**Data structure: Assignment 1**

**결과 보고서**

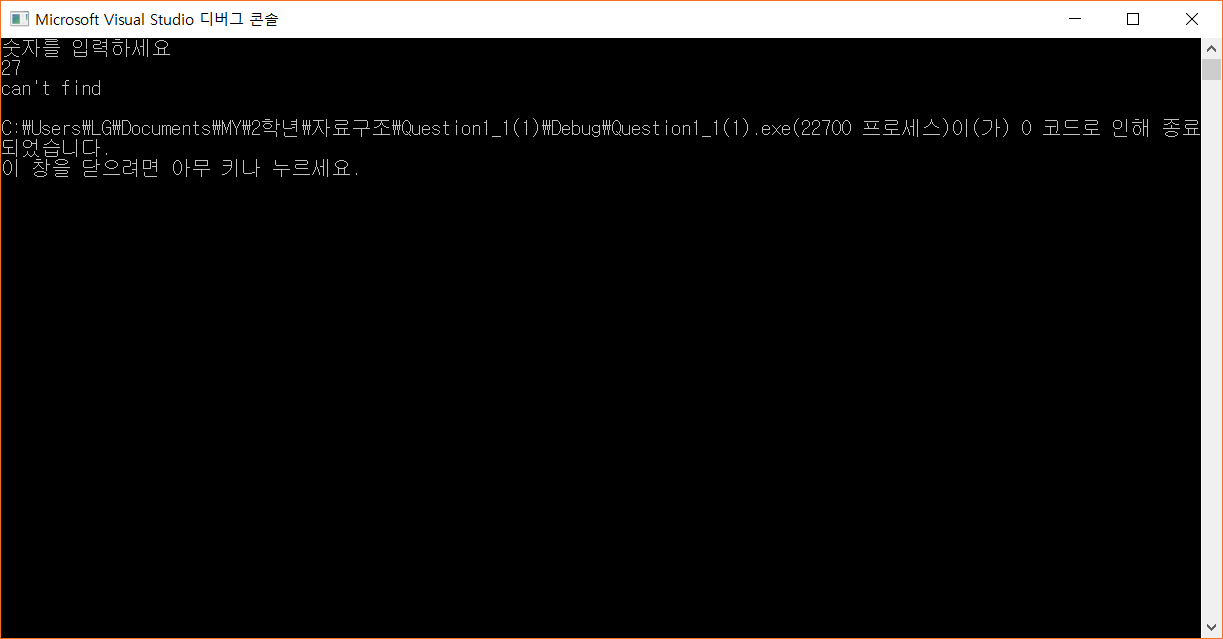
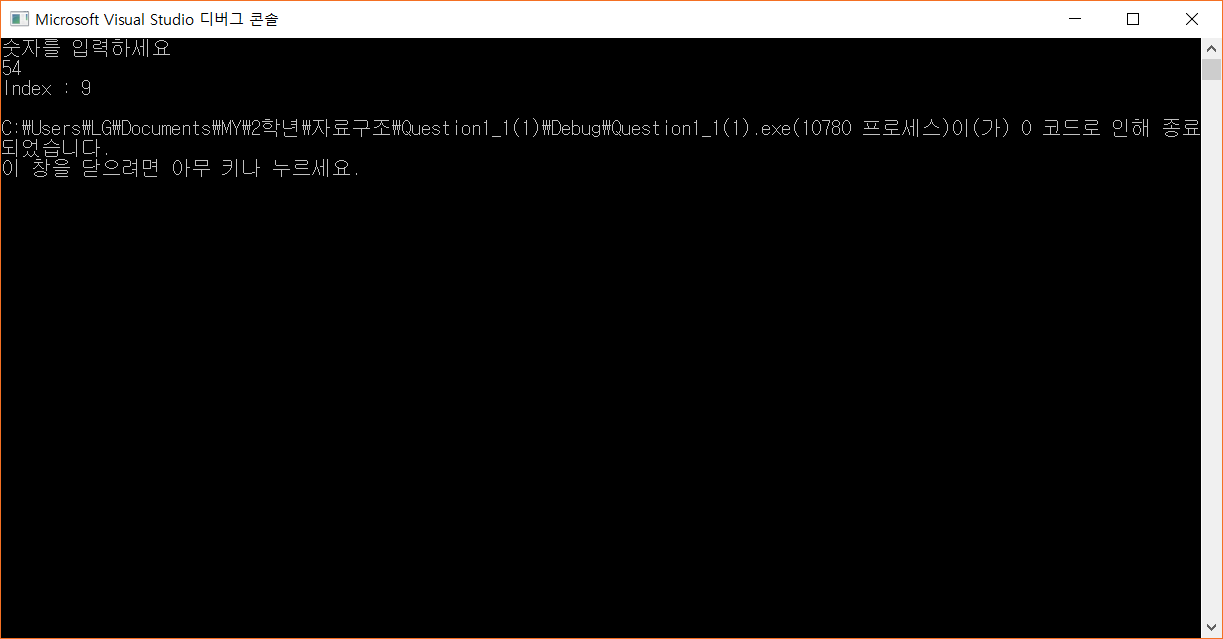
**#1.1 Binary search**

*-Recursive*

**\*구현방법**

처음 중간의 값을 임의의 값으로 정하여 그 값이 찾고자 하는 값과 크면 그 값은 새로운 끝이 되며, 작으면 그 값은 새로운 시작이 된다. 재귀는 self-call 이므로 나올 수 있는 경우마다 시작과 끝을 바꿔가면서 return하여 구현하였다. 파일은 vector로 먼저 저장한 후 int배열에 복사하여 구현하였다.

**\*실행 결과**

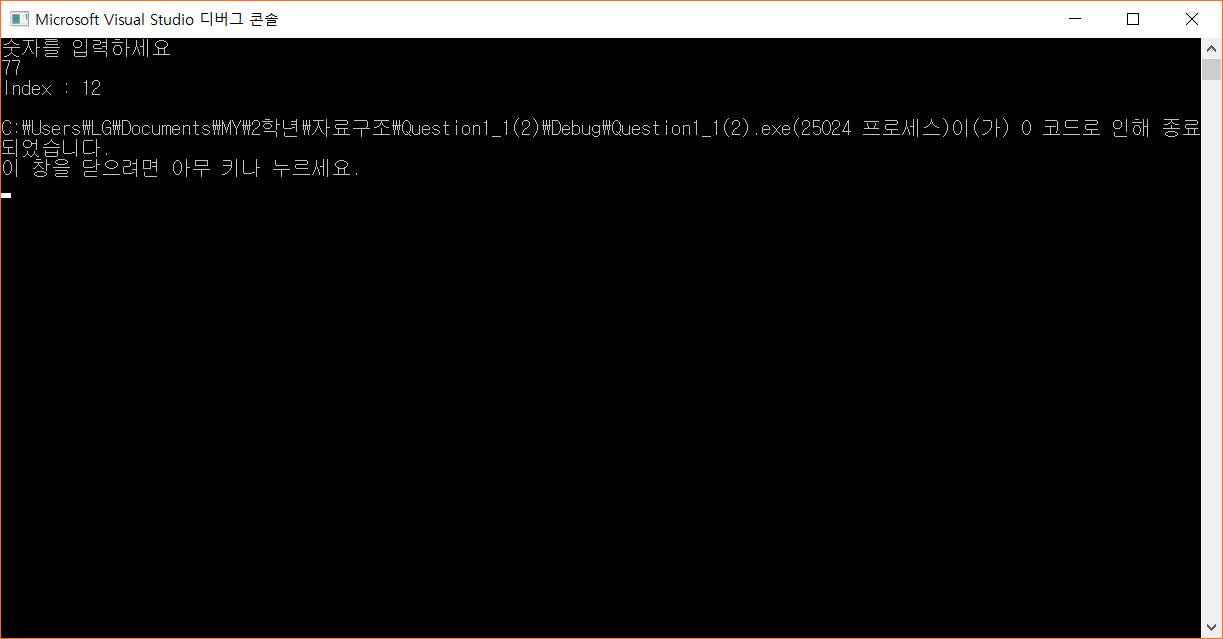
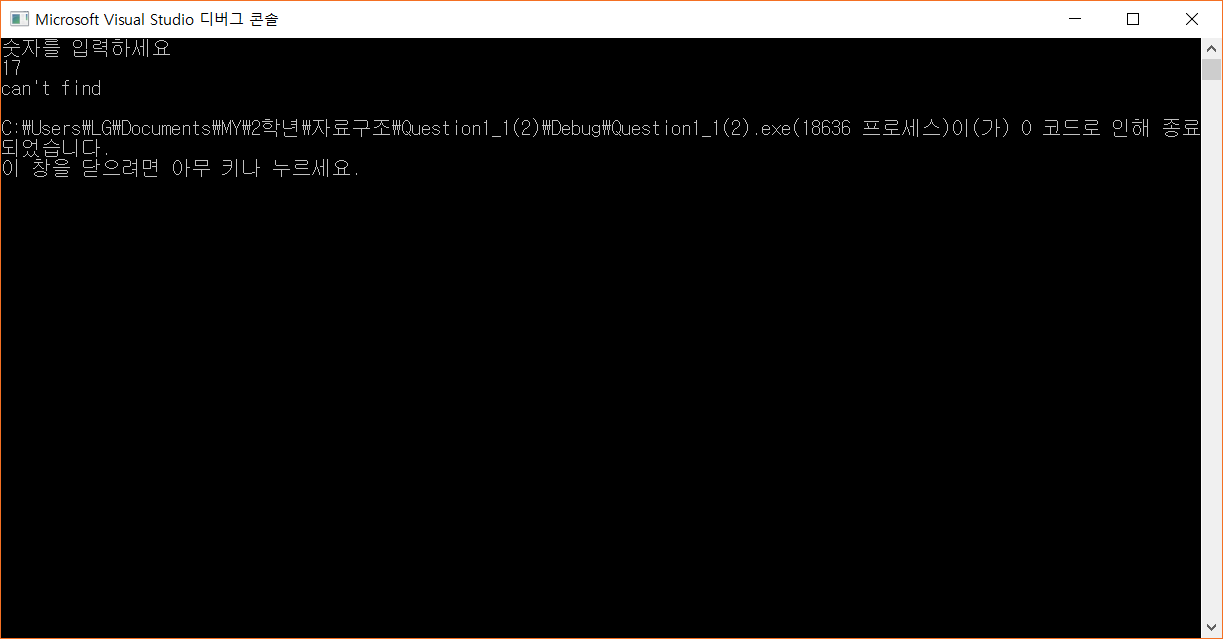


*-Iterative*

**\*구현방법**

처음 중간의 값을 임의의 값으로 정하여 그 값이 찾고자 하는 값과 크면 그 값은 새로운 끝이 되며, 작으면 그 값은 새로운 시작이 된다. 반복은 시작이 끝보다 작거나 같을 때까지 시작과 끝을 바꿔가며 구현하였다. 파일은 vector로 먼저 저장한 후 int배열에 복사하여 구현하였다.

**\*실행 결과**

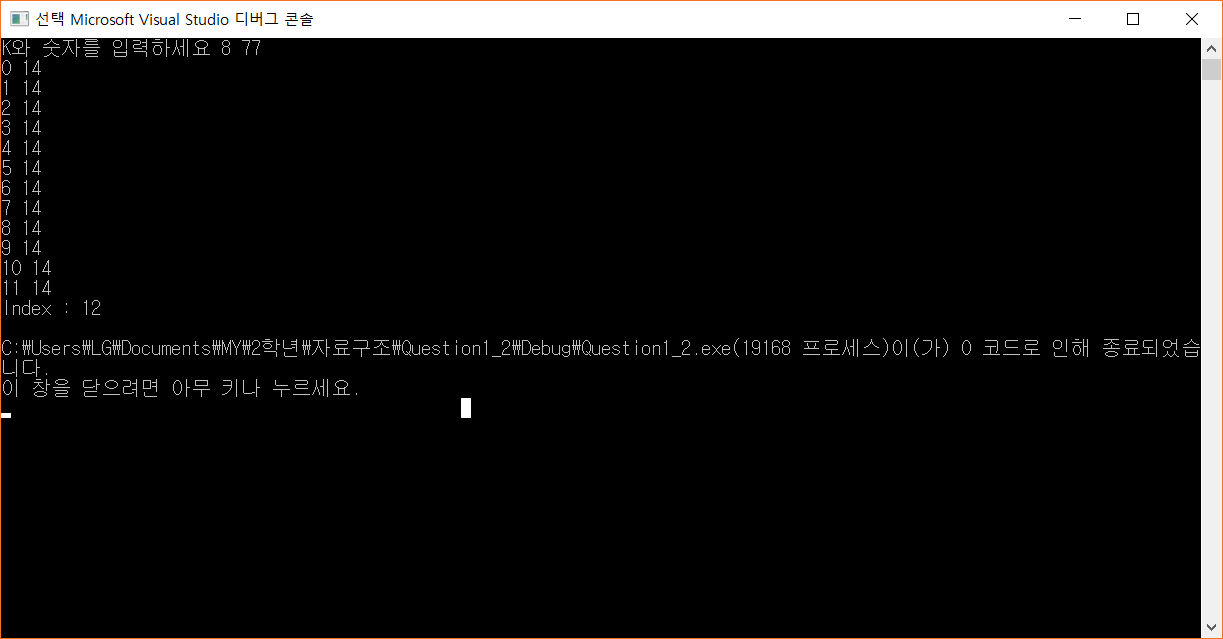
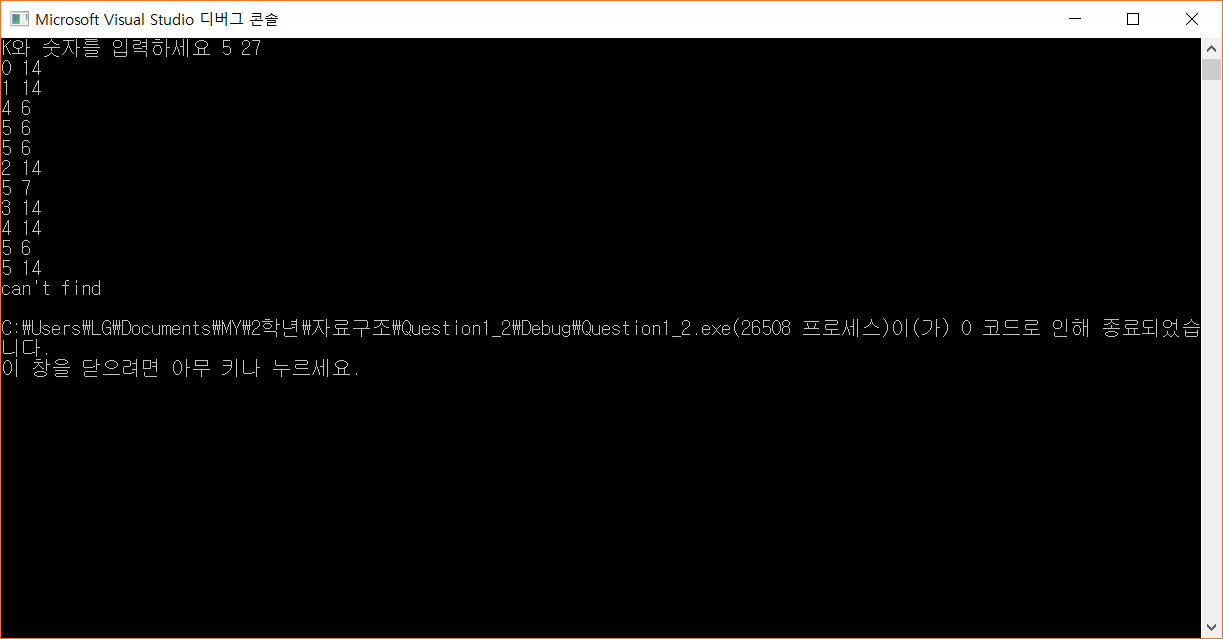


**#1.2 k-ary search**

**\*구현방법**

시작 값이 찾는 값과 같으면 시작을 return하고 끝 값이 찾는 값과 같으면 끝을 return한다. 만약 범위에 벗어날 때는 -1을 return하여 “can’t find”를 출력하게 한다. 그 외의 경우에는 찾는 값이 k개의 파트 중 처음 파트의 값과 그 다음 파트 값의 사이에 있을 경우( if (A[start + (end – start) \* I / k] <= key && key <= A[ start + ((end – start) \* (i + 1) / k ) ] ) ) 시작과 끝을 출력하고 시작을 늘려가며 몇 번째 index에 있는 지 파악한다. 이 방식을 시작이 끝보다 작거나 같을 경우 반복하여 구현하였다.

**\*실행 결과**



**#1.3 이중 연결 리스트**

-1.3.1 이중연결 리스트 구현 : 정수형 버전

**\*class 계층도**

DoublyListP.h

#include “DoublyListP.h” #include “DoublyListP.h”

DoublyListTest.cpp

DoublyListP.cpp

**\*구현방법**

1) void Insert(int Position, int Item)

기존의 노드(A)가 존재할 때 추가할 노드(X)의 다음을 기존의 노드로 연결한다. 또한 기존의 노드의 이전(B)을 추가할 노드로 연결하며 기존의 노드의 전 것을 추가할 노드와 연결시킨다. (X->Next=A, A->Prev=X, B->Next=X, X->Prev=B로 지정한다.)

2) void Delete(int Position)

삭제하려는 노드의 이전 노드(X)와 다음 노드(Y)를 찾아 둘을 연결시켜 삭제한다. (X의 Next를 Y로, Y의 Prev를 X로 지정한다.)

3) void Retrieve(int Position, int &Item)

Position번째에 있는 숫자를 Item에 저장하는 것이므로 Head부터 시작하여 Position번째까지 Next를 한다. 해당되는 인덱스가 나타나면 Item에 해당되는 인덱스의 값을 저장한다. 만약 Position이 범위를 벗어나면 “Position out of Range”을 출력하도록 한다.

4) bool isEmpty()

비어 있을 경우는 Count=0일때이므로 return(Count==0)을 한다.

5) int Length()

Count를 나타내는 함수이므로 Count를 return한다.

6) void print()

Head부터 시작하여 끝까지 Next를 하며 원소들을 출력하도록 한다.

7) listClass()

Count을 0, Head와 Tail를 NULL로 초기화한다.

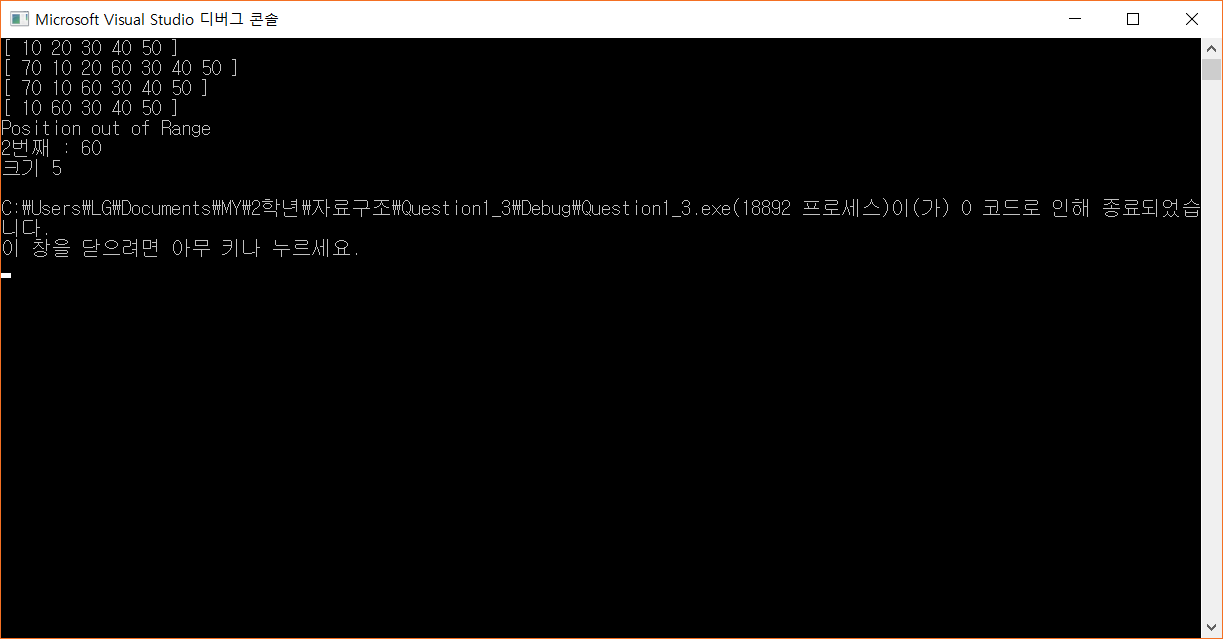
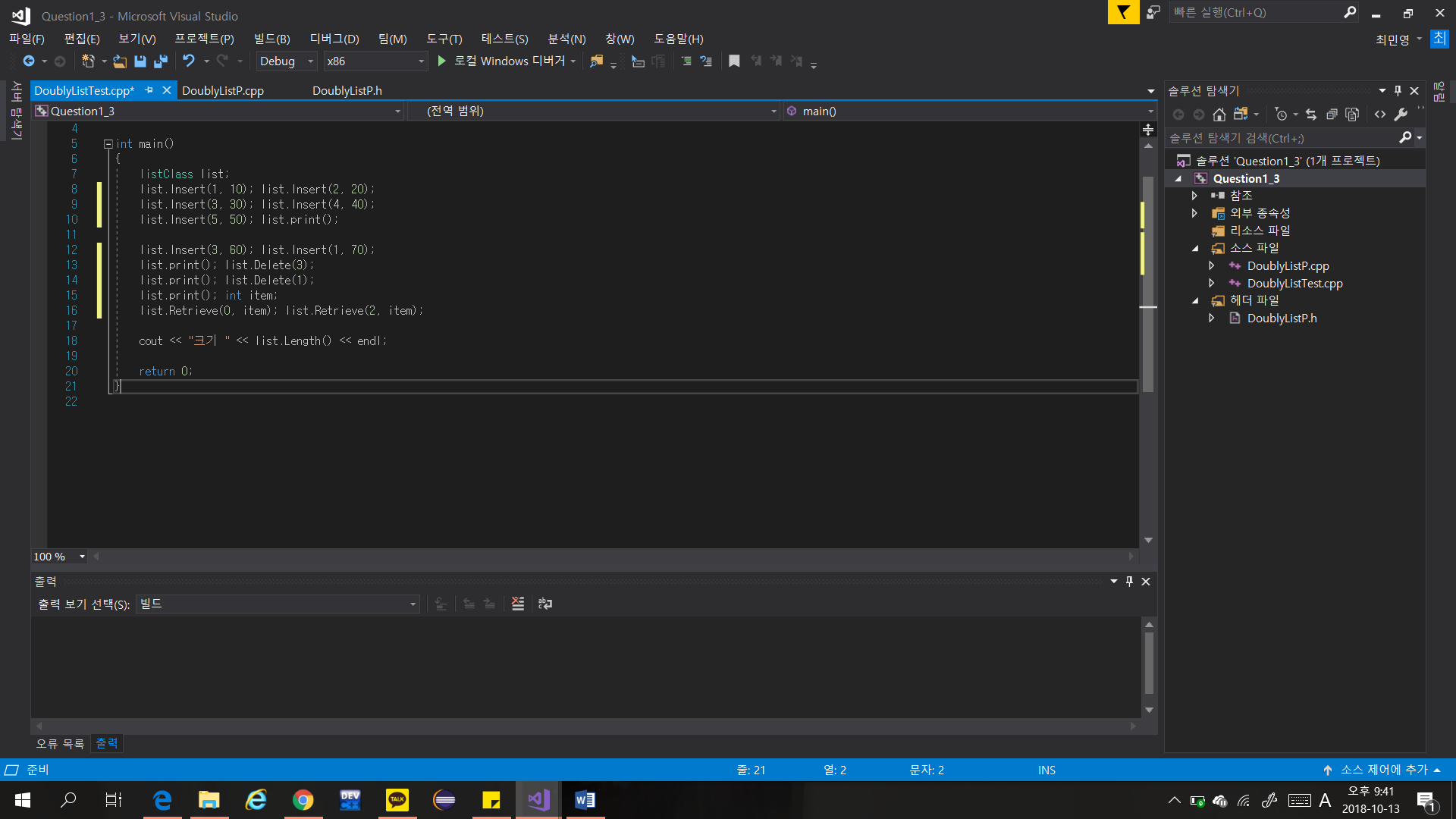
8) listClass(const listClass& L)

리스트의 길이를 복사하고, 빈 리스트면 자신도 빈 리스트가 되게 한다. 만약 빈 리스트가 아니라면 새 노드를 만들어서 데이터를 복사한다.

9) ~listClass()

리스트에 있는 것들을 모두 빈 리스트로 만든다.

**\*실행 결과**



↑(Test 내용) ↑(결과)

-1.3.2 이중연결 리스트 구현 : template 버전

**\*class 계층도**

GenericDoublyListP.h

#include “GenericDoublyListP.h” #include “GenericDoublyListP.h”

GenericDoublyListTest.cpp

GenericDoublyListP.cpp

#include “GenericDoublyListP.cpp

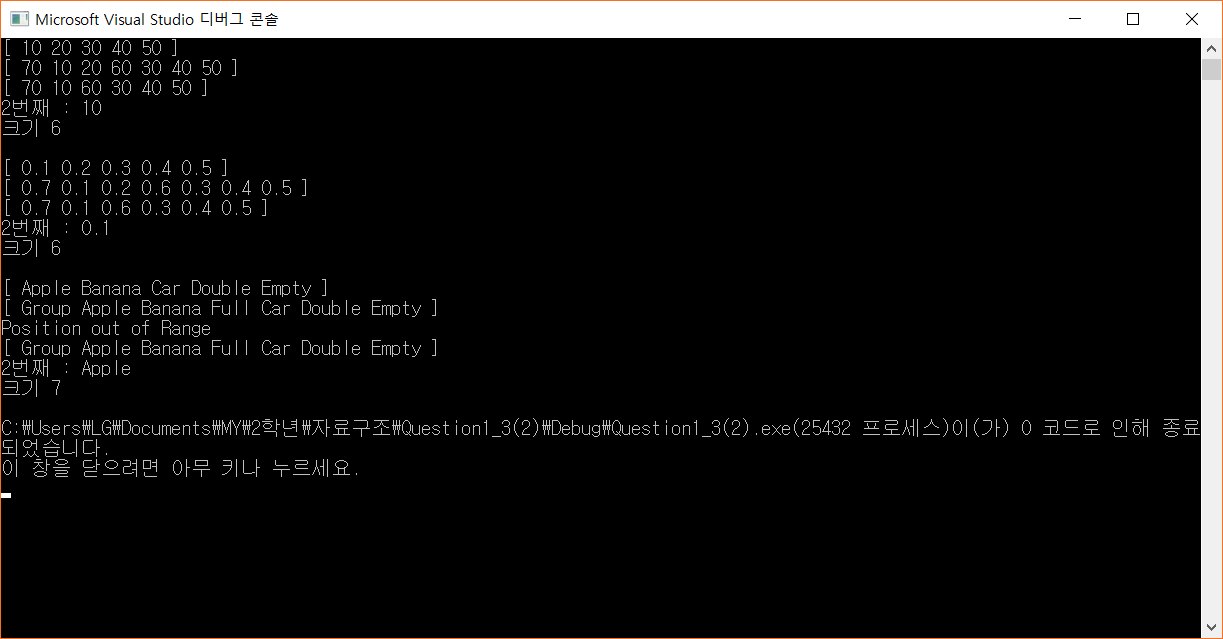
(template의 선언과 구현을 같이 나타내지 않으면 LNK2019 에러가 생긴다. 그래서 GenericDoublyListP.cpp를 Test파일에 include하여 나타냈다.)

**\*구현 방법**

template <typename T> 를 사용하여 모든 type을 쓸 수 있게 구현하였다.

함수의 구현은 1.3.1과 같으며 void Insert(int Position, T Item), void Retrieve(int Position, T& Item)에서 type을 T로 바꿔서 나타냈다.

**\*실행 결과**



(순서대로 listClass<int>, listClass<double>, listClass<string>일 때의 실행 결과)

-1.3.3 이중연결 리스트 구현 - java 버전

**\*구현방법**

1) DoublylinkedList()

리스트의 크기를 0, Head를 NULL로 초기화 시켰다.

2) class Node

1.3.1에서의 nodeRecord구조체처럼 class Node를 만들어 그와 같게 구현하였다.

3) Node node(int index)

Index에 있는 node를 가져오기 위해 함수를 구현하였다. Index의 범위가 0보다 작거나 리스트의 크기를 벗어날 시에는 예외를 만들어 Index와 Size를 출력하도록 하였다. 만약 Index가 리스트 길이의 (1/2)보다 작을 때는 Head부터 Index까지 Next를 하여 해당되는 노드를 return 하도록 하였다. 그리고 그 외의 경우에는 Tail부터 index까지 Prev를 하여 해당되는 노드를 return 하도록 하였다.

4) void addFirst(Object input) & void addLast(Object input) & void add(Object input)

addFirst는 맨 처음 index에 데이터를 추가하기 위해 추가할 노드가 다음 노드로 첫번째 노드를 가리키게 한다. 리스트가 비어 있으면 Head가 마지막 노드를 추가할 노드로 그 외의 경우에는 첫 노드의 앞 노드를 추가할 노드를 가리키도록 한다. 그런 후 Head가 첫번째 노드로 추가할 노드를 가리키도록 한다.

addLast는 맨 마지막에 데이터를 추가하기 위해 add(size, input)를 호출하여 구현하였다.

add는 input을 맨 마지막에 데이터를 추가하도록 구현하였다.

5) void add(int index, Object input)

Index에 input을 추가하기 위해서 만든 함수로 index가 0일때는 addFirst()를 호출하고 그 외에는 1.3.1의 Insert와 같은 방식으로 구현하였다.

6) Object removeFirst() & Object removeLast() & Object remove(int index)

1.3.1의 Delete와 같은 방식으로 구현하였다. (removeFirst는 첫번째 데이터를 삭제, removeLast는 마지막 데이터를 삭제하도록 하였다.)

7) Object get(int index) & Object getFirst()

get 함수는 1.3.1의 Retrieve함수와 같은 기능을 가지고 있다. Index에 해당되는 데이터를 return 하도록 구현하였다. getFirst는 맨 앞의 데이터를 return 하도록 작성하였다.

8) int indexOf(Object data)

data에 해당되는 index를 return 하도록 구현하였다. 같은 데이터인지 판단될 때까지 Next를 하며 해당되는 index를 반환한다. 만약 리스트에 없는 데이터일 경우 -1을 반환하여 다른 함수에서 예외 구문으로 넘어가게 만들었다.

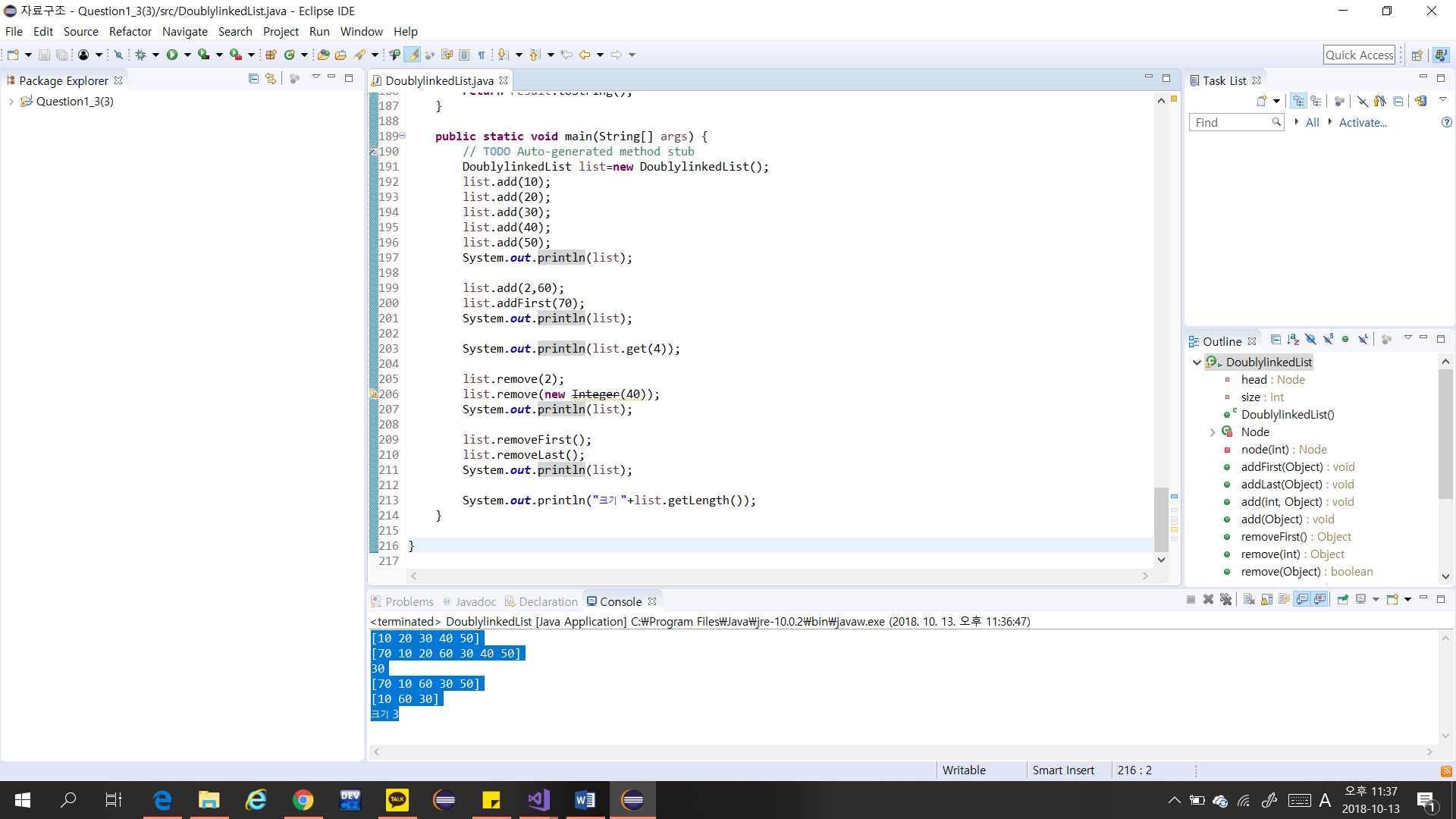
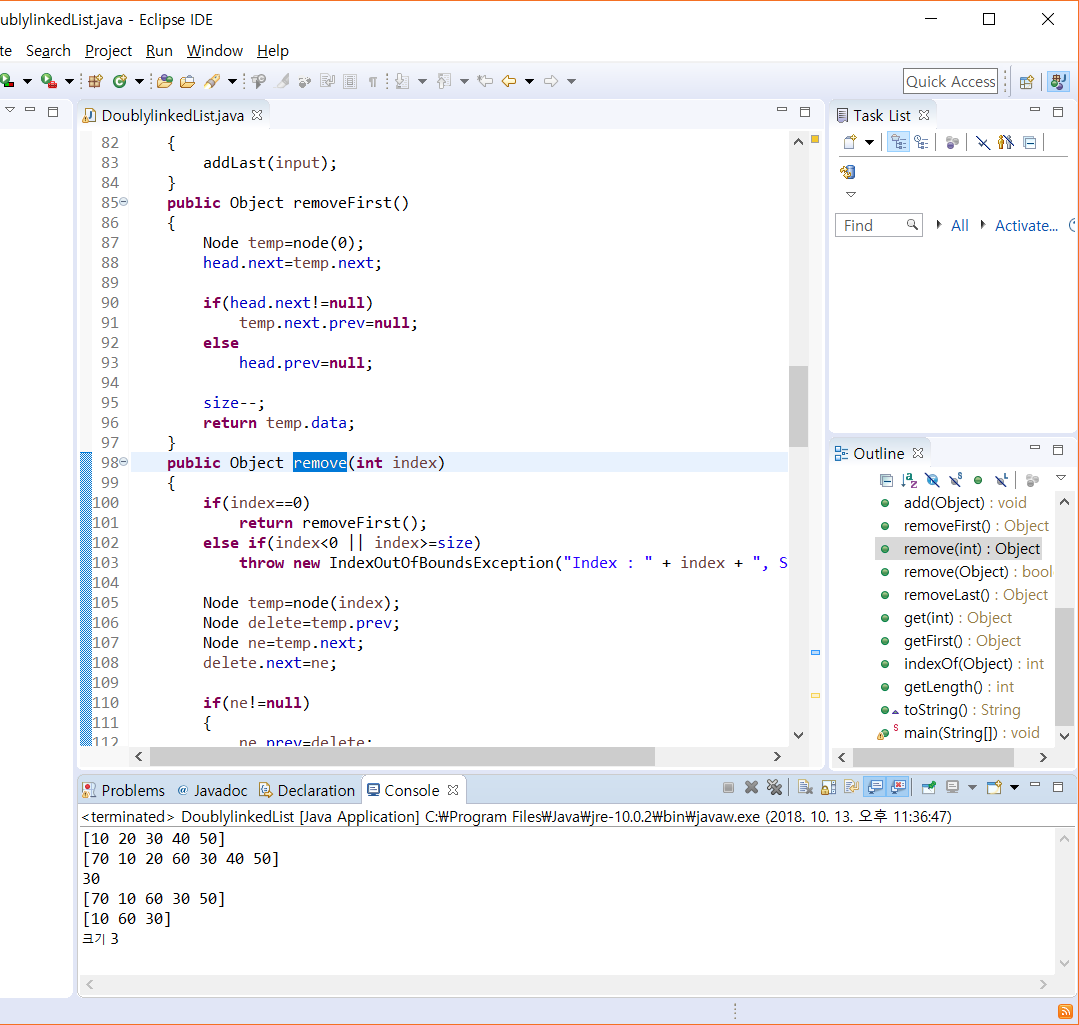
9) int getLength()

1.3.1의 Length()와 같은 기능을 하도록 리스트의 size를 반환하도록 만들었다.

10) String toString()

1.3.1의 print함수와 같은 기능으로 StringBuffer에 리스트에 있는 모든 데이터를 append하여 그것을 return 하도록 구현하였다.

**\*실행 결과**

-1.3.4 스택과 큐 구현: template 이중 연결 리스트 기반

*-stack*

**\*class 계층도**

genericStackDL.h

#include “genericStackDL.h” #include “genericStackDL.h”

stackClassTest.cpp

genericStackDL.cpp

#include “genericStackDL.cpp”

(template의 선언과 구현을 같이 나타내지 않으면 LNK2019 에러가 생긴다. 그래서 genericStackDL.cpp를 Test파일에 include하여 나타냈다.)

**\*구현 방법**

1.3.2에서 구현했던 template listClass를 이용하여 stackClass를 작성하였다.

1) stackClass(const stackClass<T>& S)

stackClass의 객체 S의 L필드는 S가 선언되는 순간에 생성되고 listClass의 객체는 그와 같으므로 L=S.L이라고 선언을 한다.

2) ~stackClass()

스택이 Empty일때까지 Pop를 하여 모든 데이터를 삭제하였다.

3) void Push(T Item)

리스트의 처음 위치를 스택의 탑으로 간주하므로 리스트의 처음 위치에 데이터를 삽입하는 것과 같다. 그러므로 listClass의 Insert함수를 이용하여 구현하였다.

4) void Pop()

스택의 Pop은 리스트의 첫 데이터를 삭제하는 것에 해당되므로 listClass의 Delete함수를 이용하여 구현하였다. 만약 빈 스택을 Pop할 때는 “Stack is empty”라는 문장을 출력하도록 하였다.

5) void Print()

스택의 모든 데이터를 출력하는 것이므로 listClass의 print함수를 이용하여 구현하였다.

6) void GetTop(T& Item)

스택의 탑을 읽는 함수이므로 리스트의 첫 데이터를 읽는 것과 같다. 그러므로 listClass의 Retrieve 함수를 이용하여 구현하였다.

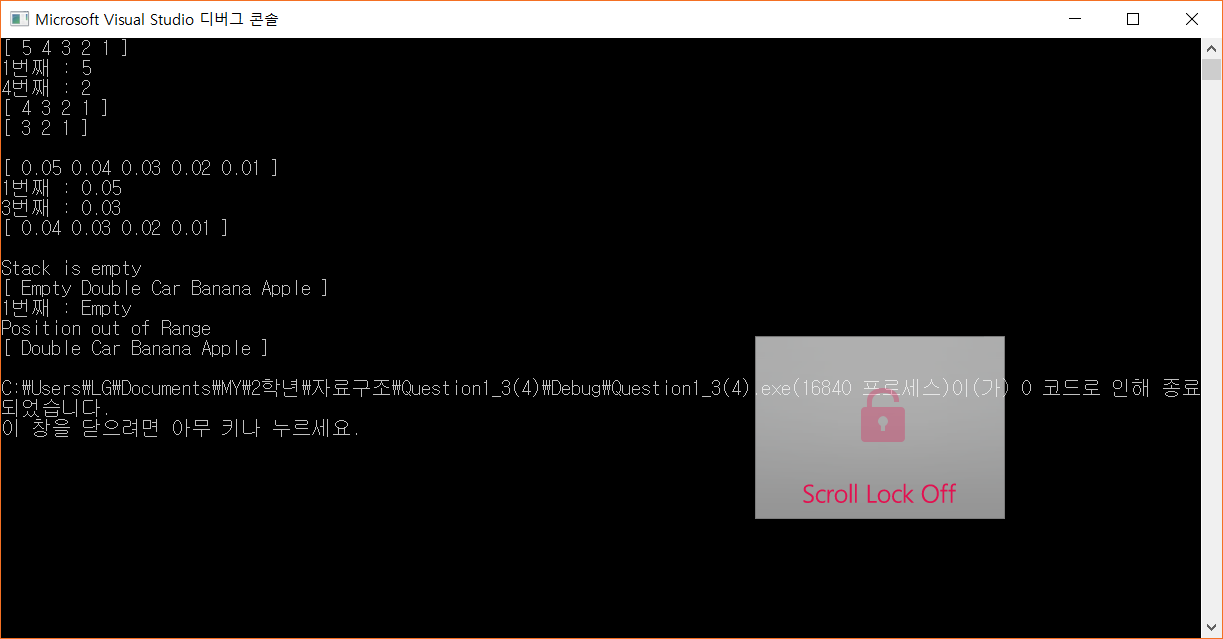
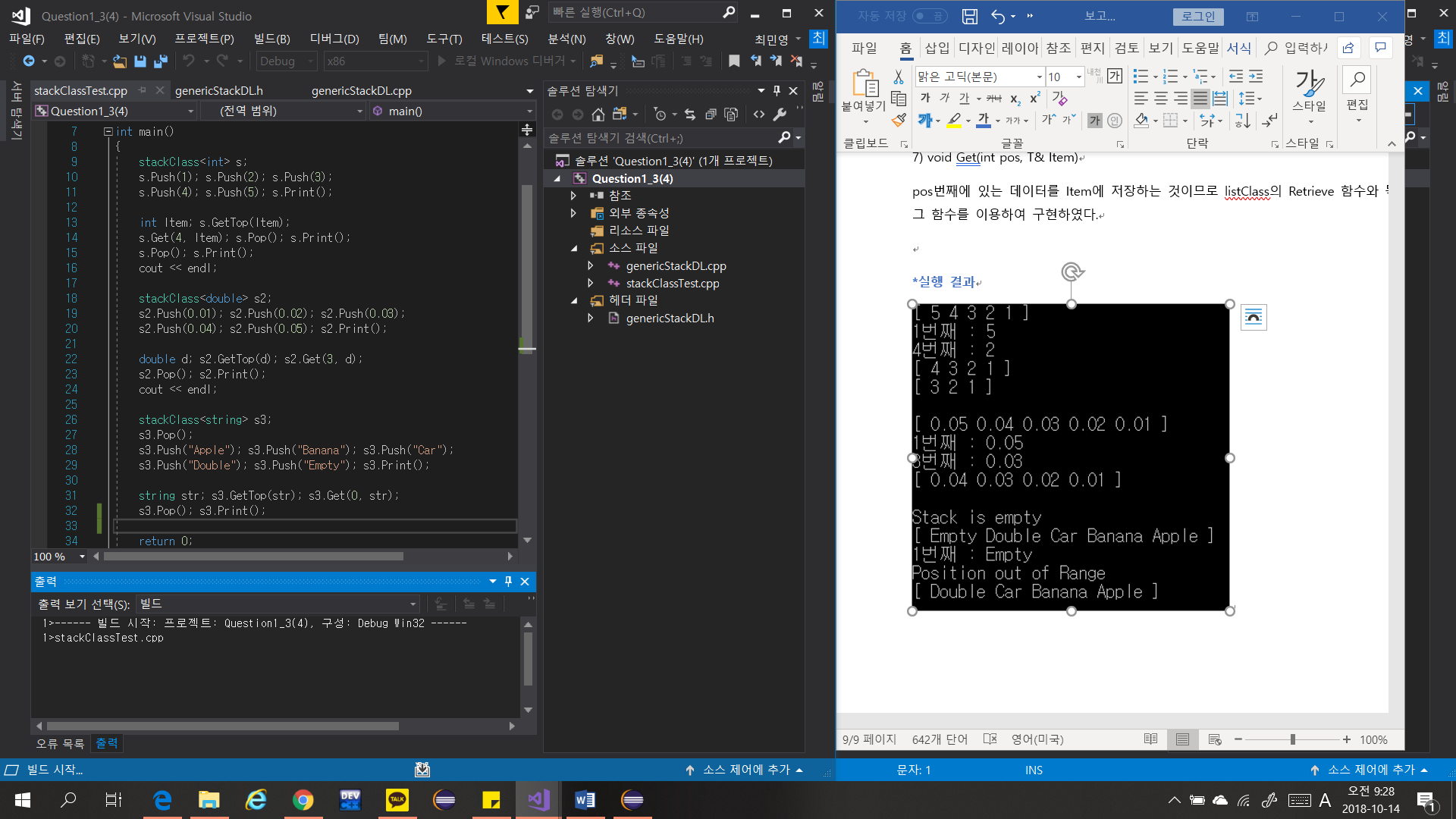
7) void Get(int pos, T& Item)

pos번째에 있는 데이터를 Item에 저장하는 것이므로 listClass의 Retrieve 함수와 똑같다. 그러므로 그 함수를 이용하여 구현하였다.

8) bool IsEmpty()

listClass의 isEmpty일때와 같은 결과이므로 return (L.isEmpty())를 작성하였따.

**\*실행 결과**



*-stack*

**\*class 계층도**

genericQueueDL.h

#include “genericQueueDL.h” #include “genericQueueDL.h”

queueClassTest.cpp

genericQueueDL.cpp

#include “genericQueueDL.cpp”

(위의 stack과 같은 이유로 genericQueueDL.cpp를 include 하였다.)

**\*구현 방법**

1.3.2에서 구현했던 template listClass를 이용하여 Queue를 작성하였다.

1) Queue(const Queue<T>& Q)

listClass의 객체 L과 Queue의 객체 Q의 L필드와 같으므로 L=Q.L이라고 선언한다.

2) ~Queue()

큐가 Empty일때까지 모든 데이터를 삭제해야 하므로 remove를 이용하여 없앤다.

3) void insert(T Item)

큐의 삽입은 Rear에서 일어나므로 리스트의 길이보다 1이 큰 자리에 데이터를 추가하는 것과 같다. 그러므로 listClass의 Insert함수를 이용하여 L.Insert(L.Length()+1, Item)으로 구현한다.

4) void remove()

큐의 삭제는 Front에서 일어나므로 listClass의 Delete함수를 이용하여 맨 앞의 데이터를 삭제한다.

5) void GetFront(T& Item)

큐의 Front를 보여주는 함수로 listClass의 Retrieve와 기능이 비슷하다. 그러므로 그 함수를 이용하여 Front의 데이터를 출력한다.

6) void Get(int pos, T& Item)

pos번째에 있는 데이터를 Item에 저장하는 것으로 listClass이 Retrieve와 기능이 같다. 그러므로 그 함수를 이용하여 구현한다.

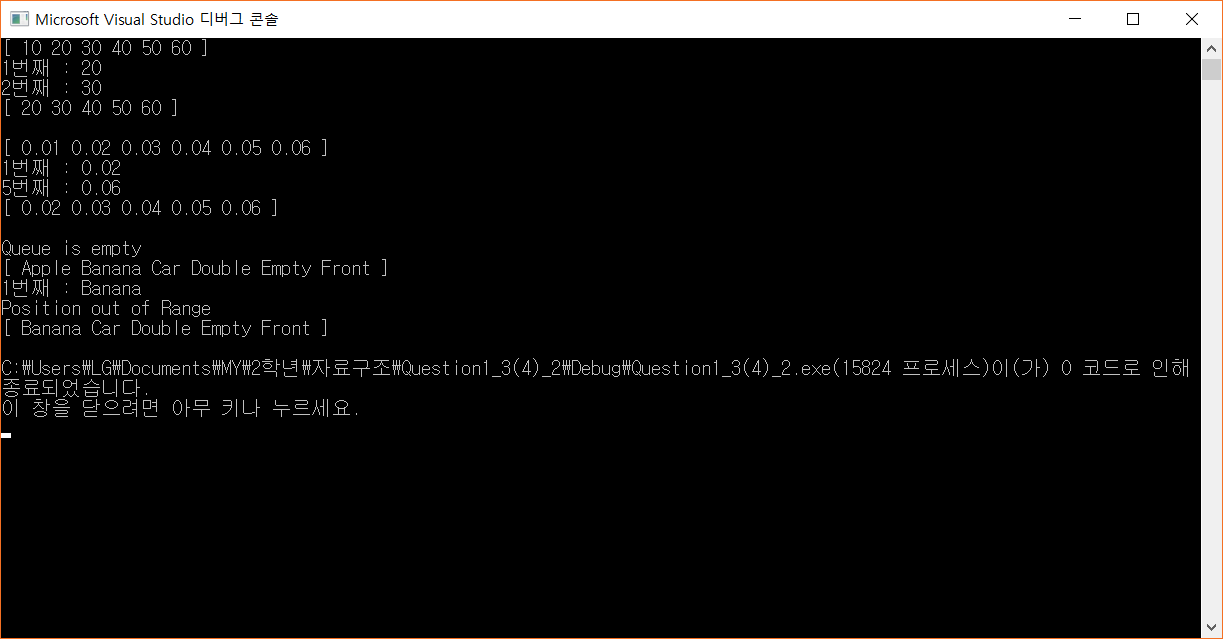
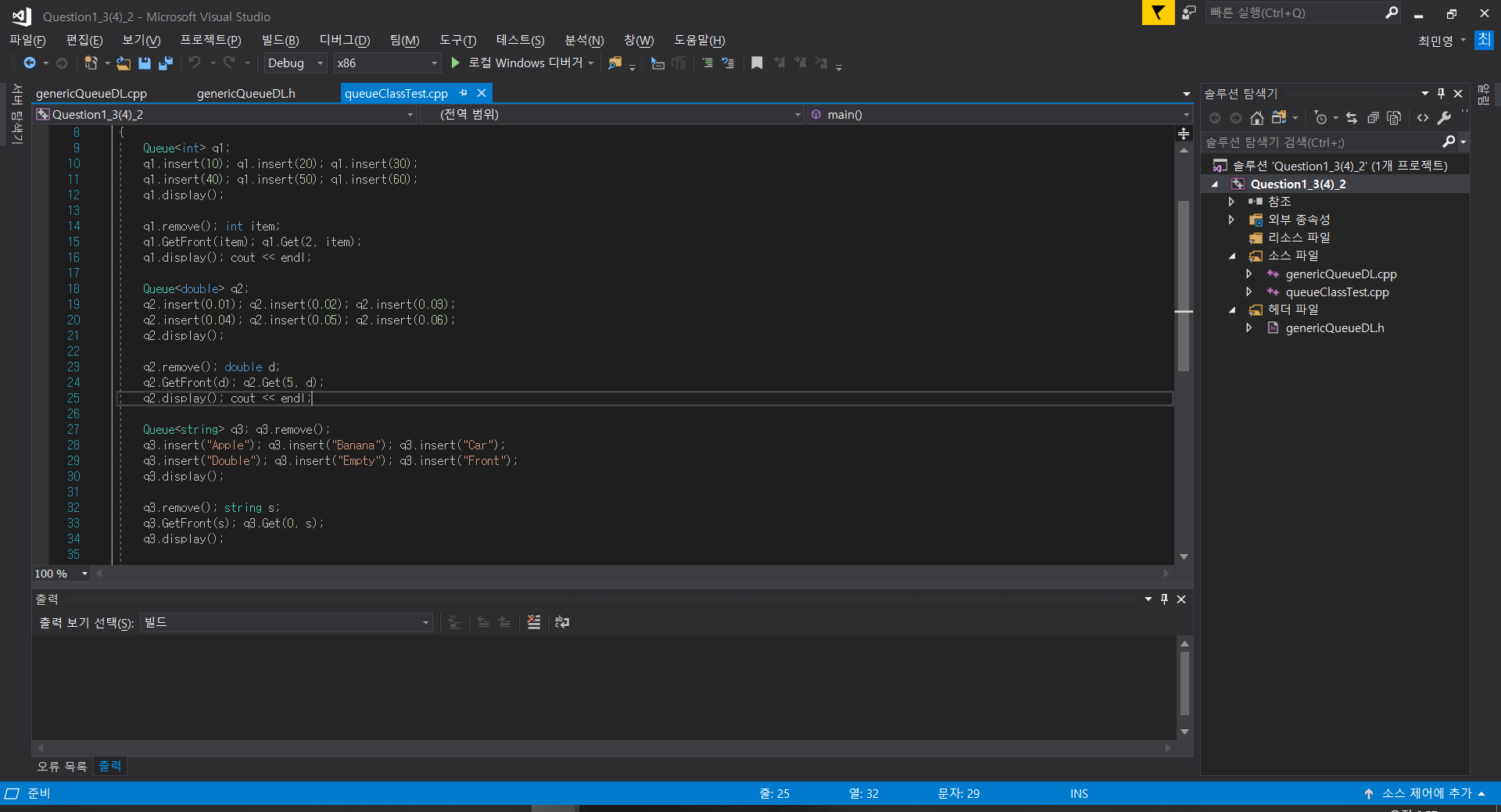
7) void display()

큐의 모든 데이터를 출력하는 함수로 listClass의 print함수를 이용하여 구현한다.

8) bool isEmpty()

listClass의 isEmpty함수의 결과와 같으므로 return (L.isEmpty)를 작성하였다.

**\*실행 결과**



**#1.4 산술식 평가: calculator구현**

**\*Pseudo-code**

Procedure result(tokens)

for i: tokens.length() do

If (number)

Push it

else if (Operator)

Pop operands

Do operation

Push result

else if (newline)

Pop and print Top of stack

else

error

End If

End for

**\*구현 방법**

1.3.4에서 구현하였던 stackClass을 이용하여 작성하였다.

1) int pre(char op)

연산자들의 우선순위를 판단하기 위해 작성한 함수이다. 우선순위가 높을수록 높은 숫자가 return 되도록 구현하였다.

2) double calculate(double a, double b, char op)

Switch를 사용하여 연산자에 따라 어떠한 연산이 시행되는지 return하는 것으로 구현하였다.

3) double result(string tokens)

입력한 token을 white space로 구분하여 연산자는 stackClass<string>에 피연산자는 stackClass<double>에 저장하고 계산하기 위해 구현한 함수이다. tokens의 끝까지 반복하여 만약 white space라면 continue를, ‘(‘라면 stackClass<string>에 push를 한다. 또한 number라면 stackClass<double>에 push를 한다. 만약 ‘)’라면 각각 top에 있는 데이터를 꺼내 calculate를 한 후 stackClass<double>에 push를 하고 사용한 연산자를 pop해준다. 모든 것이 끝난 후 stackClass<double>의 Top에서 데이터를 꺼내 return 해준다.

**\*실행 결과**

연산자를 stackClass<string>에 저장하여 계산하는 것을 제대로 구현하지 못했다. 실행시키면 연산자가 작동하지 않는다. (연산자만 출력하면 오류가 나온다.)

**#1.5 깊이 우선 탐색: depth-ﬁrst search 구현**

**\*구현 방법**

1.3.4에서 구현하였던 stackClass을 이용하여 작성하였다.

-class Graph

1) Graph(int size)

size만큼의 행과 열을 생성하여 2차원 배열을 만들어 0으로 초기화 시킨다.

2) ~Graph()

배열에 있던 모든 데이터들을 delete해서 소멸해주도록 구현하였다.

3) bool isConnected(int x, int y)

x와 y가 연결되었는지 판단하는 함수로 return (A[x-1][y-1]==1)을 작성하여 나타냈다.

4) void addEdge(int x ,int y)

x와 y가 연결된 것을 나타내는 함수로 x-1행의 y-1열과 y-1행의 x-1열을 1로 나타낸다.

5) void DFS(int start, int end)

start에서 end까지의 탐색 결과를 나타내는 함수로 stackClass<int> 객체와 방문했는지 나타내는visited배열을 생성한다. visited는 false로 초기화 시키고 start를 push하면 visited[start]을 true로 만든다. stackClass<int>가 비어 있는 스택이 아닐 때 동안 start에서 end까지 isConnected가 true이고 visited가 false이면 push를 하고 visited를 true로 지정하여 출력하도록 한다.

**\*실행 결과**

Path length와 path가 없을 때 not found를 return하는 것 그리고 파일로 받는 것에 실패하였다.

또한 직접 addEdge함수를 사용하여 연결을 시켜 아래의 사진과 같은 결과가 나왔지만 구동 시나리오와 다른 결과가 나왔다. (DFS함수가 제대로 구현이 되지 않았다.)

