**Họ và Tên: Trần Thị Ngọc Diệp**

**MSSV: 1827005**

**Lớp: B2HK182 – Cấu Trúc Dữ Liệu và Giải Thuật**

**(TUT 3 & LAB 3)**

------------------------

**TUT 3**

**Question 1:**

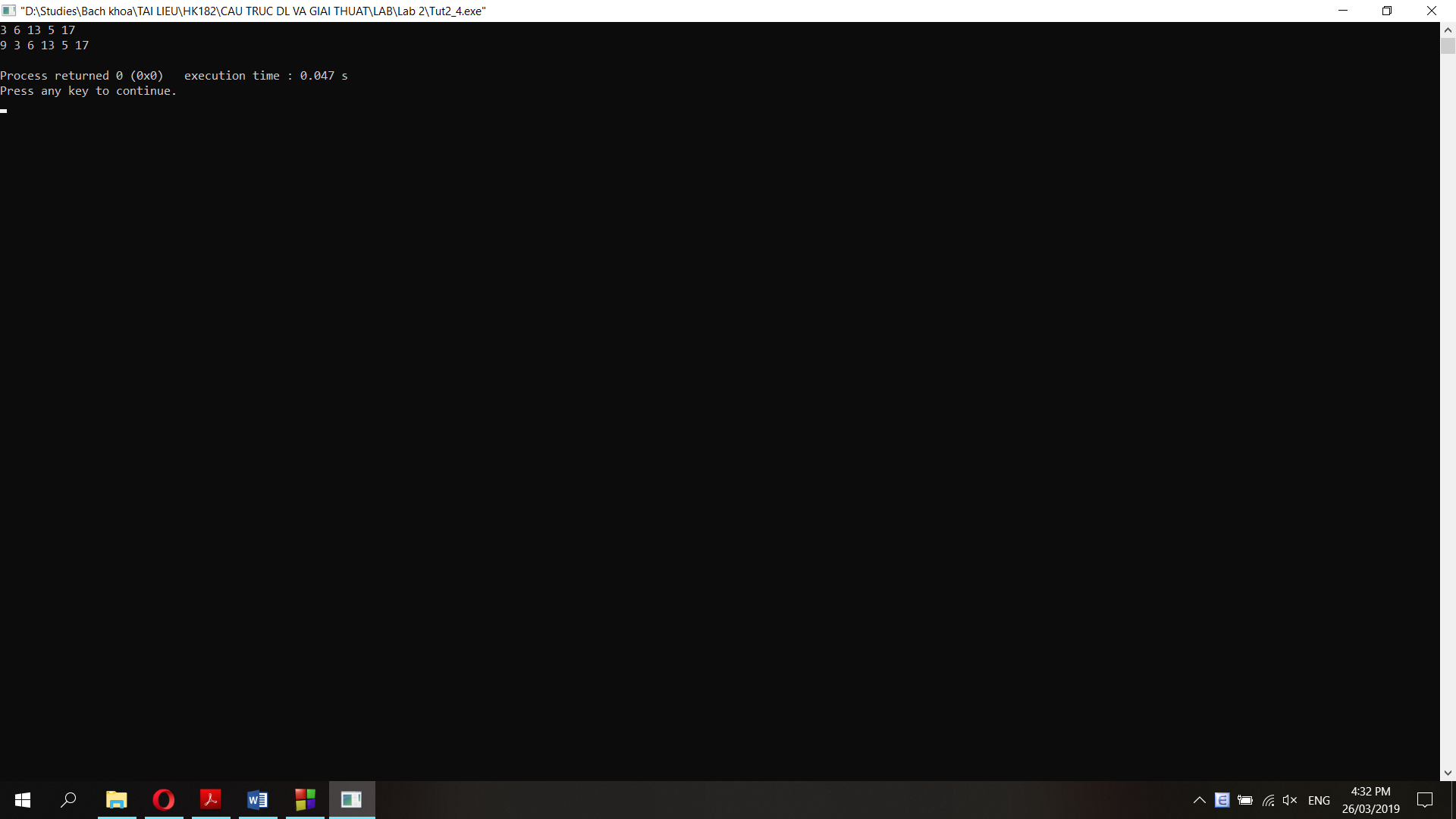
**a. Insert 1 node (data = 9) at the beginning:**

pNew = new Node;

pNew->data = 9;

pNew->next = pHead;

pHead = pNew;



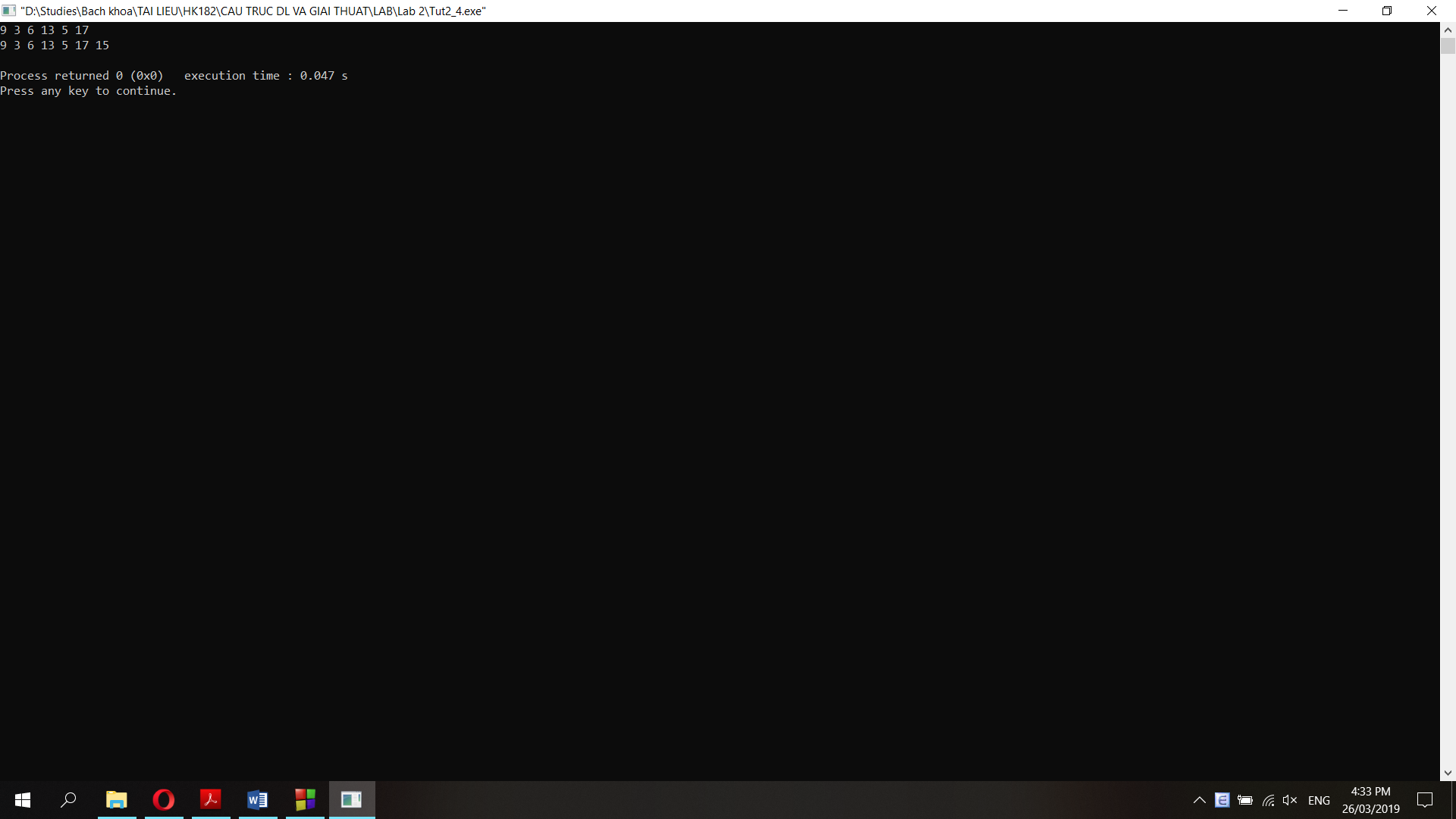
**b. Insert 1 node (data = 15) at the end:**

pNew = new Node;

pNew->data = 15;

pTail->next = pNew;

pTail = pNew;



**c. Delete the node which has the data = 17:**

Node \*pTemp2;

pNew = pHead;

while (pNew->data != 17){

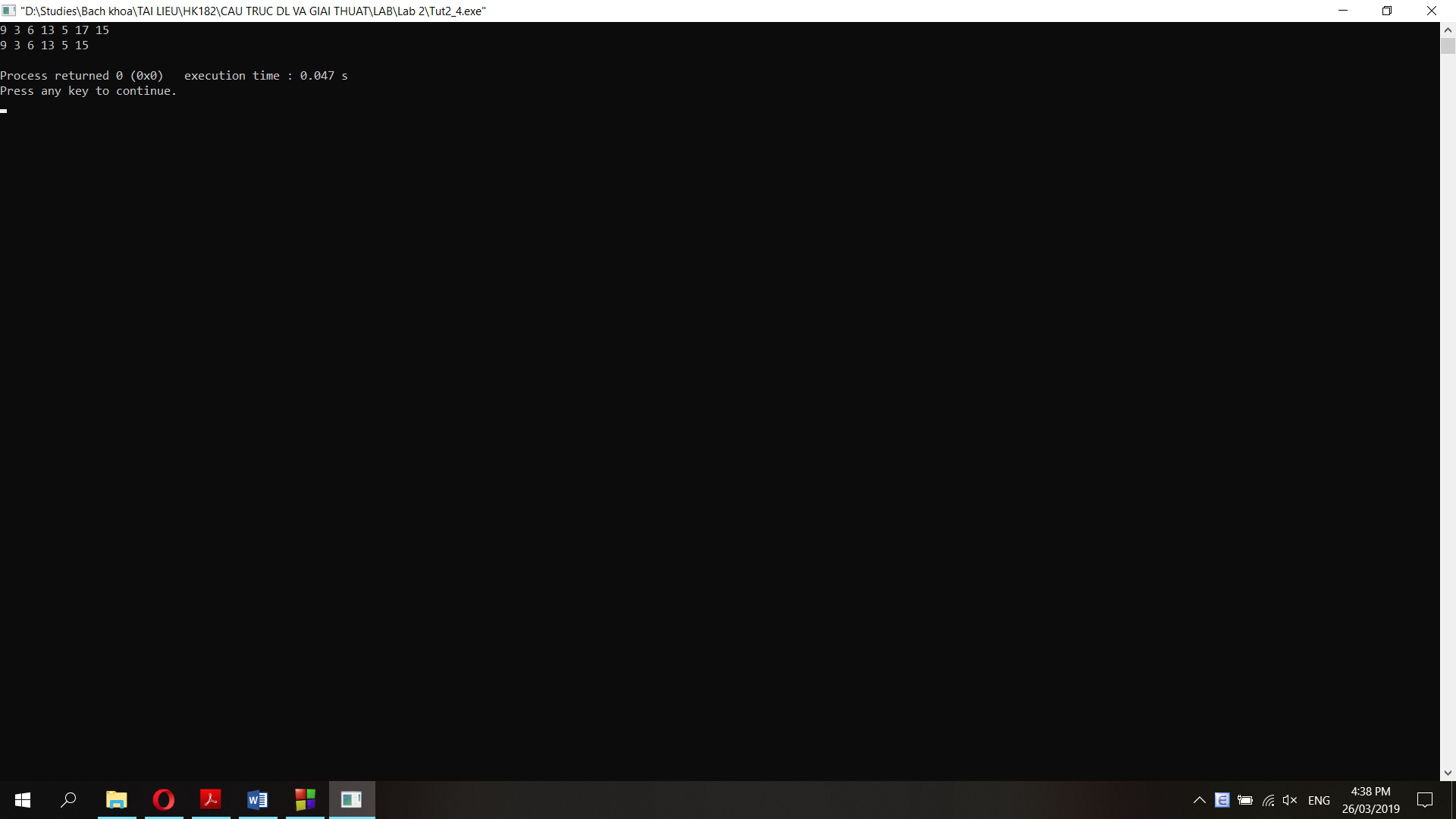
pTemp2 = pNew;

pNew = pNew->next;

}

pTemp2->next = pNew->next;

delete(pNew);



**f. Delete the node which pTemp points to:**

pTemp2 = pHead;

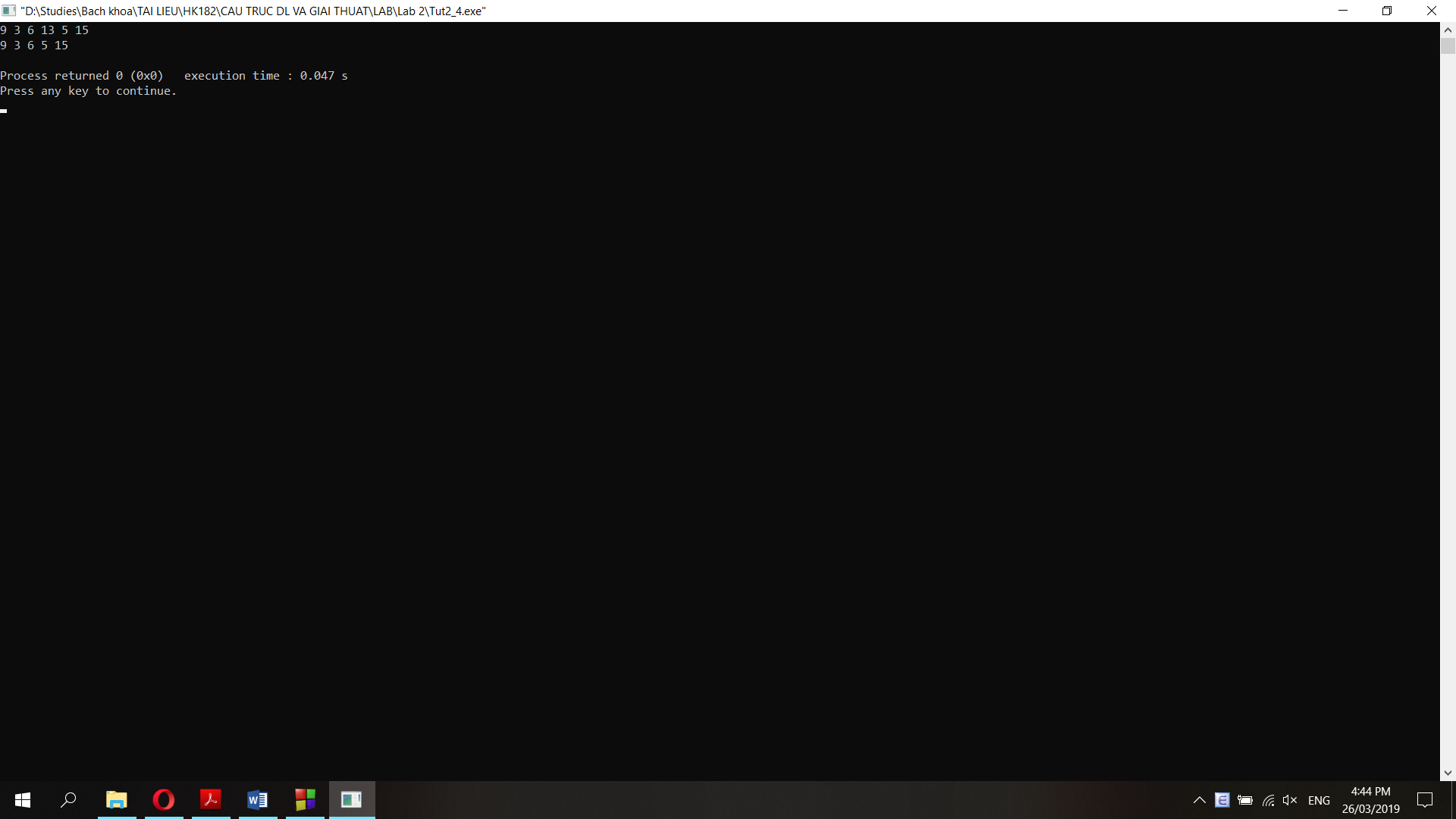
while (pTemp2->next != pTemp){

pTemp2 = pTemp2->next;

}

pTemp2->next = pTemp->next;

delete(pTemp);



**Question 2:**

**a. Function prints a single linked list (pTail->next = NULL):**

void printSingleLinkedList(node\* pHead){

node \*pTemp;

pTemp = pHead;

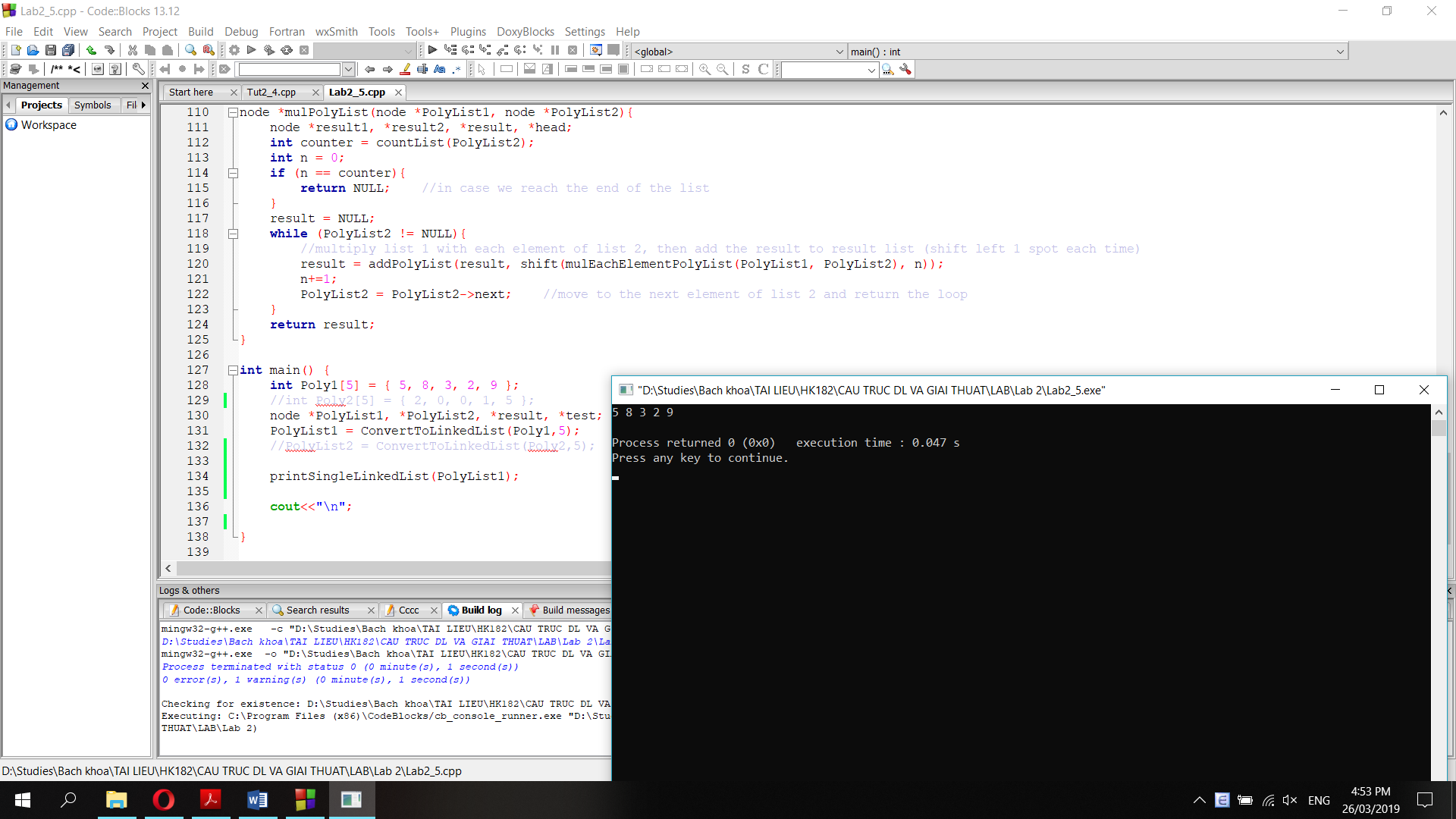
while (pTemp != NULL){

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

}

}



**b. Function prints a circular linked list (pTail->next = pHead->next)**

void printCircularLinkedList(node\* pHead){

node \*pTemp;

pTemp = pHead;

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

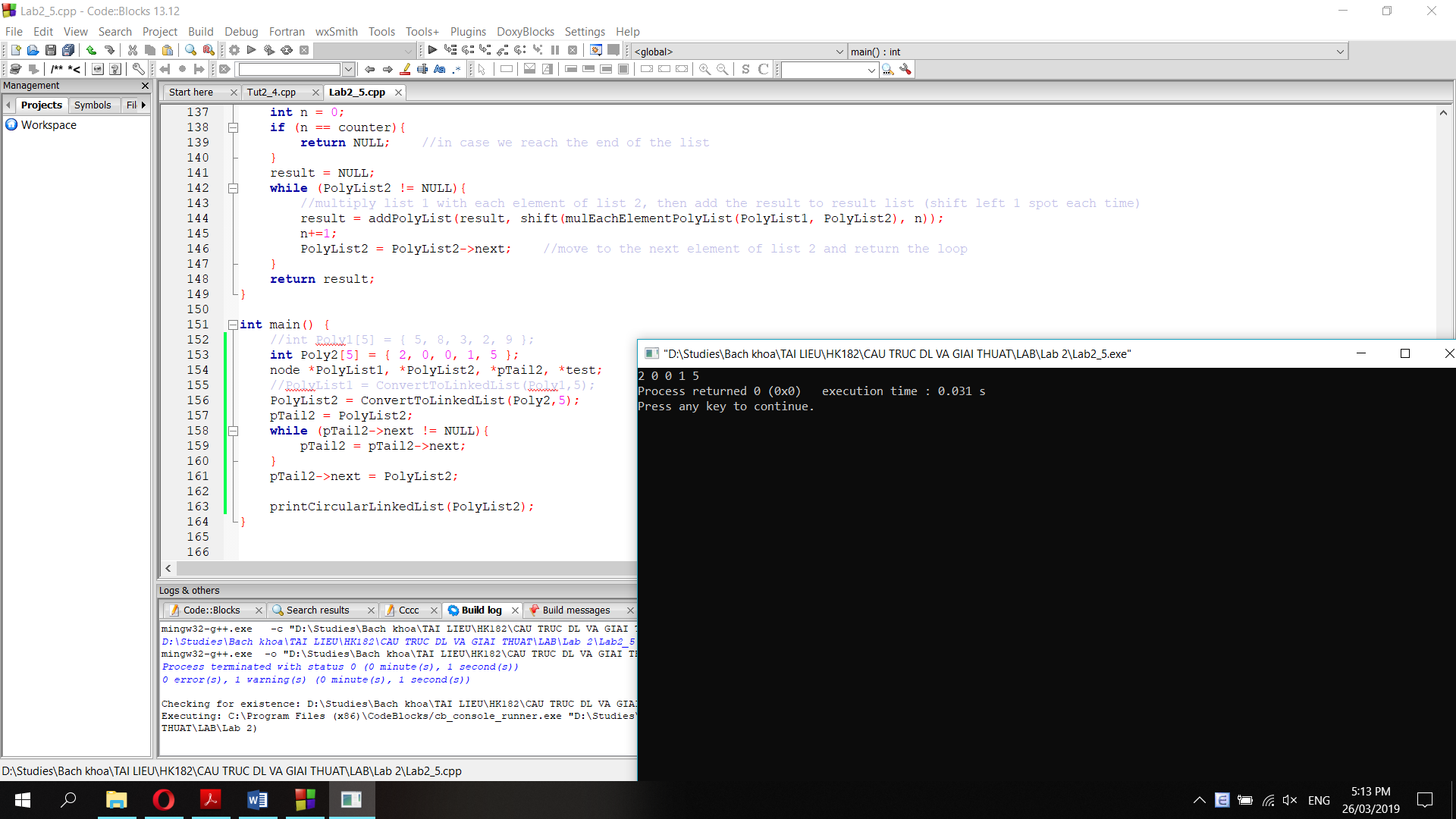
while (pTemp != pHead){

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

}

}



**c. Function prints both single linked list and circular linked list:**

void printList(node\* pHead){

node \*pTemp;

pTemp = pHead;

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

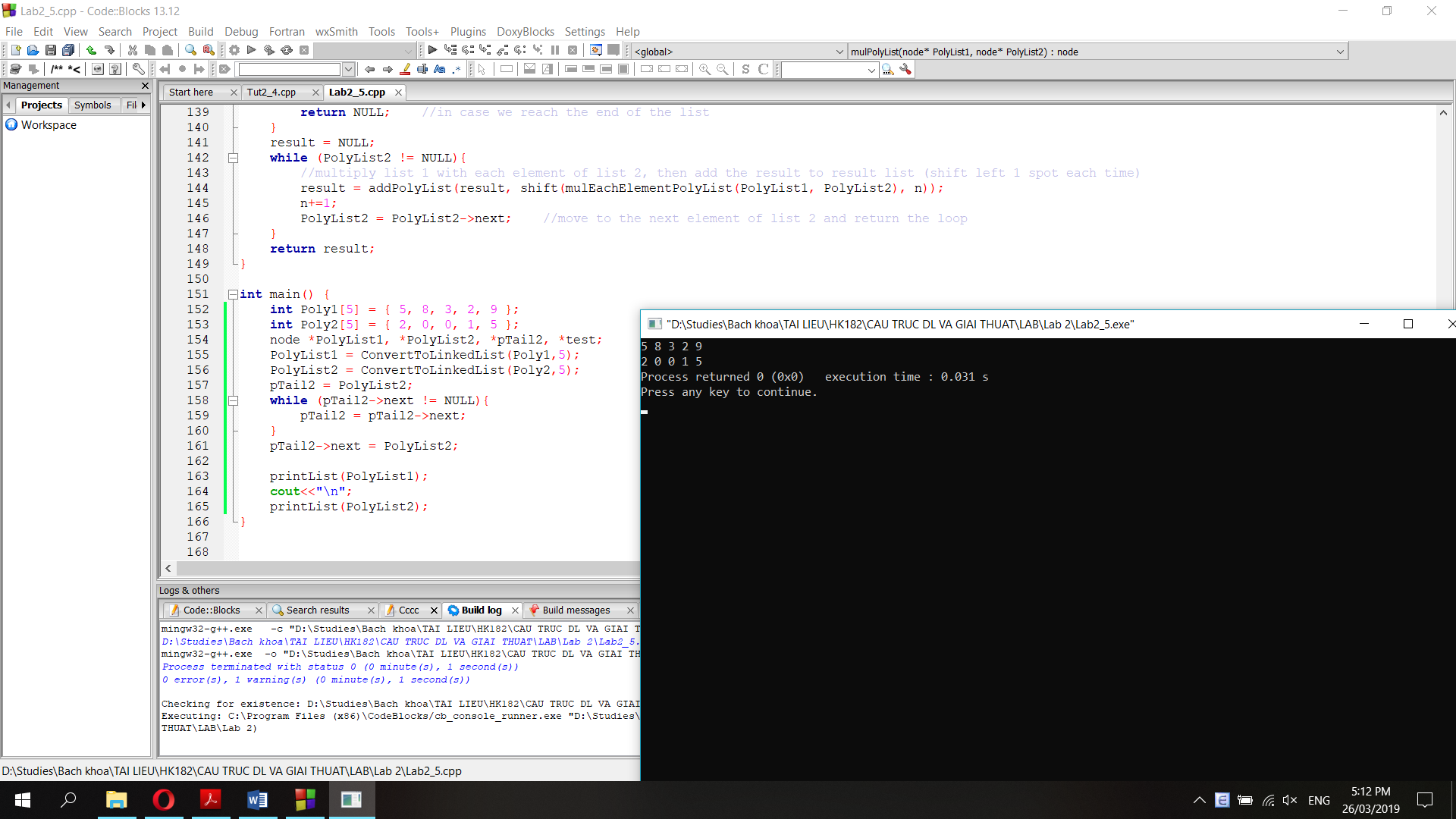
while (pTemp != NULL && pTemp != pHead){

cout<<pTemp->data<<" ";

pTemp = pTemp->next;

}

}



**Question 3: Function searches a node of a single linked list**

node\* searchList(node\* pHead, int data){

node \*result;

result = pHead;

while (result->data != data && result != NULL){

result = result->next;

}

if (result->data = data){

return result;

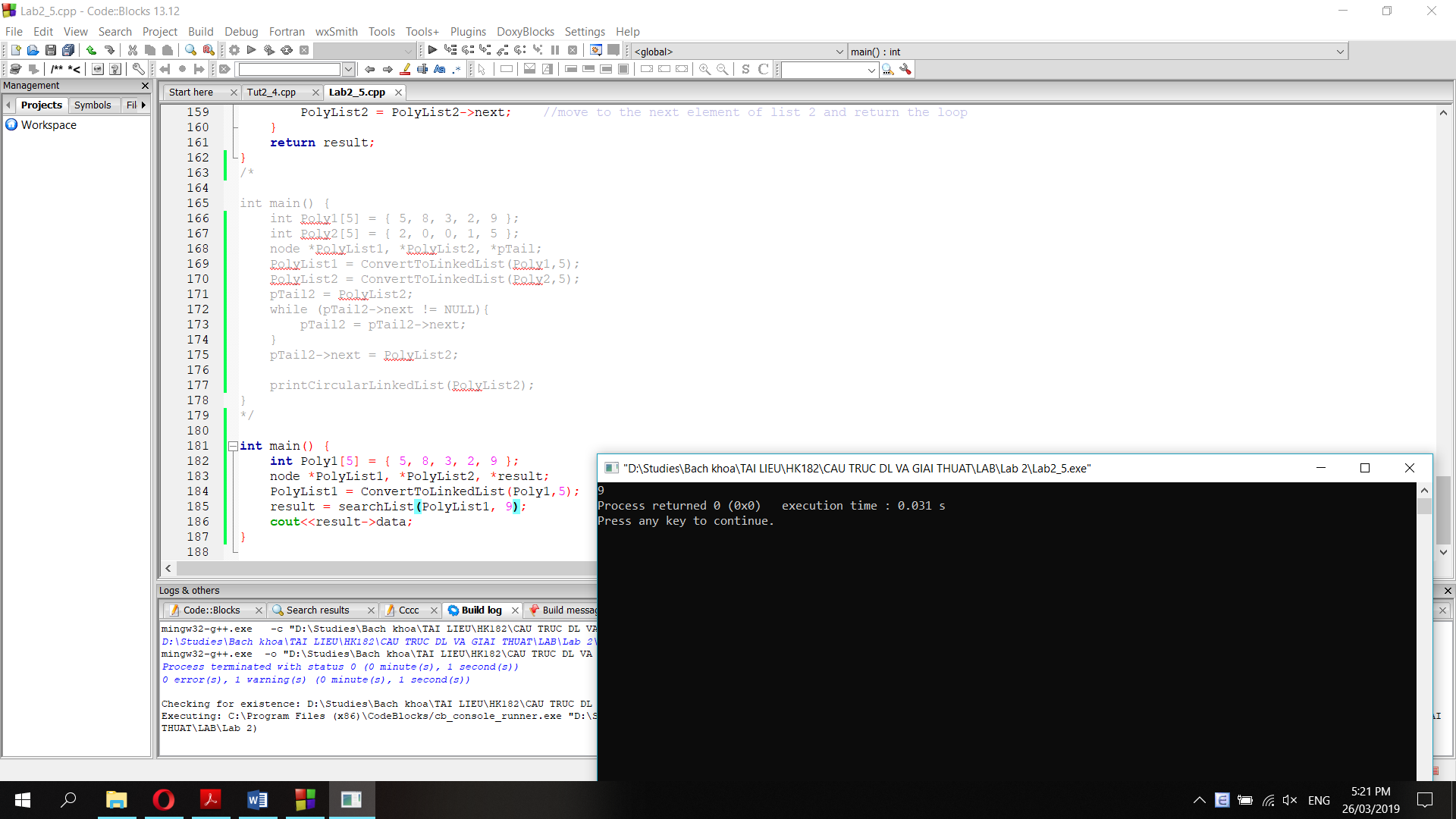
}

else {

return NULL;

}

}



**Question 4: Function delete the Nth node.**

//count the number of nodes in a list

int countList(node \*head){

int counter=1;

while (head->next != NULL){

counter += 1;

head = head->next;

}

return counter;

}

//delete the Nth node

node\* deleteNth(node\* head, int n){

node \*result;

int counter = countList(head);

if (n==0){ //if n=0 delete the head and return a new list

result = head->next;

delete(head);

return result;

}

else if (n<0 || n>counter){ //if n<0 or greater than the length of the list, do nothing

return head;

}

else {

int counter2 = 0;

result = head;

node \*temp;

while (counter2 != n-1){ //point result to the n-1 node

result = result->next;

counter2 += 1;

}

temp = result;

result = result->next;

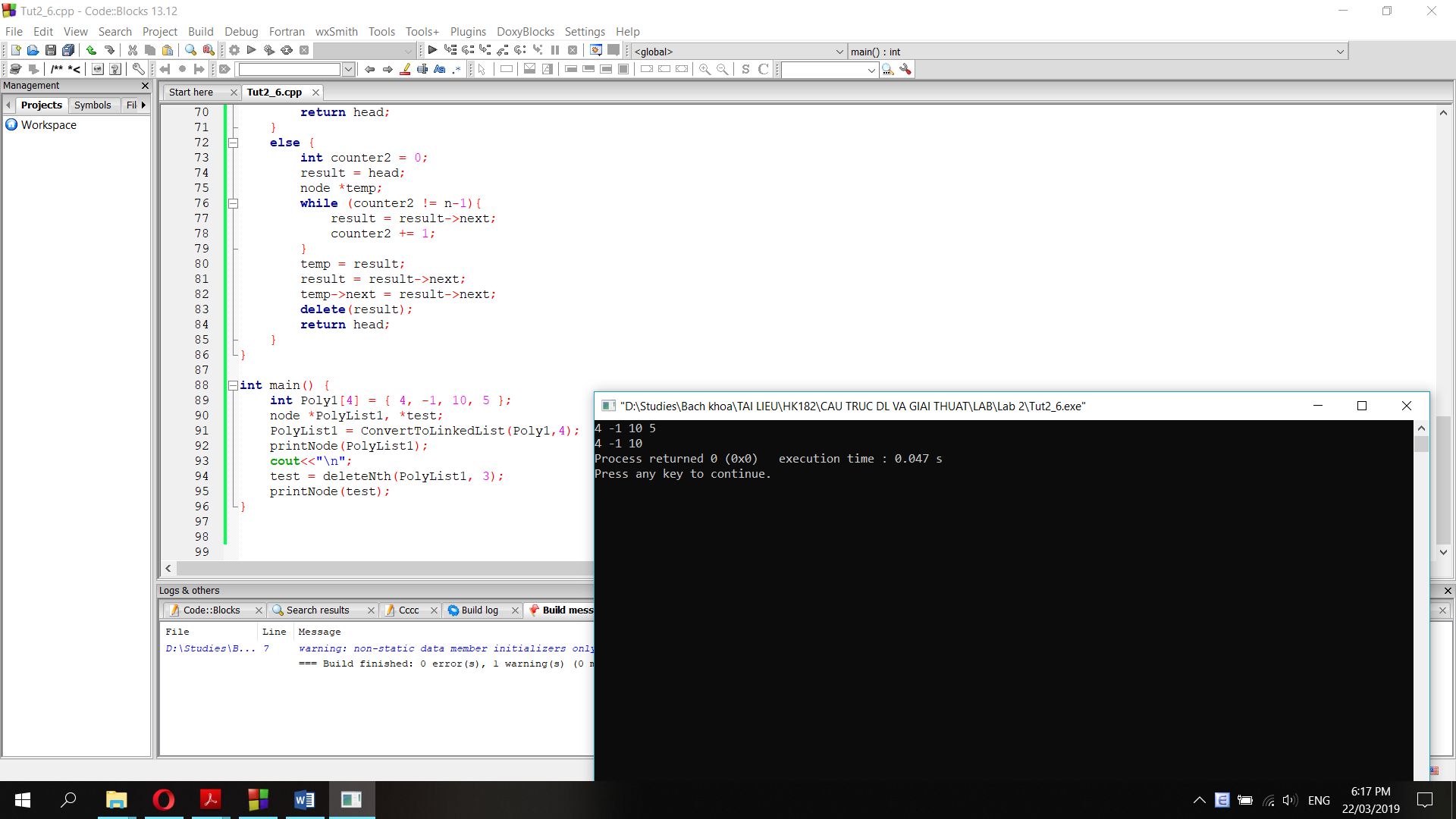
temp->next = result->next;

delete(result);

return head;

}

}



**Question 5:**

void func1(node\* head) {

node\* temp = head;

while (temp != NULL) {

if (temp->next == NULL) {

temp->next = head;

return;

}

temp = temp->next;

}

}

**Nếu truyền con trỏ head của một single linked list vào hàm func1() trên, hàm sẽ chuyển single linked list thành circular linked list.**

**Giải thích: hàm func1() duyệt con trỏ temp từ nút head cho đến nút cuối cùng, sau đấy cho temp->next = head, nghĩa là tail->next = head.**

**LAB 3**

**The full program:**

#include<iostream>

#include<sstream>

#include<stdbool.h>

using namespace std;

class Node {

public:

int data;

Node\* next;

};

//interface part

class List {

private:

int count;

Node\* pHead;

public:

List();

void addFirst(int); //add a new node at the beginning with the data as input

void addLast(int newdata); //add a new node at the end with the data as input

void display(); //print all data of a linked list

void printPoly(); //print the linked list as a polynomial

void addConstant(int nConst); //add a constant to the polynomial

List \*addPolyList(List \*Poly); //add 2 linked list and return a new list

void reverseList(); //reverse a linked list using stack method

void reverseList2(); //reverse a linked list using queue method

Node\* getFirst(); //get the first node out of the linked list

Node\* getLast(); //get the last node out of the linked list

void addFirstNode(Node \*node); //add a node to the beginning of the linked list

void addLastNode(Node \*node); //add a node to the end of the linked list

List \*getIntersection(List \*newList); //find the intersection of 2 lists and return a new list

List \*getUnion(List \*newList); //find the union of 2 lists and return a new list

void append(List &newList); //append a new list to the current list

~List();

};

//implementation part

List::List() {

count = 0;

pHead = NULL;

}

void List::addFirst(int newdata) {

Node\* pTemp = new Node;

pTemp->data = newdata;

pTemp->next = pHead;

pHead = pTemp;

count++;

}

void List::addLast(int newdata){

Node \*pTemp = pHead;

//if there's no element in the list, return the head with new data

if (pTemp == NULL){

this->addFirst(newdata);

}

//else point pTemp to the last element and add another node after it

else {

while (pTemp->next != NULL){

pTemp = pTemp->next;

}

pTemp->next = new Node;

pTemp->next->data = newdata;

pTemp->next->next = NULL;

count++;

}

}

void List::display() {

Node\* pTemp = pHead;

if (pTemp == NULL){

cout<<"List is empty.";

}

else{

while (pTemp!=NULL) {

cout << pTemp->data << " ";

pTemp = pTemp->next;

}

}

}

List::~List() {

Node\* pTemp = pHead;

while (pTemp!=NULL) {

pTemp = pTemp->next;

delete pHead;

pHead = pTemp;

}

}

void List::addConstant(int nConst) {

pHead->data += nConst;

return;

}

//add 2 polynomial lists, return a new list

List\* List::addPolyList(List \*Poly) {

List \*result;

result = new List;

Node \*pTemp1 = this->pHead;

Node \*pTemp2 = Poly->pHead;

int n = 0;

//until we reach the end of both lists

while (pTemp1 != NULL || pTemp2 != NULL){

//if we reach the end of list 1, add data of list 2 to result

if (pTemp1 == NULL && pTemp2 != NULL){

result->addLast(pTemp2->data);

pTemp2 = pTemp2->next;

}

//if we reach the end of list 2, add data of list 1 to result

else if (pTemp1 != NULL && pTemp2 == NULL){

result->addLast(pTemp1->data);

pTemp1 = pTemp1->next;

}

//else: add data of both list and add to result

else{

result->addLast(pTemp1->data + pTemp2->data);

pTemp1 = pTemp1->next; //move ptemp to next node

pTemp2 = pTemp2->next;

}

n++;

}

result->count = n;

return result;

}

//print a list using array

void List::printPoly(){

int n = this->count-1;

int \*arr = new int [n];

Node \*pTemp = pHead;

int i=0;

while (pTemp != NULL){

\*(arr+i) = pTemp->data;

pTemp = pTemp->next;

i++;

}

for (i=0; i<n; i++){

if (\*(arr+n-i)==0){

continue;

}

else if (\*(arr+n-i)==1){

cout<<"x^"<<n-i<<" + ";

}

else{

cout<<\*(arr+n-i)<<"x^"<<n-i<<" + ";

}

}

cout<<\*arr;

}

//get the first node out of the list

Node\* List::getFirst(){

Node \*pTemp = this->pHead;

if (this->pHead == NULL){

return this->pHead;

}

else if (this->pHead->next == NULL){

pHead = NULL;

this->count--;

return pTemp;

}

else {

this->pHead = this->pHead->next;

this->count--;

return pTemp;

}

}

//get the last node out of the list

Node\* List::getLast(){

Node \*pTemp = this->pHead;

if (this->pHead == NULL){

return this->pHead;

}

else if (this->pHead->next == NULL){

pHead = NULL;

this->count--;

return pTemp;

}

else {

while(pTemp->next->next!=NULL){

pTemp = pTemp->next;

}

Node \*pTemp2 = pTemp->next;

pTemp->next = NULL;

this->count--;

return pTemp2;

}

}

//add a node before the head

void List::addFirstNode(Node \*node){

if (this->pHead == NULL){

pHead = node;

pHead->next = NULL;

}

else if (this->pHead->next == NULL){

Node \*pTemp = pHead;

pHead = node;

pHead->next = pTemp;

pTemp->next = NULL;

}

else{

Node \*pTemp = pHead;

pHead = node;

pHead->next = pTemp;

}

this->count++;

}

//add a node at the end of the list

void List::addLastNode(Node \*node){

if (this->pHead == NULL){

this->pHead = node;

this->pHead->next = NULL;

}

else if (this->pHead->next == NULL){

this->pHead->next = node;

node->next = NULL;

}

else{

Node \*pTemp = this->pHead;

while (pTemp->next != NULL){

pTemp = pTemp->next;

}

pTemp->next = node;

node->next = NULL;

}

count++;

}

//reverse a list using Stack method

void List::reverseList(){

Node \*pTemp = this->pHead;

List sTemp;

int n = this->count;

while(n!=0){

//List: 1->2->3->4 => Stack: 4->3->2->1

pTemp = pTemp->next;

sTemp.addFirstNode(this->getFirst());

n--;

}

n = sTemp.count;

pTemp = sTemp.pHead;

while(n!=0){

//Stack: 4->3->2->1 => List: 4->3->2->1

pTemp = pTemp->next;

this->addLastNode(sTemp.getFirst());

n--;

}

}

//reverse a list using queue method

void List::reverseList2(){

List qTemp;

int n = this->count;

while(n!=0){

//List: 1->2->3->4 => Queue: 4->3->2->1

qTemp.addFirstNode(this->getFirst());

n--;

}

n = qTemp.count;

while(n!=0){

//Queue: 4->3->2->1 => List: 4->3->2->1

this->addFirstNode(qTemp.getLast());

n--;

}

}

//Append newList to the current list

void List::append(List &newList){

int n = newList.count;

while (n>0){

this->addLastNode(newList.getFirst());

n--;

}

}

//find the intersection of 2 lists and return a new list

List\* List::getIntersection(List \*newList){

Node \*pTemp1 = this->pHead;

Node \*pTemp2;

List \*result;

result = new List;

while (pTemp1 != NULL){

//point pTemp from pHead to tail of the current list

pTemp2 = newList->pHead;

while (pTemp2 != NULL){

//point pTemp2 from pHead to tail of the new list

if (pTemp2->data == pTemp1->data){

result->addLast(pTemp1->data); //if 2 nodes have the same data, add it to result

}

pTemp2 = pTemp2->next;

}

pTemp1 = pTemp1->next;

}

return result;

}

//find the union of 2 lists and return a new list

List\* List::getUnion(List \*newList){

Node \*pTemp1 = this->pHead;

Node \*pTemp2;

List \*result;

result = new List;

bool flag = true; //flag is true when a node data is not in result yet

//add all data of the current list to result first

pTemp2 = this->pHead;

result->addFirst(pTemp2->data);

while (pTemp2 != NULL){

pTemp1 = result->pHead;

while (pTemp1 != NULL){

flag = true;

//if a node is in result already, break and move to next node

if (pTemp2->data == pTemp1->data){

flag = false;

break;

}

pTemp1 = pTemp1->next;

}

//if flag is still true which means this node data is not in result yet, add it to result

if (flag == true){

result->addFirst(pTemp2->data);

}

pTemp2 = pTemp2->next;

}

pTemp2 = newList->pHead;

//check all node data of newList and add data to result if it's not in result yet

while (pTemp2 != NULL){

pTemp1 = result->pHead;

while (pTemp1 != NULL){

flag = true;

//if a node is in result already, break and move to next node

if (pTemp2->data == pTemp1->data){

flag = false;

break;

}

pTemp1 = pTemp1->next;

}

//if flag is still true which means this node data is not in result yet, add it to result

if (flag == true){

result->addFirst(pTemp2->data);

}

pTemp2 = pTemp2->next;

}

return result;

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(0);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: ";

Poly1.display();

cout<<"\nPolylist 2: ";

Poly2.addFirst(1);

Poly2.addFirst(3);

Poly2.addFirst(5);

Poly2.addFirst(7);

Poly2.display();

cout<<"\n";

//test: Print Polynomial:

/\*

cout<<"\n";

Poly1.printPoly();

cout<<"\n";

Poly2.printPoly();

\*/

//test: reverse a list using stack method:

/\*

Poly1.reverseList();

Poly1.display();

\*/

//test: reverse a list using queue method:

/\*

Poly1.reverseList2();

Poly1.display();

\*/

//test: append() method

/\*

Poly1.append(Poly2);

cout<<"List 1: ";

Poly1.display();

cout<<"\nList 2 is now empty: ";

Poly2.display();

\*/

//test: add 2 polynomials - addPolyList() function

/\*

cout<<"\nGet the result as the addition of List 1 and List 2: \n";

List \*result;

result = Poly1.addPolyList(&Poly2);

result->display();

\*/

//test: get the intersection of 2 polynomials

/\*

cout<<"\nGet the intersection of List 1 and List 2: \n";

List \*result;

result = new List;

result = Poly1.getIntersection(&Poly2);

result->display();

\*/

//test: get the union of 2 polynomials

/\*

cout<<"\nGet the intersection of List 1 and List 2: \n";

List \*result;

result = new List;

result = Poly1.getUnion(&Poly2);

result->display();

\*/

}

**Question 1 & 2: Enter a linked list 5->0->1->0->1 and complete the display() method to print the linked list**

void List::display() {

Node\* pTemp = pHead;

while (pTemp!=NULL) {

cout << pTemp->data << " ";

pTemp = pTemp->next;

}

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

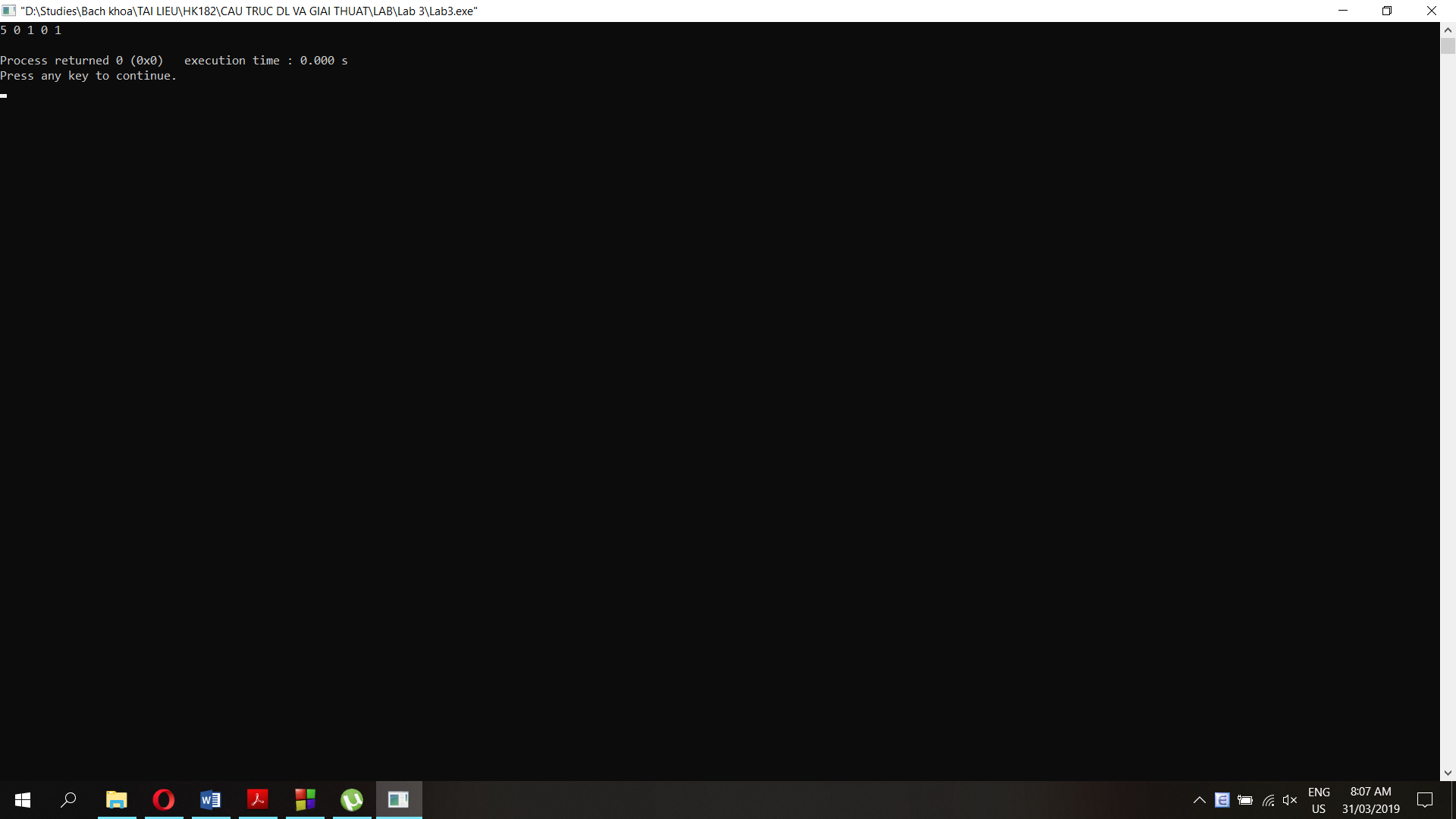
Poly1.addFirst(0);

Poly1.addFirst(5);

Poly1.display();

cout<<"\n";

}



**Question 3: Complete addConstant() method which adds a constant to the polynomial and addPoly() method which adds 2 polynomials as linked list**

//Add a new node at the end of a linked list with the data as input

void List::addLast(int newdata){

Node \*pTemp = pHead;

//if there's no element in the list, return the head with new data

if (pTemp == NULL){

this->addFirst(newdata);

}

//else point pTemp to the last element and add another node after it

else {

while (pTemp->next != NULL){

pTemp = pTemp->next;

}

pTemp->next = new Node;

pTemp->next->data = newdata;

pTemp->next->next = NULL;

count++;

}

}

void List::addConstant(int nConst) {

pHead->data += nConst;

return;

}

//add 2 polynomial lists, return a new list

List\* List::addPolyList(List \*Poly) {

List \*result;

result = new List;

Node \*pTemp1 = this->pHead;

Node \*pTemp2 = Poly->pHead;

int n = 0;

//until we reach the end of both lists

while (pTemp1 != NULL || pTemp2 != NULL){

//if we reach the end of list 1, add data of list 2 to result

if (pTemp1 == NULL && pTemp2 != NULL){

result->addLast(pTemp2->data);

pTemp2 = pTemp2->next;

}

//if we reach the end of list 2, add data of list 1 to result

else if (pTemp1 != NULL && pTemp2 == NULL){

result->addLast(pTemp1->data);

pTemp1 = pTemp1->next;

}

//else: add data of both list and add to result

else{

result->addLast(pTemp1->data + pTemp2->data);

pTemp1 = pTemp1->next; //move ptemp to next node

pTemp2 = pTemp2->next;

}

n++;

}

result->count = n;

return result;

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: ";

Poly1.display();

cout<<"\nPolylist 2: ";

Poly2.addFirst(1);

Poly2.addFirst(3);

Poly2.addFirst(5);

Poly2.addFirst(7);

Poly2.display();

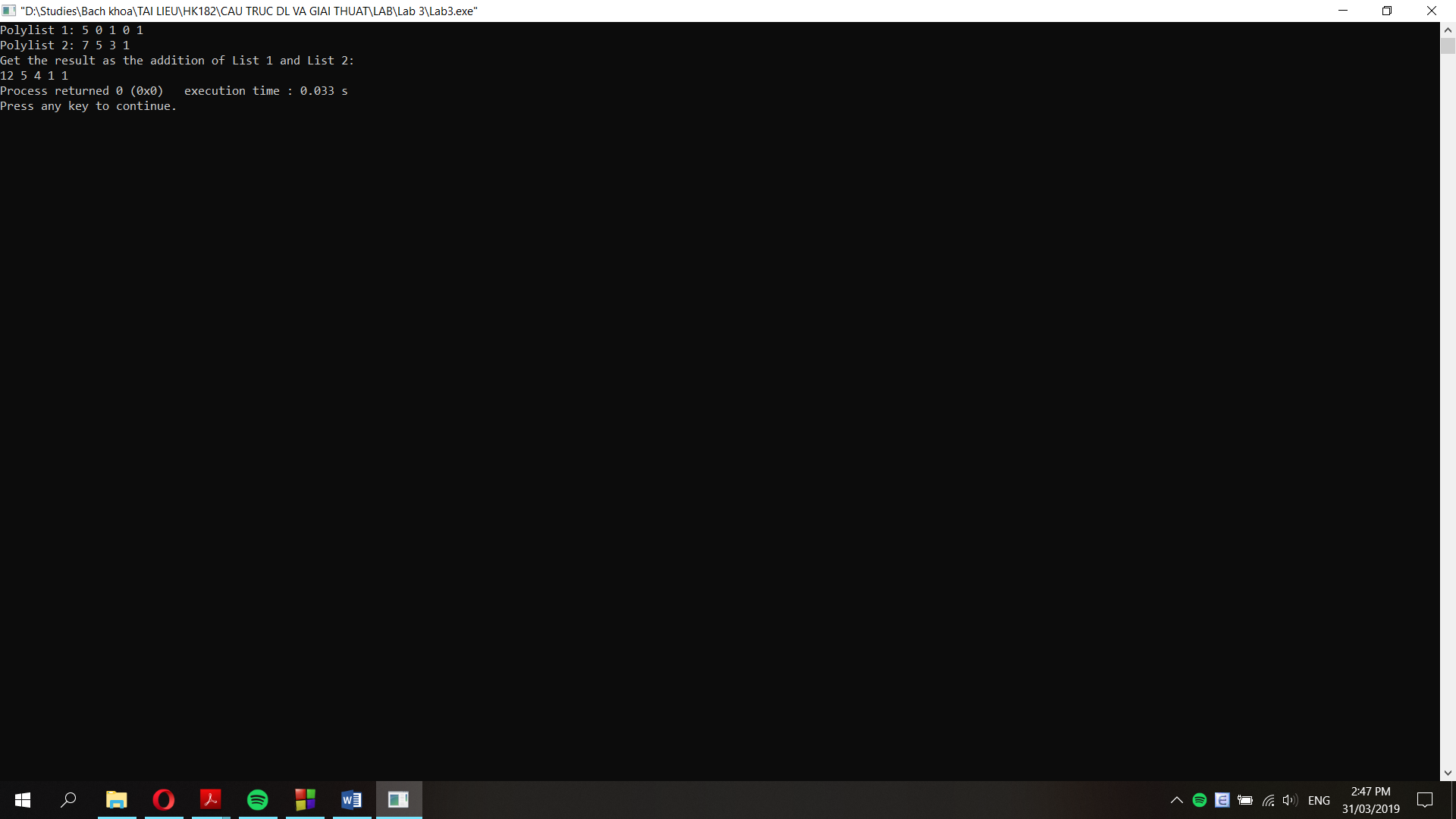
cout<<"\nGet the result as the addition of List 1 and List 2: \n";

List \*result;

result = Poly1.addPolyList(&Poly2);

result->display();

}



**Question 4: printPoly() method**

//print a list using array

void List::printPoly(){

int n = this->count-1;

int \*arr = new int [n];

Node \*pTemp = pHead;

int i=0;

while (pTemp != NULL){

\*(arr+i) = pTemp->data;

pTemp = pTemp->next;

i++;

}

for (i=0; i<n; i++){

if (\*(arr+n-i)==0){

continue;

}

else if (\*(arr+n-i)==1){

cout<<"x^"<<n-i<<" + ";

}

else{

cout<<\*(arr+n-i)<<"x^"<<n-i<<" + ";

}

}

cout<<\*arr;

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(5);

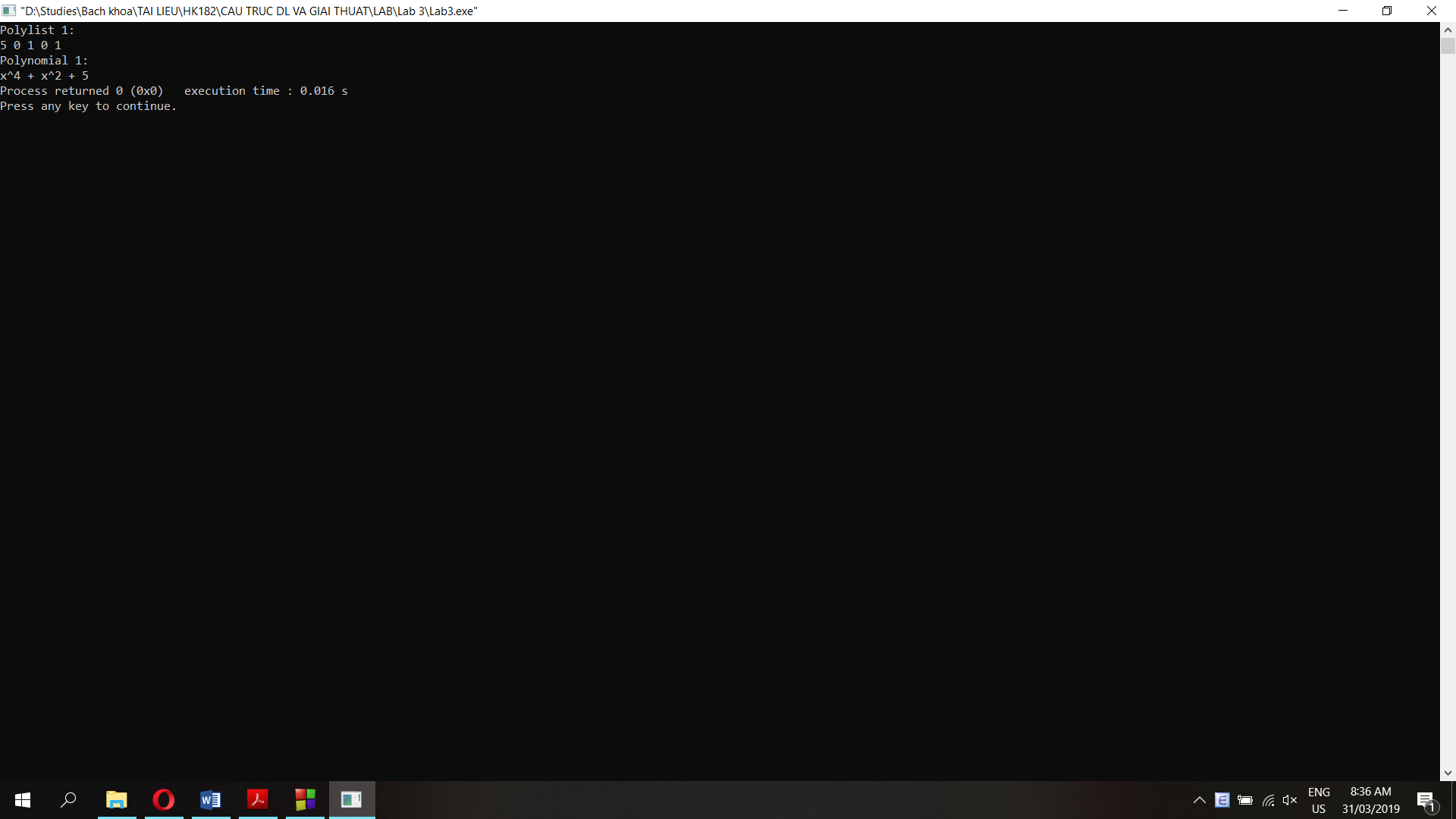
cout<<"Polylist 1: \n";

Poly1.display();

cout<<"\nPolynomial 1: \n";

Poly1.printPoly();

}



**Question 5:**

**Question 6:**

**a. Reverse a list using stack method**

//get the first node out of the list

Node\* List::getFirst(){

Node \*pTemp = this->pHead;

if (this->pHead == NULL){

return this->pHead;

}

else if (this->pHead->next == NULL){

pHead = NULL;

this->count--;

return pTemp;

}

else {

this->pHead = this->pHead->next;

this->count--;

return pTemp;

}

}

//add a node before the head

void List::addFirstNode(Node \*node){

if (this->pHead == NULL){

pHead = node;

pHead->next = NULL;

}

else if (this->pHead->next == NULL){

Node \*pTemp = pHead;

pHead = node;

pHead->next = pTemp;

pTemp->next = NULL;

}

else{

Node \*pTemp = pHead;

pHead = node;

pHead->next = pTemp;

}

this->count++;

}

//add a node at the end of the list

void List::addLastNode(Node \*node){

if (this->pHead == NULL){

this->pHead = node;

this->pHead->next = NULL;

}

else if (this->pHead->next == NULL){

this->pHead->next = node;

node->next = NULL;

}

else{

Node \*pTemp = this->pHead;

while (pTemp->next != NULL){

pTemp = pTemp->next;

}

pTemp->next = node;

node->next = NULL;

}

count++;

}

//reverse a list using Stack method

void List::reverseList(){

Node \*pTemp = this->pHead;

List sTemp;

int n = this->count;

while(n!=0){

//List: 1->2->3->4 => Stack: 4->3->2->1

pTemp = pTemp->next;

sTemp.addFirstNode(this->getFirst());

n--;

}

n = sTemp.count;

pTemp = sTemp.pHead;

while(n!=0){

//Stack: 4->3->2->1 => List: 4->3->2->1

pTemp = pTemp->next;

this->addLastNode(sTemp.getFirst());

n--;

}

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: \n";

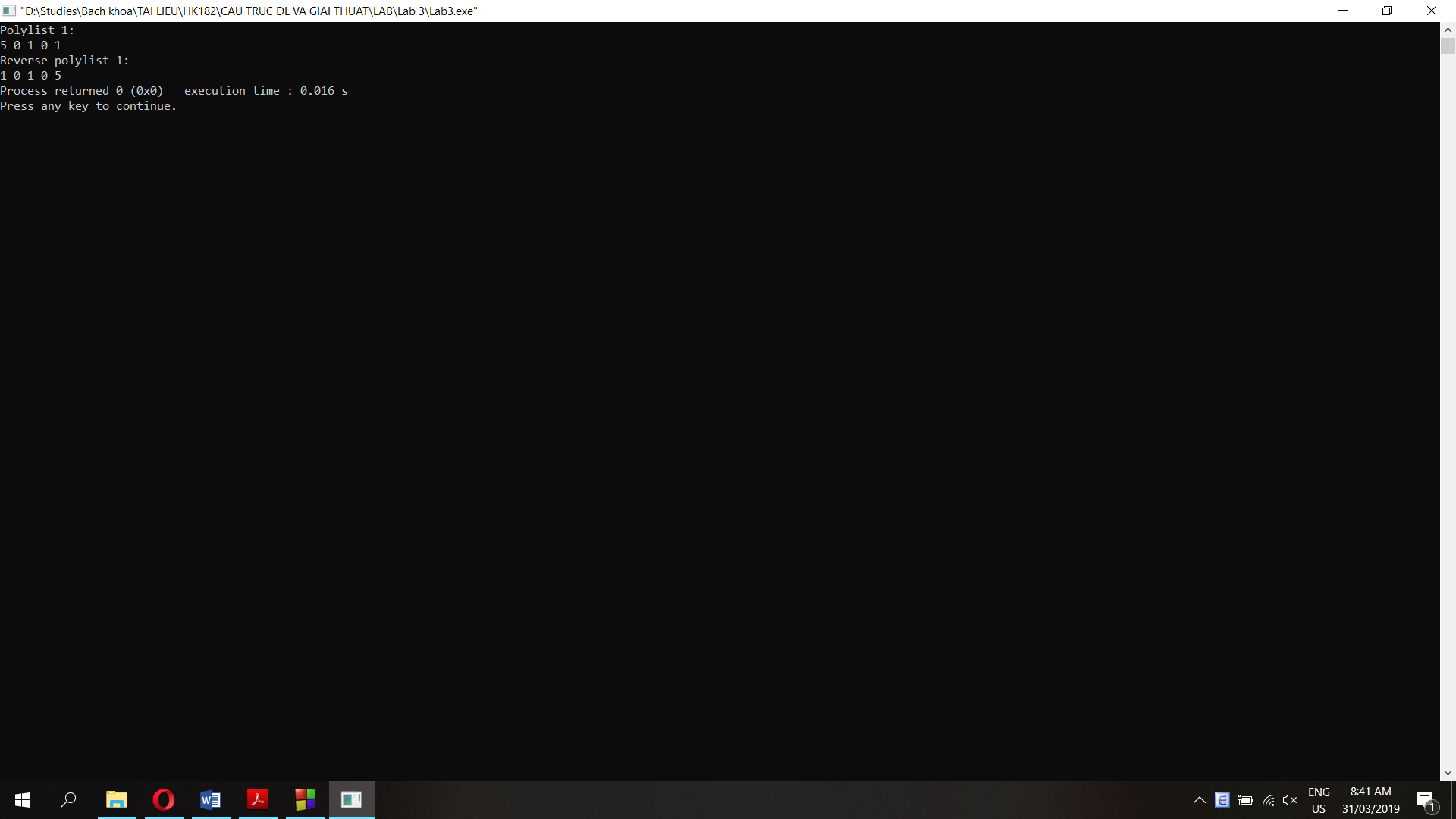
Poly1.display();

Poly1.reverseList();

cout<<"\nReverse polylist 1: \n";

Poly1.display();

}



**b. Reverse a linked list using queue method**

//get the first node out of the list

Node\* List::getFirst(){

Node \*pTemp = this->pHead;

if (this->pHead == NULL){

return this->pHead;

}

else if (this->pHead->next == NULL){

pHead = NULL;

this->count--;

return pTemp;

}

else {

this->pHead = this->pHead->next;

this->count--;

return pTemp;

}

}

//get the last node out of the list

Node\* List::getLast(){

Node \*pTemp = this->pHead;

if (this->pHead == NULL){

return this->pHead;

}

else if (this->pHead->next == NULL){

pHead = NULL;

this->count--;

return pTemp;

}

else {

while(pTemp->next->next!=NULL){

pTemp = pTemp->next;

}

Node \*pTemp2 = pTemp->next;

pTemp->next = NULL;

this->count--;

return pTemp2;

}

}

//add a node before the head

void List::addFirstNode(Node \*node){

if (this->pHead == NULL){

pHead = node;

pHead->next = NULL;

}

else if (this->pHead->next == NULL){

Node \*pTemp = pHead;

pHead = node;

pHead->next = pTemp;

pTemp->next = NULL;

}

else{

Node \*pTemp = pHead;

pHead = node;

pHead->next = pTemp;

}

this->count++;

}

//reverse a list using queue method

void List::reverseList2(){

List qTemp;

int n = this->count;

while(n!=0){

//List: 1->2->3->4 => Queue: 4->3->2->1

qTemp.addFirstNode(this->getFirst());

n--;

}

n = qTemp.count;

while(n!=0){

//Queue: 4->3->2->1 => List: 4->3->2->1

this->addFirstNode(qTemp.getLast());

n--;

}

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: \n";

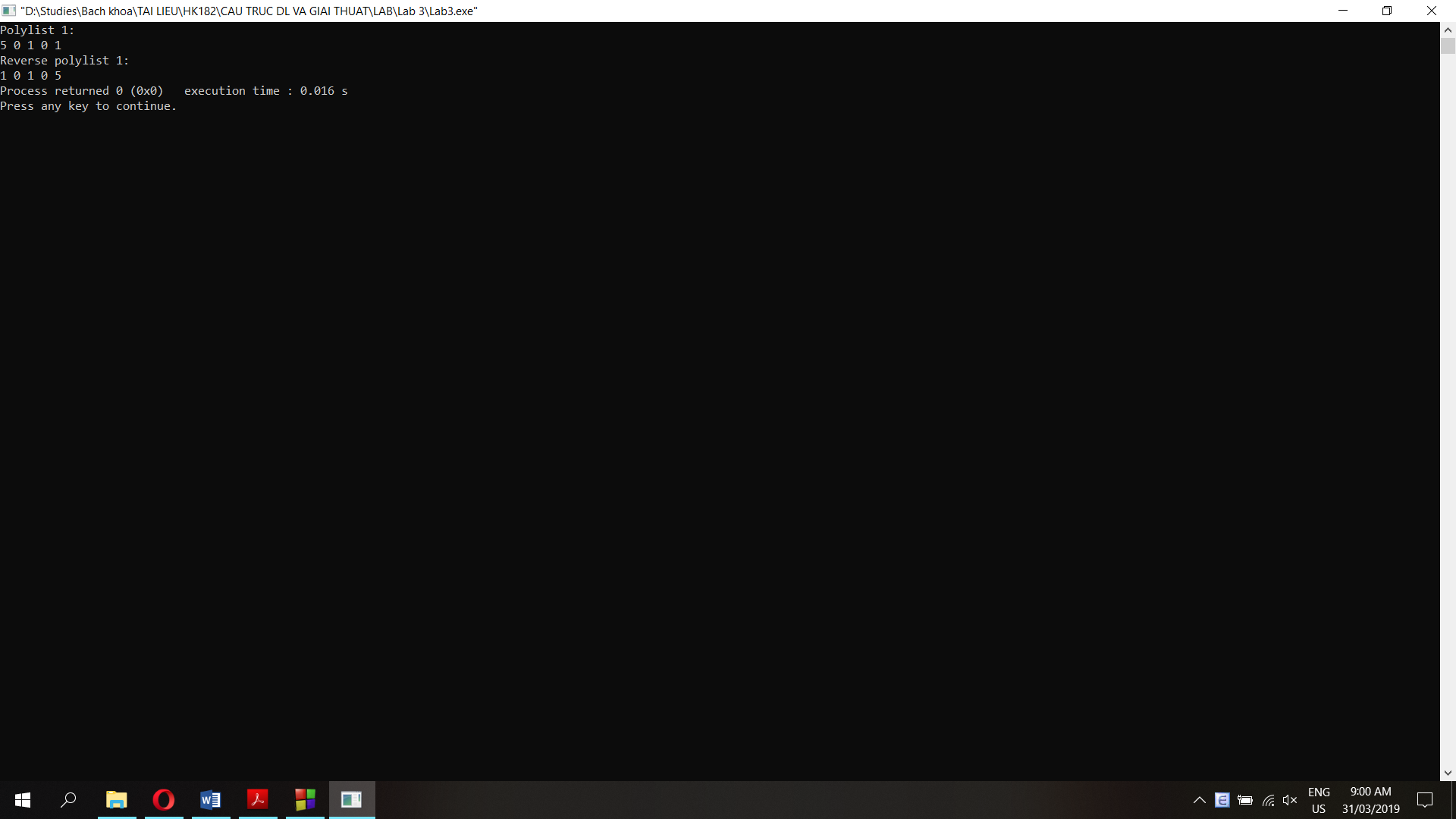
Poly1.display();

Poly1.reverseList2();

cout<<"\nReverse polylist 1: \n";

Poly1.display();

}



**Question 7: append() method which appends a new list to the current list then empty the new list**

//Append newList to the current list

void List::append(List &newList){

int n = newList.count;

while (n>0){

this->addLastNode(newList.getFirst());

n--;

}

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: ";

Poly1.display();

cout<<"\nPolylist 2: ";

Poly2.addFirst(1);

Poly2.addFirst(3);

Poly2.addFirst(5);

Poly2.addFirst(7);

Poly2.display();

cout<<"\nAppend Polylist 2 to Polylist 1: \n";

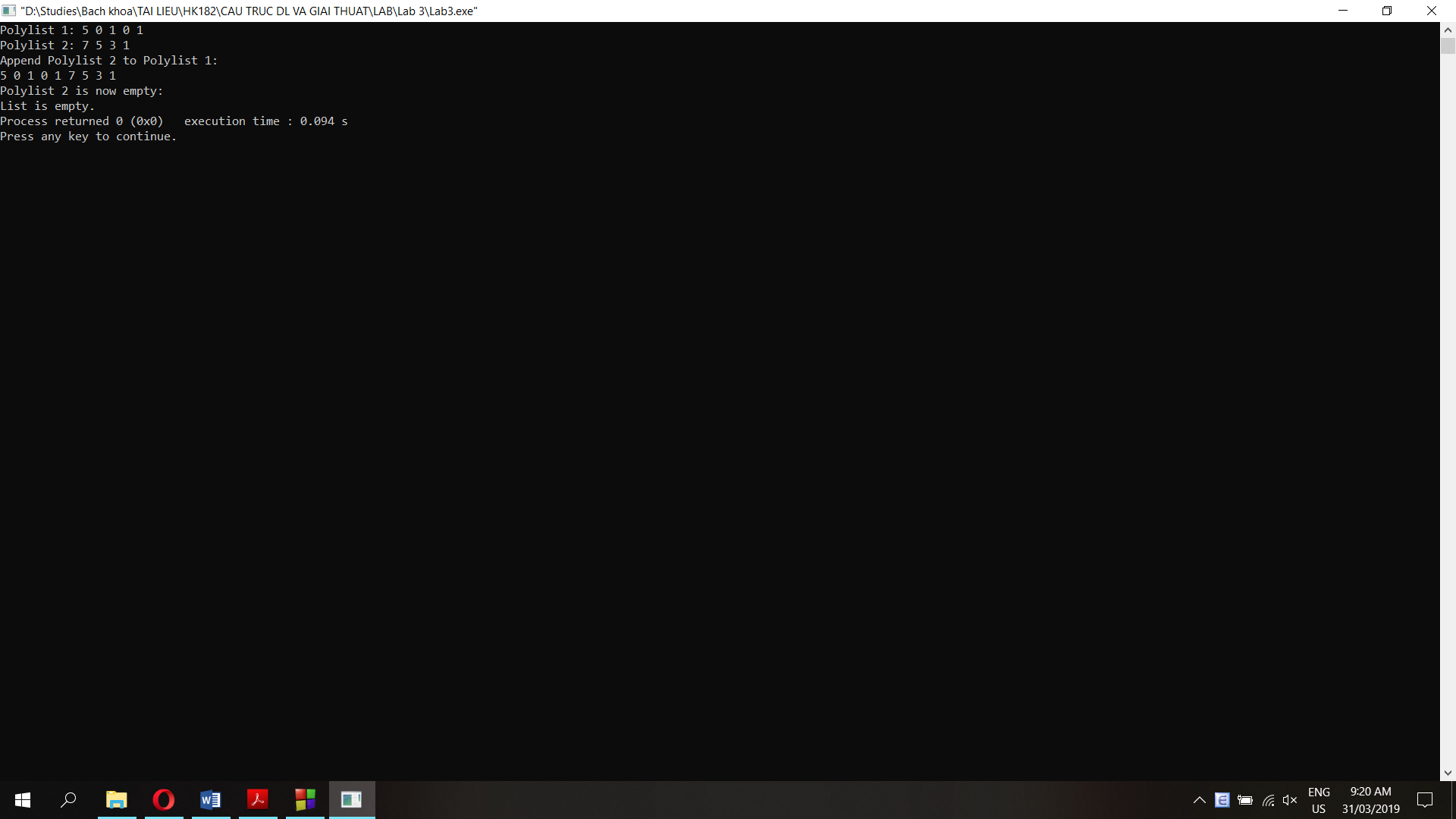
Poly1.append(Poly2);

Poly1.display();

cout<<"\nPolylist 2 is now empty: \n";

Poly2.display();

}



**Question 8: getIntersection() method which returns a new list including the intersection of 2 linked lists**

//find the intersection of 2 lists and return a new list

List\* List::getIntersection(List \*newList){

Node \*pTemp1 = this->pHead;

Node \*pTemp2;

List \*result;

result = new List;

while (pTemp1 != NULL){

//point pTemp from pHead to tail of the current list

pTemp2 = newList->pHead;

while (pTemp2 != NULL){

//point pTemp2 from pHead to tail of the new list

if (pTemp2->data == pTemp1->data){

result->addLast(pTemp1->data); //if 2 nodes have the same data, add it to result

}

pTemp2 = pTemp2->next;

}

pTemp1 = pTemp1->next;

}

return result;

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(0);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: ";

Poly1.display();

cout<<"\nPolylist 2: ";

Poly2.addFirst(1);

Poly2.addFirst(3);

Poly2.addFirst(5);

Poly2.addFirst(7);

Poly2.display();

cout<<"\nGet the intersection of List 1 and List 2: \n";

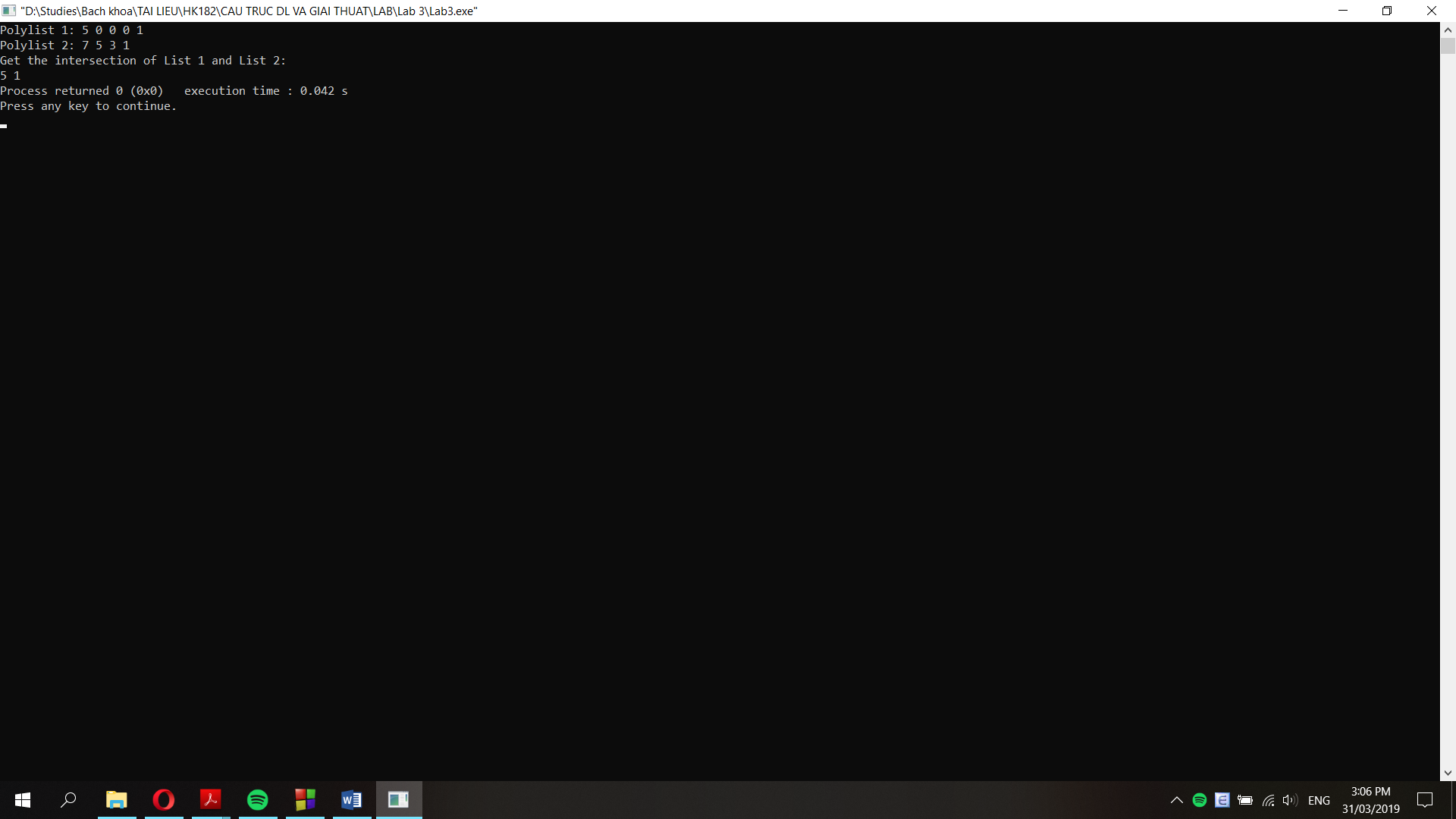
List \*result;

result = new List;

result = Poly1.getIntersection(&Poly2);

result->display();

}



**Question 9: getUnion() method which returns a new list including the union of 2 linked lists**

//find the union of 2 lists and return a new list

List\* List::getUnion(List \*newList){

Node \*pTemp1 = this->pHead;

Node \*pTemp2;

List \*result;

result = new List;

bool flag = true; //flag is true when a node data is not in result yet

//add all data of the current list to result first

pTemp2 = this->pHead;

result->addFirst(pTemp2->data);

while (pTemp2 != NULL){

pTemp1 = result->pHead;

while (pTemp1 != NULL){

flag = true;

//if a node is in result already, break and move to next node

if (pTemp2->data == pTemp1->data){

flag = false;

break;

}

pTemp1 = pTemp1->next;

}

//if flag is still true which means this node data is not in result yet, add it to result

if (flag == true){

result->addFirst(pTemp2->data);

}

pTemp2 = pTemp2->next;

}

pTemp2 = newList->pHead;

//check all node data of newList and add data to result if it's not in result yet

while (pTemp2 != NULL){

pTemp1 = result->pHead;

while (pTemp1 != NULL){

flag = true;

//if a node is in result already, break and move to next node

if (pTemp2->data == pTemp1->data){

flag = false;

break;

}

pTemp1 = pTemp1->next;

}

//if flag is still true which means this node data is not in result yet, add it to result

if (flag == true){

result->addFirst(pTemp2->data);

}

pTemp2 = pTemp2->next;

}

return result;

}

int main() {

List Poly1, Poly2;

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(1);

Poly1.addFirst(0);

Poly1.addFirst(5);

cout<<"Polylist 1: ";

Poly1.display();

cout<<"\nPolylist 2: ";

Poly2.addFirst(1);

Poly2.addFirst(3);

Poly2.addFirst(5);

Poly2.addFirst(7);

Poly2.display();

cout<<"\nGet the union of List 1 and List 2: \n";

List \*result;

result = new List;

result = Poly1.getUnion(&Poly2);

result->display();

}

