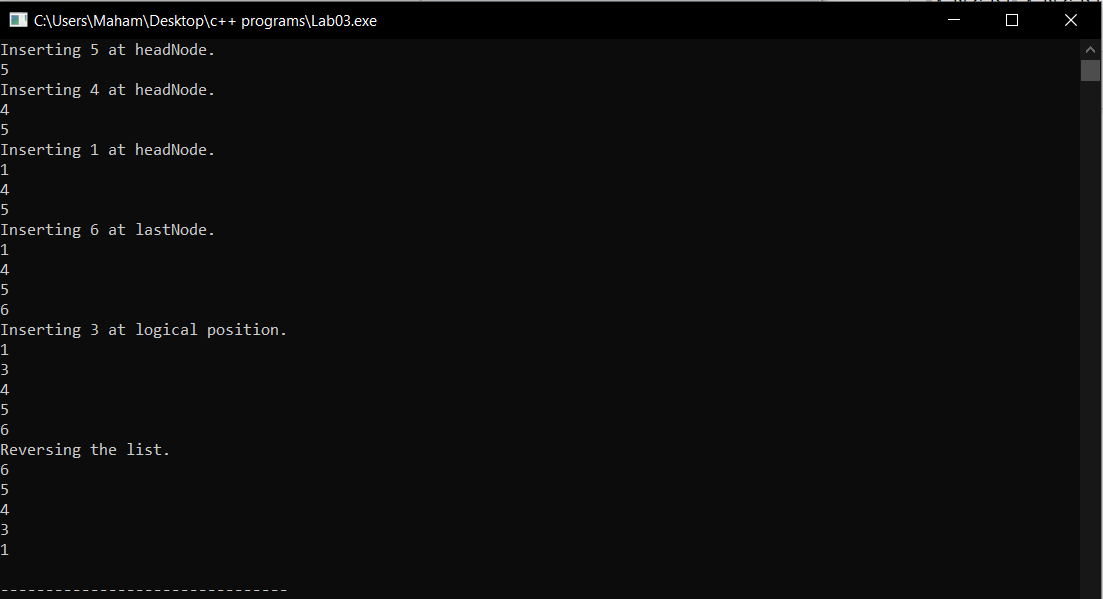
linkedList -> InsertSorted(3);

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

linkedList -> reverse();

}

### Output:



swapNodes( )

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

linkedList -> InsertSorted(3);

int x = 0 ,y = 0;

cout << "Enter the 1st number : " << endl;

cin >> x;

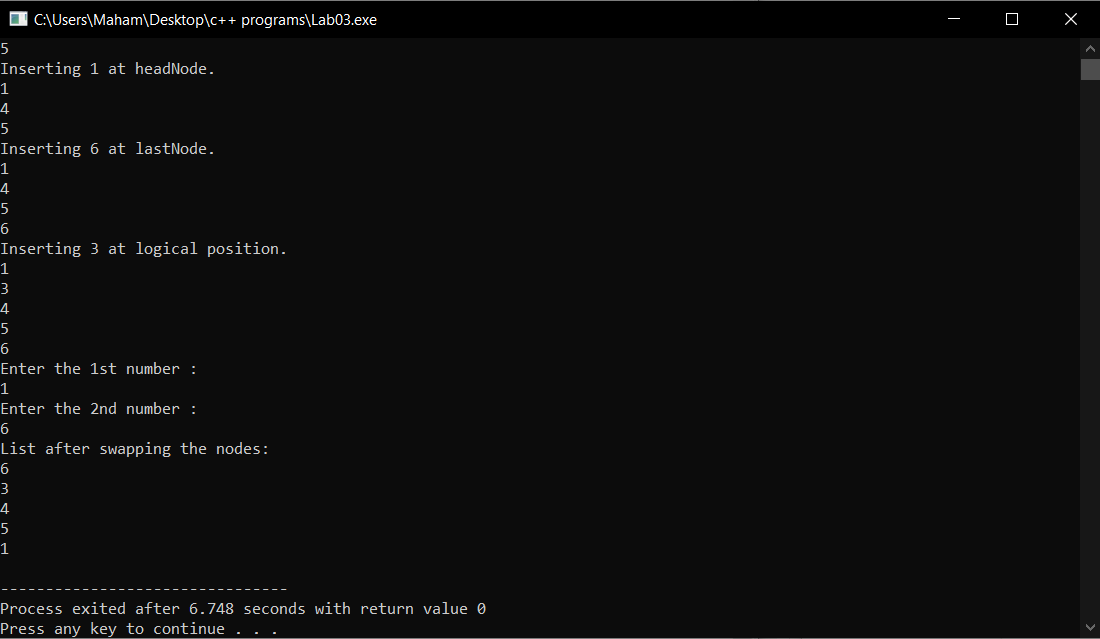
cout << "Enter the 2nd number : " << endl;

cin >> y;

linkedList -> swapNodes(x,y);

}

### Output:



rearrangeEvenOdd( )

int main()

{

LinkedList \*linkedList = new LinkedList();

//cout << linkedList -> isEmpty() << endl ;

cout << "Inserting 5 at headNode." << endl;

linkedList -> InsertValueAtFront(5);

cout << "Inserting 4 at headNode." << endl;

linkedList -> InsertValueAtFront(4);

cout << "Inserting 1 at headNode." << endl;

linkedList -> InsertValueAtFront(1);

cout << "Inserting 6 at lastNode." << endl;

linkedList -> InsertValueAtEnd(6);

cout << "Inserting 3 at logical position." << endl;

linkedList -> InsertSorted(3);

cout << "Rearrange even and odd nodes." << endl;

linkedList -> rearrangeEvenOdd();

}

### Output:

