Chapter 03.
연결리스트1

3주차

추상 자료형(abstract data type:ADT)의 이해

74-76p

◆ 추상 자료형이란?

- 구체적인 기능의 완성과정을 언급하지 않고, 순수하게 기능이 무엇인지를 나열한 것

◆ (예) 지갑의 추상 자료형

- 카드의 삽입
- 카드의 추출(카드를 빼냄)
- 동전의 삽입
- 동전의 추출(동전을 빼냄)
- 지폐의 삽입
- 지폐의 추출(지폐를 빼냄)

76-78p

Wallet의 ADT

Wallet ADT

```
void PutMoney(Wallet *pw, int coinNum, int billNum);
int TakeOutMoney(Wallet *pw, int coinNum, int billNum );
```

Wallet ADT 기반의 main

```
int main(void) {
    Wallet myWallet;        // 지갑 하나 장만 했음!
    . . . .
    PutMoney(&myWallet, 5, 10);       // 돈을 넣는다.
    . . . .
    ret = TakeOutMoney(&myWallet, 2, 5);    // 돈을 꺼낸다.
    . . . .
}
```

자료구조의 학습에 ADT의 정의를 포함

◆ 자료구조
- ADT정의가 우선
- 내부 구현을 알지 못하더라도 활용가능
- 하나의 ADT에 대해, 구현방법은 다양할 수 있음

◆ 자료구조 학습의 옳은 순서
- 리스트 자료구조의 **ADT를 정의**한다.
- ADT를 근거로 리스트 자료구조를 **활용하는 main 함수를 정의**한다.
- ADT를 근거로 **리스트를 구현**한다.

리스트의 이해

◆ 리스트의 ADT

- 저장 형태 : 데이터를 나란히 저장.
  - ❖ 앞 뒤 순서가 명확
  - ❖ 앞과 뒤에 각각 1개씩의 요소만 존재 가능
- 저장 특성
  - ❖ 중복데이터 저장 허용
- 구현 방법에 따른 구분
  - ❖ 순차리스트 (배열 기반)
  - ❖ 연결리스트 (메모리의 동작 할당 기반)

◆ 모든 자료구조의 핵심 연산 3가지

- 원소 삽입
- 원소 삭제
- 원소 조회

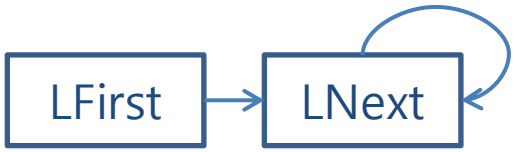
리스트의 ADT 정의

```
// 초기화
void ListInit(List * plist);

// 원소 삽입
void LInsert(List * plist, LData data);

// 원소 조회(탐색)
int LFirst(List * plist, LData * pdata); // 저장된 데이터의 탐색위치 초기화
int LNext(List * plist, LData * pdata); // 다음 데이터의 참조(반환)
int LCount(List * plist);      // 데이터의 수를 반환

// 원소 삭제
LData LRemove(List * plist);  // 바로 이전에 참조(반환) 데이터의 삭제
```



리스트의 배열기반구현
(List 정의)

```
#include <stdio.h>

#define TRUE      1
#define FALSE     0

/*** List의 정의 ****/
#define LIST_LEN 100
typedef int LData;

typedef struct __ArrayList
{
        LData arr[LIST_LEN];
        int numOfData;
        int curPosition;
} List;
```

리스트의 배열기반구현
(List 사용하는main 함수 1/2)

```
int main(void) {
    List list;
    int data;
    ListInit(&list);

    LInsert(&list, 11);  LInsert(&list, 11);
    LInsert(&list, 22);  LInsert(&list, 22);
    LInsert(&list, 33);

    printf("현재 데이터의 수: %d ₩n", LCount(&list));

    if ( LFirst(&list, &data) ) {
        printf("%d ", data);
        while( LNext(&list, &data) )
            printf("%d ", data);
    }
    printf("₩n₩n");
```

ArrayList의 생성 및 초기화

5개의 데이터 저장

저장된 데이터의 전체 출력

첫 번째 데이터 조회

두 번째 이후의 데이터 조회

리스트의 배열기반구현
(List 사용하는main 함수 2/2)

```
    if (LFirst(&list, &data)) {
        if (data == 22)
            LRemove(&list);

        while (LNext(&list, &data)) {
            if (data == 22)
                LRemove(&list);
        }
    }

    printf("현재 데이터의 수: %d ₩n", LCount(&list));

    if (LFirst(&list, &data)) {
        printf("%d ", data);
        while (LNext(&list, &data))
            printf("%d ", data);
    }
    printf("₩n₩n");
    return 0;
}
```

숫자 22을 탐색하여 모두 삭제

삭제 후 저장된 데이터 전체 출력

리스트의 배열기반구현
(List와 관련된 연산들)

```
void ListInit(List * plist)
{
	(plist->numOfData) = 0;
	(plist->curPosition) = -1;
}

void LInsert(List * plist, LData data)
{
	if (plist->numOfData > LIST_LEN)
	{
		puts("저장이 불가능합니다.");
		return;
	}

	plist->arr[plist->numOfData] = data;
	(plist->numOfData)++;
}
```

리스트의 배열기반구현
(List와 관련된 연산들)

```
int LFirst(List * plist, LData * pdata)
{
	if (plist->numOfData == 0)
		return FALSE;

	(plist->curPosition) = 0;
	*pdata = plist->arr[0];
	return TRUE;
}

int LNext(List * plist, LData * pdata)
{
	if (plist->curPosition >= (plist->numOfData) - 1)
		return FALSE;

	(plist->curPosition)++;
	*pdata = plist->arr[plist->curPosition];
	return TRUE;
}
```

89-91p

리스트의 배열기반구현
(List와 관련된 연산들)

```
LData LRemove(List * plist)
{
	int rpos = plist->curPosition;
	int num = plist->numOfData;
	int i;
	LData rdata = plist->arr[rpos];

	for (i = rpos; i < num - 1; i++)
		plist->arr[i] = plist->arr[i + 1];

	(plist->numOfData)--;
	(plist->curPosition)--;
	return rdata;
}

int LCount(List * plist)
{
	return plist->numOfData;
}
```

95-99 구조체 배열로 구현
-> skip

100p

배열 기반 리스트의 장점과 단점

◆ 배열 기반 리스트의 단점

- 배열의 길이가 초기에 결정되어야 한다. 변경이 불가능하다.
- 삭제의 과정에서 데이터의 이동(복사)가 매우 빈번히 일어난다.

◆ 배열 기반 리스트의 장점

- 데이터 참조가 쉽다. 인덱스 값 기준으로 어디든 한 번에 참조 가능!