

10

재귀 연결 리스트

목적

이번 실습에서 여러분은

- 재귀가 양 방향으로 연결 리스트 순회에 어떻게 사용되는지 조사한다.
- 리스트내에서 원소의 삽입, 삭제, 이동에 재귀를 사용한다.
- 반복적인 품으로 재귀적 루틴을 변경한다.
- 재귀를 반복적인 품으로 변경할 때 스택이 필요한 이유를 분석한다.

개요

재귀 함수 또는 자기자신을 호출하는 함수는 수학, 컴퓨터 그래픽, 컴파일러 설계, 인공지능의 문제 등을 포함한 넓은 범위의 문제의 해결책을 구현하고 서술하는 세련된 방법을 제공한다. 예로 factorial 함수를 사용하여 재귀 함수를 만드는 법을 조사하기로 한다.

여러분은 다음의 반복적인 공식을 사용하여 양수 n 의 factorial을 표현한다.

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

이 공식을 4!에 적용하여 $4 \times 3 \times 2 \times 1$ 의 생성물을 얻는다. 이 생성물의 관계를 $4 \times (3 \times 2 \times 1)$ 로 다시 조정한다면 $3! = 3 \times 2 \times 1$ 로 표시한다. 4!는 $4 \times (3!)$ 로 쓸수 있다. 여러분은 이 이유를 다음의 factorial의 재귀적인 정의 형태로 일반화할 수 있다.

$$n! = n \times (n-1)!$$

0!은 1로 정의한다. 이 정의를 4!에 적용하면 다음의 계산의 순서를 얻는다.

$$\begin{aligned} 4! &= 4 \times (3!) \\ 4! &= 4 \times (3! \times (2!)) \\ 4! &= 4 \times (3! \times (2 \times (1!))) \\ 4! &= 4 \times (3! \times (2 \times (1 \times (0!)))) \\ 4! &= 4 \times (3! \times (2 \times (1 \times (1)))) \end{aligned}$$

(n-1)!로 평가되는 n!는 갖는 이 계산에서 첫 번째 네 단계는 재귀적이다. 마지막 단계(0!=1)은 재귀적이지 않다. 다음의 표기는 명확히 재귀 단계와 비재귀 단계를 n!의 정의에서 구별한다.

$$n! = \begin{cases} 1 & \text{if } n=0 \text{ (base case)} \\ n(n-1) & \text{if } n>0 \text{ (recursive step)} \end{cases}$$

다음의 factorial() 함수는 factorial을 계산하는데 재귀(recursion)를 사용한다.

```
Long factorial (int n)
// Computes n! using recursion.
{
    long result;    // Result returned
    int (n == 0)
        result = 1;                // Base case
    else
        result = n * factorial(n-1); // Recursive step
}
```

factorial(4)를 호출하는 것을 살펴보자. 4는 0과(기본 경우의 상태) 같지 않기 때문에 factorial() 함수는 factorial(3)을 재귀적으로 호출한다. 재귀적인 호출은 기본적인 경우가 될 때까지 즉 n이 0이 될 때까지 계속된다.

```
factorial(4)
  ↓ RECURSIVE STEP
4*factorial(3)
  ↓ RECURSIVE STEP
3*factorial(2)
  ↓ RECURSIVE STEP
2*factorial(1)
```

↓ RECURSIVE STEP
1*factorial(0)
 ↓ BASE CASE
 1

factorial() 호출은 factorial들이 만든 순서의 반대로 평가된다. 평가 프로세스는 (evaluation process)는 factorial(4)가 24를 반환할 때까지 계속된다.

```

factorial(4)
  ↑ Result = 24
    4*factorial(3)
      ↑ Result = 6
        3*factorial(2)
          ↑ Result = 2
            2*factorial(1)
              ↑ Result = 1
                1*factorial(0)
                  ↑ Result = 1
                    1
  
```

재귀는 수 계산에 더 많이 사용된다. 다음의 함수는 연결 리스트의 원소를 만났을 때 그것을 출력하면서 순회한다.

```

Template <class LE>
void List<LE>::write ( ) const
// Outputs the elements in a list from beginning to end.
// Assumes that objects of type LE can be output to the
// cout stream.
{
    cout << "Mirror :";
    writeSub(head);
}

// -----
template <class LE>
void List<LE>::writeSub (ListNode<LE> *p) const

// Recursive partner of the write( ) function. Processes
// the sublist that begins with the node pointed to by p.
{
  
```

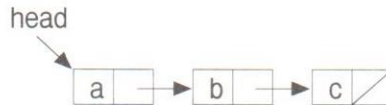
```

    if ( p != 0 )
    {
        cout << p->element; // Output element

        writeSub(p->next);    // Continue with next node
    }
}

```

write() 함수의 일은 재귀 작업(recursive process)을 초기화하고, 재귀 동반자인 writeSub() 함수에 의해 전방향으로 이동된다. write() 문자 연결 리스트에서 write() 를 호출하여



다음의 일련의 호출을 얻고 "abc" 를 출력한다.

```

WriteSub(head)
  ↓ RECURSIVE STEP
Output 'a' writeSub(p->next)
  ↓ RECURSIVE STEP
Output 'b' writeSub(p->next)
  ↓ RECURSIVE STEP
Output 'c' writeSub(p->next)
  ↓ BASE CASE
No output

```

재귀는 연결 리스트에 노드를 첨가하는데 사용되어질 수도 있다. 다음의 함수는 리스트의 끝에 원소를 삽입한다.

```

Template <class LE>
void List <LE>::insertEnd (const LE &newElement)

// Insert newElement at the end of a list. Moves the cursor
// to newElement.

{
    insertEndSub (head, newElement);
}

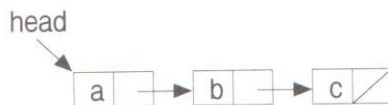
```

```
// -----

template <class LE>
void List<LE>::insertEndSub (ListNode<LE> *p,
                           const LE &newElement)
// Recursive partner of the insertEnd( ) function.
// Processes the sublist that begins with the node pointed
// to by p.

{
    * if (p != 0)
        insert?EndSub(p->next, newElement); // Continue
                                              // searching for
    else                                     // end of list
    {
        p = new ListNode <LE>(newElement, 0); // Insert new
                                              // node
        cursor = p;                          // Move cursor
    }
}
```

insertEnd() 함수는 대부분 재귀 동반자 insertEndSub() 함수에 의해 이루어 지는 작업을 갖는 삽입 작업을 초기화한다. 문자 '!' 리스트의 끝에 삽입하는 InsertEnd() 함수 호출은



다음의 일련의 호출을 얻는다.

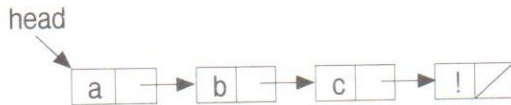
```
insertEndSub (head)
  ↓ RECURSIVE STEP
insertEndSub (p->next)
  ↓ RECURSIVE STEP
insertEndSub (p->next)
  ↓ RECURSIVE STEP
insertEndSub (p->next)
  ↓ BASE CASE
```

문자 '!'를 갖는 새로운 노드를 만든다.

마지막 호출에서 p는 널(null)이고 문장은

```
p = new ListNode<LE>(newElement, 0); // Insert new node
```

문자 '!'를 갖는 새로운 노드 생성하기 위해 실행된다. 이 노드의 주소는 p에 할당된다. p는 call by reference를 사용하여 지나치기(pass) 때문에 이 할당은 리스트의 마지막 노드의 next 포인터를 새로운 노드를 가리키게 변경한다. 따라서 다음의 리스트를 생성한다.



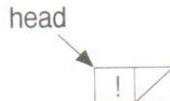
빈 리스트에 문자 '!'를 삽입하기 위해 insertEnd()를 호출하면 insertEndSub() 함수를 한번 호출하게 된다.

```
insertEndSub(head)
```

↓ RECURSIVE STEP

'!'을 포함하는 새로운 노드를 생성하라.

이 경우에 새로 생성된 노드의 주소를 p에 할당하는 것은 리스트의 헤드 포인터를 이 노드를 가리키게 한다.



insertEnd() 함수는 이 next 포인터 또는 리스트의 헤드 포인터를 변경하는지 결정하는 특별한 시험 없이 자동적으로 존재하는 리스트나 빈 리스트로의 노드 연결을 생성하는 것을 주목하여야. 핵심은 매개변수 P를 참조적 호출을 이용해서 전달한다는 것이다.

실습 10 : 표지

날짜_____ 구분_____

성명_____

강사가 여러분에게 할당한 다음에 연습문제들에 할당된 열에 체크표시를 하고 이 결장을 앞으로의 실습에서 여러분이 제출할 과제물 앞에 붙이시요

	할 일	완 료
실습 전 연습	✓	
연 결 연습	✓	
실습 중 연습 1		
실습 중 연습 2		
실습 중 연습 3		
실습 후 연습 1		
실습 후 연습 2		
	총 점	

실습 10 : 실습 전 연습

날짜 _____ 구분 _____
성명 _____

우리는 재귀함수의 집합을 검사하는 것으로 작업의 수행을 시작했다. 이러한 함수들은 파일 *listrec.cs*에 모아 놓았다. 여러분은 이 함수들을 파일 *test10.cpp*에 있는 시험 프로그램을 사용하여 실행할 수 있다.

PART A

1단계 : 이번 실습을 완료하기 위해 여러분은 여러분의 List ADT 단일 연결 리스트 구현에서 약간의 함수를 사용할 필요가 있다. 여러분이 실습 7에서 개발한 연결 리스트 구현에서 다음의 함수를 추가하여 파일 *listrec.cs*에 있는 List ADT의 부분적 구현을 완성한다.

- ListNode class에 대한 객체 생성자
- List class 객체 생성자 객체 파괴자, insert(), clear(), showStructure() 함수들

이들 함수에 관한 프로토타입은 파일 *listrec.h*에 있는 List class의 선언에 포함되어 있다.

2단계 : 파일 *listrec.cpp*에 구현의 결과를 저장하라

3단계 : 파일 *test10.cpp*에서 "//PA"로 시작하는 행에서 주석 경계자와 문자 "PA"를 제거하여 write() 함수와 insertEnd() 함수의 호출을 활성화하라

4단계 : write()와 insertEnd() 함수를 다음의 리스트를 이용하여 실행하라.



5단계 : write()는 어떤 결과를 생성하는가?

6단계 : insertEnd()는 어떤 리스트를 생성하는가 ?

7단계 : 빈 리스트를 이용하여 이 함수들을 실행하라.

8단계 : write()는 어떤 결과를 생성하는가 ?

9단계 : insertEnd()는 어떤 리스트를 생성하는가 ?

PART B

여러분이 연결 리스트를 갖는 재귀적 용법을 사용하는 일반적인 이유는 리스트의 끝에서 처음으로 반대로 순회하는 것을 지원하는 것이다. 다음의 함수들은 처음에서 끝까지 순회한 리스트와 다시 끝에서 거꾸로 처음으로 순회하는 각 리스트의 원소를 두 번씩 출력한다.

```

Template <class LE>
void List <LE>::writeMirror ( ) const

// Outputs the elements in a list from beginning to end and
// back again. Assumes that objects of type LE can be
// output to the cout stream

{
    cout << "Mirror:";
    writeMirrorSub(head);
    cout << endl;
}

// - - - - -

template <class LE>
void List <LE>::writeMirrorSub (ListNode<LE> *p) const

// Recursive partner of the writeMirror( ) function.
// Processes the sublist that begins with the node pointed
// to by p.

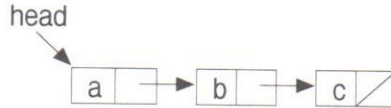
{
    if ( p != 0 )
    {
        cout << p->element;           // Output forward
        writeMirrorSub (p->next);      // Continue with
                                         // next node
        cout << p->element;           // Output backward
    }
}

```



1단계 : 파일 `test10.cpp`에 있는 시험 프로그램의 `writeMirror()` 함수의 호출을 "PB"로 시작하는 행에서 주석 경계자와 문자 PB를 제거하여 활성화하라.

2단계 : `writeMirror()` 함수를 다음의 리스트를 이용하여 실행하라.



3단계 : `writeMirror()`은 어떤 결과를 생성하는가?

4단계 : `writeMirrorSub()` 함수에서 매개변수 `p`가 'a'를 갖는 노드를 가리키는 호출 동안 무엇을 하는지 서술하라.

5단계 : 매개변수 `p`가 널(null)인 `writeMirrorSub()`를 호출하는 중요성은 무엇인가?

6단계 : "mirrored" 결과를 생성하도록 `writeMirrorSub()` 호출을 조합할 수 있는지 설명하라. 답을 설명하는데 다이어그램을 사용하라.

PART C

다음의 함수는 각 노드의 next 포인터를 변경하여 리스트의 방향을 바꾼다. 리스트의 반대 방향으로 포인터들이 바뀐다는 것을 주의하라.

```

Template <class LE>
void List <LE>::reverse ( )

    // Reverses the order of the elements in a list.

{
    reverseSub(0, head);
}
// -----

template <class LE>
void List <LE>::reverseSub (ListNode<LE> *p,
                           ListNode<LE> *nextP)

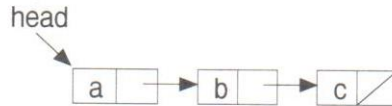
    // Recursive partner of the reverse( ) function. Processes
    // the sublist that begins with the node pointed to
    // by nextP.

{
    if (nextP != 0)
    {
        reverseSub(nextP, nextP->next); // Continue with
                                         // next node
        nextP->next = p;                 // Reverse link
    }
    else
        head = p; // Move head to end of list
}

```

1단계 : 시험 프로그램에서 "//PC"로 시작하는 행에서 주석 경계자와 문자 'PC'를 제거하여 reverse() 함수 호출을 활성화하라.

2단계 : reverse() 함수를 다음의 리스트를 이용하여 실행하라.



3단계 : `reverse()` 는 어떤 리스트를 생성하는가 ?

4단계 : 매개변수 `P`가 'a'를 갖는 노드를 가리키는 호출을 하는 동안 `reverseSub()` 함수의 각 단계를 서술하라. 특히 호출의 결과로 어떻게 링크되나?

5단계 : 매개변수 `p`가 널(null)인 `reverseSub()` 호출에서 중요한 것은 무엇인가 ?

6단계 : 리스트의 방향을 바꾸기 위해 어떻게 `reverseSub()`를 호출하는지 서술하라. 답을 설명하는데 다이어그램을 이용하라.

PART D

서론에서 리스트의 끝에 노드를 삽입하는데 call by reference와 결합한 재귀적 용법의 사용 방법을 알아 보았다.

```

Template <class LE>
void List <LE>::deleteEnd ( )

// Deletes the element at the end of a list.
// Moves the cursor to the beginning of the list.

{
    deleteEndsub(head);
    cursor = head;
}
// -----

template <class LE>
void List <LE>::deleteEndSub (ListNode<LE> *&)

// Recursive partner of the deleteEnd( ) function.
// Processes the sublist that begins with the node
// pointed to by p.

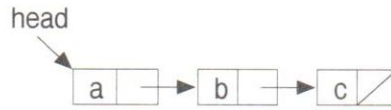
{
    if (p ->next != 0)
        deleteEndSub(p->next); // Continue looking for the
                                // last node

    else
    {
        delete p;           // Delete node
        p = 0;              // Set p (link or head) to null
    }
}

```

1단계 : 시험 프로그램에서 "//PD"로 시작하는 행에서 주석 경계자와 문자 'PD'를 삭제하여 **deleteEnd()** 함수 호출을 활성화하라.

2단계 : 다음의 리스트를 이용하여 **deleteEnd()**를 실행하라.



3단계 : **deleteEnd()**는 어떤 리스트를 생성하는가?

4단계 : **p->next**가 널이 아닌 **deleteEndSub()**의 호출에서 중요한 것은 무엇인가?

5단계 : **p->next**가 널인 호출 동안 **deleteEndSub()**의 각 상태를 설명하라. 다이어그램을 이용하여 답을 설명하라.

6단계 : 하나의 원소를 갖는 리스트에서 호출되었을 때 **deleteEnd()**는 어떤 리스트를 생성하는가? 수행의 결과를 설명하라. 답을 설명할 때 다이어그램을 이용하라.

PART E

다음의 함수는 리스트의 길이를 결정한다. 이들 함수는 리스트에서 처음에서 끝으로 이동할 때 단순히 노드의 갯수를 세지 않는다(반복적인 함수일 경우도) 대신 $p \rightarrow \text{next}$ (리스트에 남아 있는 노드)에 더하기 '1'한 것(p 에 의해서 지정된 노드)으로 리스트의 길이를 나타내는 것처럼 포인터 P 에 의해서 지시된 리스트의 길이를 측정하는 길이에 대한 재귀적인 정의를 사용한다.

$$\text{length}(p) = \begin{cases} 0 & \text{if } p = 0 \text{ (base case)} \\ \text{length}(p \rightarrow \text{next}) + 1 & \text{if } p \neq 0 \text{ (recursive step)} \end{cases}$$

```
template <class LE>
int List <LE>::length ( ) const

// Returns the number of elements in a list

{
    return lengthSub(head);
}

// -----

template <class LE>
int List <LE>::lengthSub (ListNode <LE> *p) const

// Recursive partner of the length() function. Processes
// the sublist that begins with the node pointed to by p.

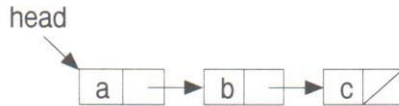
{
    int result; // Result returned
    if ( p == 0 )
        result = 0; // End of list reached
    else
        result = ( lengthSub(p->next) + 1 ); // Number of
                                                // nodes after
                                                // this one + 1

    return result;
}
```

1단계 : 시험 프로그램에서 "PE"로 시작하는 행에서 주석 경계자와 문자 'PE'를 제

거하여 `length()` 함수 호출을 활성화하라.

2단계 : 아래의 리스트를 이용하여 `length()`를 실행하라.



3단계 : `length()`의 결과는 어떻게 되는가?

4단계 : 매개변수 `p→next`가 null인 `lengthSub()`의 호출에서 중요한 것은 무엇인가?

5단계 : 리스트의 길이를 반환하기 위한 `lengthSub()`를 호출하는 방법을 설명하라.
다이어그램을 이용하여 설명하라.

6단계 : 빈 리스트에서 호출되었을 때 `length()` 함수는 어떤 값을 반환하는가? 이 값이 계산되는 방법을 설명하라. 여러분의 답을 다이어그램을 이용하여 설명하라.

실습 10 : 연결 연습

날짜 _____ 구분 _____
이름 _____

강사와 함께 실습 기간 전이나 실습 기간 동안 이 연습을 마칠 수 있는지 점검하라.

PART A

다음의 두 개의 함수는 지정하지 않은 동작을 수행한다.

```

Template <class LE>
void List <LE>::unknown1 ( ) const

// Unknown function 1.

{
    unknown1Sub (head);
    cout << endl;
}

// -----

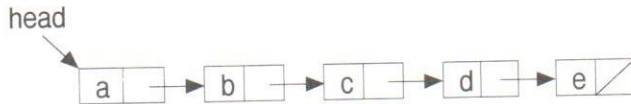
template <class LE>
void List <LE>::unknown1Sub (ListNode <LE> *p) const

// Recursive partner of the unknown1 ( ) function.

{
    if (p != 0)
    {
        cout << p->element;
        if(p->next != 0)
        {
            unknown1Sub(p->next->next);
            cout << p->next->element;
        }
    }
}

```

- 1단계** : 파일 `test10.cpp`의 시험 프로그램에서 "//BA"로 시작되는 행에서 주석 경계자와 문자 'BA'를 제거하여 `unknown1()` 함수 호출을 활성화하라.
- 2단계** : 다음의 리스트를 이용하여 `unknown1()` 함수를 실행하라.



- 3단계** : `unknown1()`의 결과는?
- 4단계** : 매개변수 `p`가 'a'를 갖는 노드를 가리키는 호출 동안 `unknown1Sub()` 함수의 각 상태를 설명하라.
- 5단계** : 리스트를 출력하도록 `unknown1Sub()`의 호출을 조합하는 방법을 설명하라. 다이어그램을 사용하여 설명하라.

PART B

다음의 함수는 아직도 지정하지 않은 행동을 수행한다.

```

Template <class LE>
void List <LE>::unknown2 ( )

// Unknown function 2.

{
    unknown2Sbu (head);
}

// -----

template <class LE>
void List <LE>::unknown2Sub (ListNode<LE> *p)

// Recursive partner of the unknown2 ( ) function.

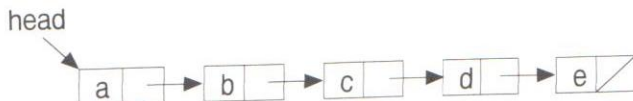
{
    ListNode<LE> *q;

    if (p != 0 && p->next != 0)
    {
        q = p;
        p = p->next;
        q->next = p->next;
        p->next = q;
        unknown2Sub (q->next);
    }
}

```

1단계 : 시험 프로그램에서 "//BB" 시작하는 행에서 주석 경계자와 문자 'BB'를 삭제하여 unknown2() 함수 호출을 활성화하라.

2단계 : 아래의 리스트를 이용하여 unknown()함수를 수행하라.



3단계 : unknown2()는 어떤 리스트를 생성하는가 ?

4단계 : 매개변수 p가 'a'를 갖는 노드를 가리키는 호출 동안 unknown2Sub() 함수의 각 상태를 서술하라. 특히 이 호출에서 call by reference를 이용하여 전송(pass)하는 p는 어떤 작업을 하는가?

5단계 : 리스트를 재구성하는데 unknown2Sub() 함수 호출을 조합하는 방법을 서술하라. 다이어그램을 이용하여 설명하라.

실습 10 : 실습 중 연습 1

날짜 _____ 구분 _____
 성명 _____


비록 재귀적 용법이 알고리즘을 표현하는 직관적인 방법이라도 재귀를 반복으로 대체하려는 경우가 있다. 이런 경우는 프로그램의 실행에 있어 특별한 재귀 루틴이 메모리 사용, 시간측면에서 오버헤드를 갖는 경우 대개 반복으로 대체한다.


PART A

`length()`(실습 전 연습 Part E) 함수처럼 루틴에서 재귀를 교체하는 것은 쉽다. 리스트에서 이동을 호출하는 재귀를 사용하는 것이 아니라 여러분은 노드에서 노드로 포인터(type `ListNode*`의 포인터)를 이동시킨다. `Length()` 함수의 경우 이 반복적인 프로세스는 리스트에 도달할 때까지 계속한다.

`reverse()` 함수(실습 전 연습 Part C)는 더 고무적인 문제를 보여준다. 이 루틴의 반복적인 품은 리스트에서 조정된 방법(coordinated manner)에 따라 포인터들의 집합을 이동시킨다. 이 포인터들을 리스트에서 이동시키면 포인터들은 노드들의 사이의 링크들의 순서를 바꾼다.

1단계 : 포인터들의 작은 집합과 연관되고 재귀를 대신하는 반복을 사용하는 `reverse()` 함수의 구현을 작성하라. 이 `iterReverse()` 함수를 호출하고 파일

 `listrec.cpp`에 추가하라. 이 함수에 관한 프로토타입은 파일 `listrec.h`에 `List class`의 선언에 있다.

 **2단계 :** 파일 `test10.cpp`에서 `"//1A"`로 시작되는 행에서 주석 경계자와 문자 '1A'를 제거하여 함수 `iterReverse()` 함수 호출을 활성화하라.

3단계 : 단일 원소를 갖는 리스트를 포함해서 다른 길이들의 리스트를 모두 다루는 `iterReverse()` 함수에 대한 시험 계획을 준비하라. 시험 계획 품은 다음과 같다.

4단계 : 여러분의 시험 계획을 수행하라. `iterReverse()` 함수에 잘못이 있으면 수정하고 다시 실행하라.

iterReverse () 함수에 대한 시험 계획			
시험 항목	리스트	예상 결과	검사

PART B

writeMirror() 함수(실습 전 연습문제, Part B)는 보다 고무적인 표현이다. 이 루틴은 인터랙티브한 형태의 리스트 노드들의 포인터를 저장하는데 스택을 사용한다. 이 스택은 다음 형태의 인터랙티브 프로세서에 사용된다.

```

stNode <LE> *p;                                // Iterates through list

Set p to the head of the list.                  // Traverse list from
while (p!= 0) do                                // beginning to end
{
    tempStack.push(p);
    Process the list node pointed to by p (if necessary).
    Advance p to the next node in the list.
}

while (!tempStack.empty( )) do                  // Traverse list from
{                                                // beginning to end
    p = tempStack.pop( );
    process the list node pointed to by p.
}

```

1단계 : 재귀를 대신해서 스택과 함께 반복을 사용하는 **writeMirror()** 함수의 구현을 작성하라. 함수 **stackWriteMirror()**을 호출하고 파일 **listrec.cpp**에 추가하라. 이 함수에 대한 프로토타입은 파일 **listrec.h**에 List class의 선언에 있다. 여러분의 실습 5에서의 stack ADT 구현을 기본으로 **stackWriteMirror()** 함수를 작성하라.

2단계 : "//1B"로 시작되는 행에서 주석 경계자와 문자 '1B'를 제거하여 시험 프로그램에 있는 함수 **stackWriteMirror()** 호출을 활성화하라.

3단계 : 단일 원소를 갖는 리스트를 포함해서 다른 길이를 갖는 리스트를 모두 다

루는 `stackWriteMirror()` 함수에 대한 시험 계획을 준비하라. 시험 계획은 아래와 같다.

4단계 : 시험 계획을 실행하고 `stackWriteMirror()` 함수에 잘못이 있으면 수정하고 다시 실행하라.