Data Structures

Lab # 08

Data & Knowledge Engineering Laboratory
Kyung Hee University, Korea

Chapter 7 Programming Exercise

Lab 08

- 1. Exercise 12
- 2. Exercise 13
- 3. Exercise 14
- 4. Exercise 16
- 5. Exercise 21

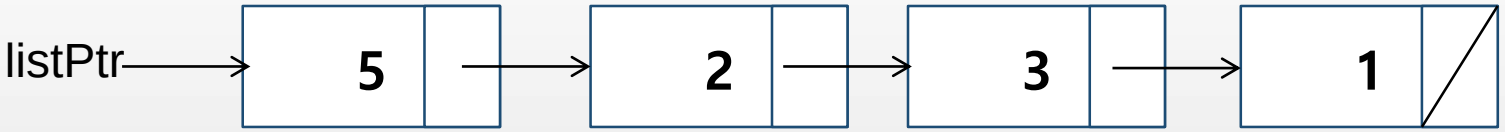
실습 준비

- **1번,5번 문제**
  - Chapter 5의 Linked List의 구현 소스중 UnsortedType.h를 사용
  - ₩student₩labplus₩lab,c++3rd₩Chapter5₩ListLL

1. Exercise 12

■ 문 제

- 학생들에게 프로그래밍 수업에 대한 학점을 부여해야 한다. 그래서 바로 학생들에게 재귀 함수에 대해 설명하고 다음과 같이 간단한 과제를 제시한다. 정수를 포함하는 순환 리스트의 포인터를 인자로 가지며 리스트의 각 요소들의 제곱 합을 구하는 SumSquares 재귀 함수를 작성하라.
- 예 :



  - 위의 경우 SumSquares(listPtr)은 (5*5)+(2*2)+(3*3)+(1*1)에 대한 결과로 39를 리턴
  - 가정 : 리스트가 비어있지 않음
- 데이터를 넣은 후 문제의 a, b, c, d, e의 SumSquares가 각각을 동작하도록 잘못된 부분을 수정한후 결과물을 출력하고 에러가 발생한 이유에 대해서 기술하시오.

1-help slide

```
template <class ItemType>
ItemType SumSquares_a(NodeType<ItemType>* list)
{
    return 0;
    if (list != NULL)
        return (list->info*list->info)+SumSquares(list->next);
}
template <class ItemType>
ItemType SumSquares_b(NodeType<ItemType>* list)
{
    int sum = 0;
    while(list!=NULL)
    {
        sum = list->info +sum;
        list = list->next;
    }
    return sum;
}
template <class ItemType>
ItemType SumSquares_c(NodeType<ItemType>* list)
{
    if(list == NULL)
        return 0;
    else
        return list->info*list->info + SumSquares(list->next);
}

template <class ItemType>
ItemType SumSquares_d(NodeType<ItemType>* list)
{
    if(list->next == NULL)
        return list->info*list->info;
    else
        return list->info*list->info+SumSquares(list->next);
}
template <class ItemType>
ItemType SumSquares_e(NodeType<ItemType>* list)
{
    if(list == NULL)
        return 0;
    else
        return (SumSquares(list->next)*SumSquares(list->next));
}
```

1-help slide

클라이언트 샘플 코드

```
int main()
{
        int item1 = 1;
        int item2 = 3;
        int item3 = 2;
        int item4 = 5;

        UnsortedType<int> list;

        list.InsertItem(item1);
        list.InsertItem(item2);
        list.InsertItem(item3);
        list.InsertItem(item4);

        list.PrintSumSquares();

        return 0;
}
```

UnsortedType.h

```
template <class ItemType>
class UnsortedType
{
public:
        void PrintSumSquares();
        …
}

template <class ItemType>
void UnsortedType<ItemType>::PrintSumSquares()
{
   //a, b, c, d, e의 문제를 수정 후 출력
   cout << SumSquares_a(listData) << endl;  //  a 번 함수
   cout << SumSquares_b(listData) << endl;  //  b 번 함수
   cout << SumSquares_c(listData) << endl;  //  c 번 함수
   cout << SumSquares_d(listData) << endl;  //  d 번 함수
   cout << SumSquares_e(listData) << endl;  //  e 번 함수
};

//함수 정의 및 문제점 수정, 클래스 멤버함수 가 아니다.
template <class ItemType>
ItemType SumSquares_a(NodeType<ItemType>* list)
{
        //a번 문제의 내용 수정
}

… 다른 문제 역시 위처럼 구현, 수정.
```

2. Exercise 13

■ 문 제

- ❖ 피보나치 수열(Fibonacci sequence)은 다음과 같은 정수의 연속이다.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...
- ❖ 이 정수의 연속은 현 시점에서 이전의 두 수에 대한 합이 된다. 수열의 n번째 수를 계산하는 재귀식은 다음과 같다. 단 n일 때 Fib(0)는 0이다.
- ❖ Fib(N) = N (N=0 혹은 1일 때), Fib(N-2)+Fib(N-1) (N이 1보다 클 때)

- ❖ A. Fibonacci 함수를 재귀 버전으로 구현하시오.
- ❖ B. Fibonacci 함수를 비 재귀 버전으로 구현하시오.
- ❖ C. 재귀 버전이 더 효율적으로 성능을 향상시킨다고 생각하는가?
  - 답을 기록하고 답에 대한 근거를 테스트 드라이버의 상단에 주석으로 기록

2-help slide

클라이언트 코드 예제

```
#include <iostream>
using namespace std;

int Fibonacci_recursive(int n);
Int Fibonacci_non_recursive(int n);

int main ()
{
		cout << Fibonacci_recursive(10) << endl;
		cout << Fibonacci_non_recursive(10) << endl;

		return 0;
}
```

3. Exercise 14

■ 문 제

❖ 다음은 주어진 근사치 답(approx)과 지정된 허용치(tol) 내의 수에 대한 제곱근 근사치를 계산하는 함수이다.

```
SqrRoot(number, approx, tol)
{
        |approx^2-number|<=tol 일 때
                approx
        |approx^2-number|>tol 일 때
                SqrRoot(number, (approx^2+number)/(2*approx),tol)
}
```

변수 설명
•number는 제곱근을 구하고자 하는 수

•approx는 제곱근의 근사치입니다. 계산을 반복할 수록 제곱근에 근접합니다. 함수 호출 시 초기 조건으로 이 값이 들어가야 하는데(예를 들어 1과 같은) 어떠한 값이 들어가야 할지 생각해 보세요.

•tol은 얼마만큼 제곱근에 정확한지 그 범위를 지정합니다. 작은 수(소수 이하의 수)를 입력할 수록 정확해 집니다.

- A. 재귀 버전으로 SqrRoot_recursion 함수를 구현하시오.
- B. 비재귀 버전으로 SqrRoot_non_recursion 함수를 구현하시오.

3-help slide

함수 동작의 예제:
2의 제곱근을 구하기. (고등학교 시간에 1.414라고 외웠습니다.)
초기 조건을 다음과 같이 준다고 가정합니다.
number:2, approx:1, tol:0.1

먼저 조건문 |approx^2-number| 을 계산.
|1*1 – 2 | = 1
이 값은 tol에 지정한(0.1) 정확도보다 높습니다.
근사치 계산을 수행합니다.

↓

(approx^2+number)/(2*approx)
(1*1+2)/(2*1) = 3/2 = 1.5
첫 번째 근사치가 나왔습니다.
정확도를 비교해 봅시다.

↓

|approx^2-number|
|1.5*1.5 – 2| = 2.25 – 2 = 0.25 (>tol)
아직은 원하는 만큼의 정확도가 아닙니다.
다시 계산 합니다.

↓

(approx^2+number)/(2*approx)
(1.5*1.5 + 2)/(2*1.5) = (2.25 + 2) / 3 = 1.416666…
두 번째 근사치가 나왔습니다.
정확도를 비교해 봅시다.

↓

|approx^2-number|
|1.416*1.416 – 2| = 2.005 – 2 = 0.005
만족할만한 정확도를 가지는 값을 구했습니다.

↓

계산한 approx (1.416)을 리턴합니다.

3-help slide

문제에 제시된 절대값은 fabs() 함수를 사용

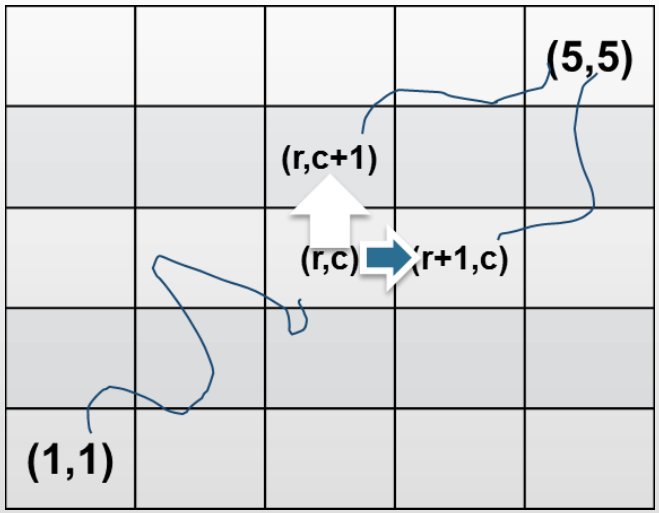
- #include <cmath> 추가

```
#include <cmath>

//main 함수 작성. 메인에서 함수들을 호출.
(SqrRoot를 클래스로 만들어서 사용하셔도 됩니다.)

//재귀 버전. 문제에 적힌 내용을 적으시면 됩니다.
float SqrRoot_recursion(float number, float approx, float tol)
{
		if(fabs(approx*approx - number) <= tol)
			return approx;
		else
			return SqrRoot_R(number, (approx*approx + number) / (2 * approx), tol);
}


//비재귀 버전. 재귀 버전에서 반복적으로 계산하는 부분을 조건에 맞춰 돌리면 됩니다.
float SqrRoot_non_recursion(float number, float approx, float tol)
{
		while(fabs(approx*approx - number) > tol)
		{
			// (approx*approx + number)계산 수행
		}
		return approx;
}
```

4. Exercise 16

■ 문 제

16번 한글판 교제 문제 내용
주어진 N*N 이차원 격자 내에서 (1,1)을 시작으로 (N,N)까지 이동 가능한 경로의 수를 계산하고자 한다. 단 위 방향, 우측 방향으로만 한 칸씩 이동 가능하며 사선으로는 이동할 수 없다. 다음 그림은 N이 5일 때 이동 가능한 경로 중 3개를 나타낸다.



- (r,c) 에서 한칸만 움직인다면 경우의 수는 오른쪽 이동과 위쪽 이동 두 가지만 존재한다.
- 따라서 , (r,c)부터 (n,n)까지 가는 경로의 수는 (r+1,c)부터 (n,n)까지 가는 (r, c+1)부터 (n,n)까지 가는 경우의 수의 합이 된다.
- 현재 위치의 r 또는 c가 n인 경우는 base case로서 경로의 수는 1

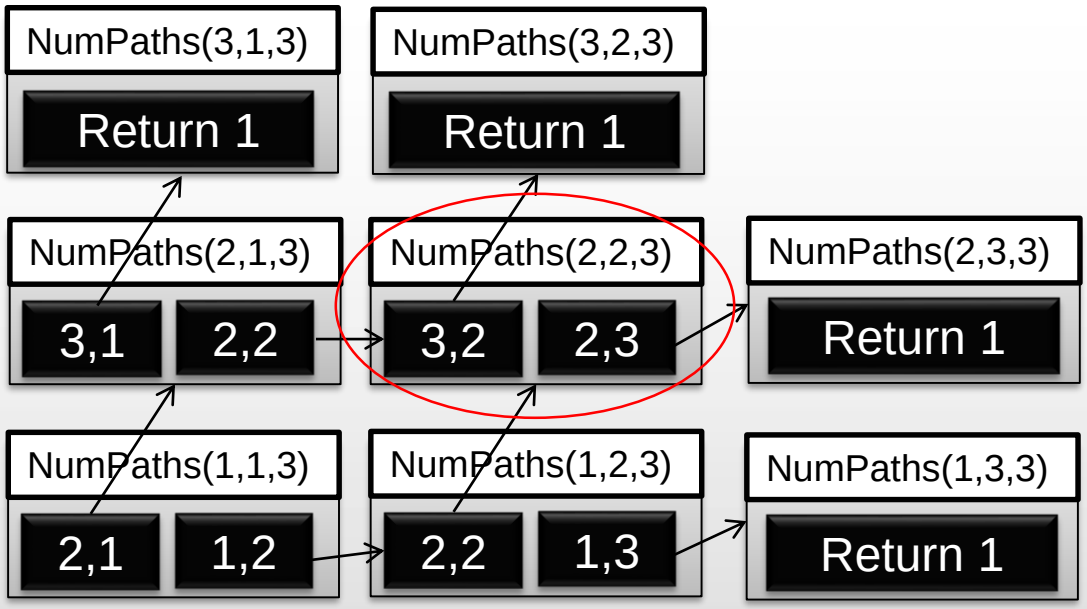
- A. (1,1)에서 (n,n)까지 이동 가능한 경로를 계산하는 NumPaths 함수를 구현하시오.
- B. A에서 구현한 방법은 같은 칸에서 n,n까지의 경로를 재계산 하는 경우가 발생한다. 2차원 행렬을 이용하여 재계산하지 않도록 알고리즘을 수정하시오.

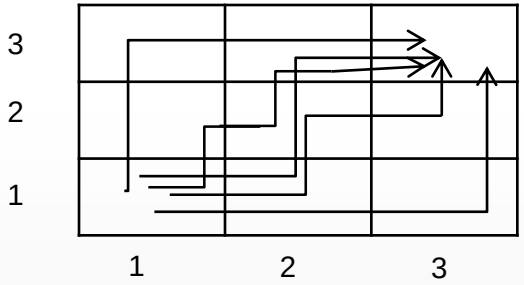
4-help slide

- **A. (1,1)에서 (n,n)까지 이동 가능한 경로를 계산하는 NumPaths 함수를 구현하시오.**

```
int NumPaths(int row,int col,int n)
{
        if((row==n)||(col==n))
                //경우의 수가 1가지 이므로 1을 리턴.
        else
                //row+ 1, col 과 row, col+ 1을 더한다.
}
```

4-help slide





NumPaths(2,2,3) : 2번

복잡한 계산을 반복하게 된다.

4-help slide

- **B. 2차원 배열을 이용하여 아직 계산하지 않은 칸은 -1로 설정하고 계산한 값은 현재 칸부터 목적지까지 총 이동 경로수를 기억시켜 놓는다. 따라서 재귀적으로 호출될 때-1일때만 수행하여 프로그램의 성능을 향상시킨다.**

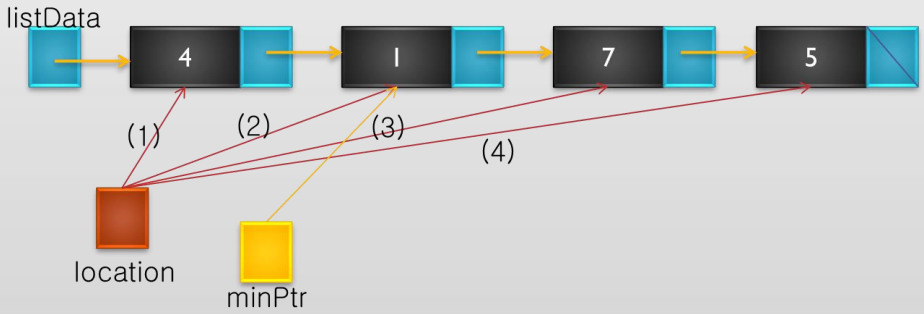
```
//프로그램 수행시 배열 mat[][]는 모두 -1 (아직 경로 개수의
값이 구해지지 않은 것을 나타냄)로 초기화 되어있다고 가정한다.
int mat[MAX_ROWS][MAX_COLS];
int NumPaths_C(int row,int col,int n)
{
          if(mat[row][col] == -1) { // 아직 구해지지 않은
경우
              // (a) 번의 코드를 이용하여 계산
              위에서 구한 값을 mat[row][col]에 기억시켜 놓음
         }
         return mat[row][col];
}
```

5. Exercise 21

문제

- 다음의 두 재귀 루틴이 갖는 인자는 중복되지 않고 정렬되지 않는 숫자로 구성된 단일 연결 리스트에 대한 포인터이다. 그리고 리스트의 각 노드는 info(숫자)와 next(다음 노드에 대한 포인터) 멤버로 구성된다.
  - 1번 문제와 같은 Unsorted Linked List 파일 사용 (리스트에 저장되는 데이터 타입은 Integer)
- **A. 가장 최소값을 갖는 노드를 리턴하는 멤버함수 MinLoc을 구현하시오. (단, 재귀버전으로 구현해야함)**
- **B. 원소들을 정렬하기 위한 멤버함수 Sort를 구현하시오. (단, MinLoc함수를 이용하여 구현할 것)**

예제



location으로 각 노드를 비교하면서 가장 작은 값을 가지고 있는 노드를 리턴(minPtr)합니다.(location->next == NULL 까지 동작)

5 help slide

구현의 예 : 첫번째 버전 (앞에서 뒤로 이동하면서 minPtr를 구함): 클래스 정의

```
template<class ItemType>
class UnsortedType
{
      public:
            ...
            //b번의 sort 함수 정의
            void Sort(NodeType<ItemType> *location);
      protected:
            //최소값을 찾는 함수 정의
            NodeType<ItemType>* MinLoc(NodeType<ItemType> *location,
      NodeType<ItemType>* minPtr);
            // minPtr은 리스트의 location 이전에 있는 minimum 노드를 가리키는 입력값임
}
```

MinLoc 함수 구현

```
//함수 호출 시 파라미터로 클래스 내부 변수인 listData가 넘겨져야 한다.
template <class ItemType>
NodeType<ItemType>* UnsortedType<ItemType>::MinLoc(NodeType<ItemType> *location,
NodeType<ItemType>* minPtr)
{
                  if(location != NULL) //general case
                  {
                              if(location이 가리키는 값 <  minPtr가 가리키는 값)
                                          // minPtr가 location이 되게 함
                              return MinLoc( ..., ...); // 다음 노드로 함수 재귀 호출
                  }
                  else  // base case
                              return minPtr;
}
```

5 help slide

구현의 예 : 두 번째 방법 (리스트 끝까지 간 후 리턴하면서 minPtr를 구함): 클래스 정의

```
template<class ItemType>
class UnsortedType
{
    public:
        ...
        //b번의 sort 함수 정의
        void Sort(NodeType<ItemType> *location);
    protected:
        //최소값을 찾는 함수 정의
        NodeType<ItemType>* MinLoc(NodeType<ItemType> *location);
        // 앞 페이지 버전과 비교하여 파라메터 minPtr가 필요하지 않음에 주목
}
```

MinLoc 함수 구현

```
//함수 호출 시 파라미터로 클래스 내부 변수인 listData가 넘겨져야 한다.
template <class ItemType>
NodeType<ItemType>* UnsortedType<ItemType>::MinLoc(NodeType<ItemType> *location)  {
    if (location == NULL) // base case: 원래 리스트가 empty (listData가 NULL) 일 때
        return NULL;
    else if (location->next == NULL) // another base case : 리스트의 마지막 노드
        return location;
    else { // general case: location != NULL
        NodeType<ItemType> *minPtr = MinLoc(location->next);
        if(location이 가리키는 값 <  minPtr가 가리키는 값) // minPtr은 절대 NULL이 아님. Why?
            // minPtr가 location이 되게 함
        return minPtr;
    }
}
```

5. help slide

Sort 함수 구현

```
//int 타입의 임시 저장 공간을 만든다.
//MinLoc함수를 통해 작은 값의 위치를 파악, 그 값을 임시 저장 공간에 저장한다.
//현재 가리키고 있는 노드의 info를 MinLoc의 info에 대입하고, 임시 저장한
//최소값으로 location의 info를 바꿔가면서 정렬을 구현한다.

//Sort함수의 파라메터는 listData를 넘겨준다.

template <class ItemType>
void UnsortedType<ItemType>::Sort(NodeType<ItemType> *location)
{
        NodeType<ItemType> *minPtr; //최소값을 가리키는 포인터
        ItemType temp;

        if(location != NULL) // empty 리스트가 아니면
        {
                minPtr = MinLoc(loctaion, location);  // ***
                // location에 저장된 값과 minPtr에 저장된 값을 exchange
                        // temp에 minPtr->info의 내용을 저장
                        //minPtr->info에 location->info를 저장
                        //location->info에 temp의 내용 저장
                //다음 노드로(location->next) 재귀함수 호출
        }
        // base case는 do nothing
}

// 위 ***에서 현재 location이 마지막 노드이거나 리스트의 minimum 값을 갖는다면 어떻게 될까? 좀 더
efficient하게 수정하려면?
```