연결 리스트(Linked List) 1

Chapter 03. 연결 리스트(Linked List) 1

Chapter 03-1:

추상 자료형: Abstract Data Type

추상 자료형(ADT)의 이해

추상 자료형이란?

구체적인 기능의 완성과정을 언급하지 않고, 순수하게 기능이 무엇인지를 나열한 것

지갑의 추상 자료형

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

기능의 명세를 가리켜 왜 자료형이라 하는 것일까?

지갑을 의미하는 구조체 Wallet의 정의

자료형 Wallet의 정의

```
typedef struct _wallet
{
    int coin100Num;      // 100원짜리 동전의 수
    int bill5000Num;     // 5,000원짜리 지폐의 수
} Wallet;
```

자료형 int와 관련해서 우리가 아는것을 나열해 보자! 어떻게 생겼는지 아는 것이 아니라 어떠한 연산이 가능한지를 알고 있지 않은가?

완전한 자료형의 정의로 인식되기 위해서는
해당 자료형과 관련이 있는 연산이 함께 정의되어야 한다!

자료형 Wallet 정의의 일부

```
int TakeOutMoney(Wallet * pw, int coinNum, int billNum);    // 돈 꺼내는 연산
void PutMoney(Wallet * pw, int coinNum, int billNum);       // 돈 넣는 연산
```

구조체 Wallet의 추상 자료형 정의

Wallet 기반의 main

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

Wallet의 ADT

Operations:

- `int TakeOutMoney(Wallet * pw, int coinNum, int billNum)`
  - 첫 번째 인자로 전달된 주소의 지갑에서 돈을 꺼낸다.
  - 두 번째 인자로 꺼낼 동전의 수, 세 번째 인자로 꺼낼 지폐의 수를 전달한다.
  - 꺼내고자 하는 돈의 총액이 반환된다. 그리고 그만큼 돈은 차감된다.

- `void PutMoney(Wallet * pw, int coinNum, int billNum)`
  - 첫 번째 인자로 전달된 주소의 지갑에 돈을 넣는다.
  - 두 번째 인자로 넣을 동전의 수, 세 번째 인자로 넣을 지폐의 수를 전달한다.
  - 넣은 만큼 동전과 지폐의 수가 증가한다.

구조체 Wallet의 정의를 ADT에 포함시키지 않아도 됨을 보인다.

자료구조의 학습에 ADT의 정의를 포함합니다.

자료구조 학습의 옳은 순서

1. 리스트 자료구조의 ADT를 정의한다.
2. ADT를 근거로 리스트 자료구조를 활용하는 main 함수를 정의한다.
3. ADT를 근거로 리스트를 구현한다.

자료구조의 내부 구현을 모르고도 해당 자료구조의 활용이 가능하도록 ADT를 정의하는 것이 옳다.

main 함수를 먼저 접하게 되면, 구현할 자료구조를 구성하는 함수들을 잘 이해할 수 있다.

Chapter 03. 연결 리스트(Linked List) 1

Chapter 03-2:

배열을 이용한 리스트의 구현

리스트의 이해

리스트의 구분

- 순차 리스트 : 배열을 기반으로 구현된 리스트
- 연결 리스트 : 메모리의 동적 할당을 기반으로 구현된 리스트

이는 구현 방법을 기준으로 한 구분이다. 따라서 이 두 리스트의 ADT가 동일하다고 해서 문제가 되지는 않는다.

리스트의 특징

- 저장 형태 : 데이터를 나란히(하나의 열로) 저장한다.
- 저장 특성 : 중복이 되는 데이터의 저장을 허용한다.

이것이 리스트의 ADT를 정의하는데 있어서 고려해야 할 유일한 내용이다.

리스트 자료구조의 ADT 1

- `void ListInit(List * plist);`
  - 초기화할 리스트의 주소 값을 인자로 전달한다.
  - 리스트 생성 후 제일 먼저 호출되어야 하는 함수이다.

리스트의 초기화

- `void LInsert(List * plist, LData data);`
  - 리스트에 데이터를 저장한다. 매개변수 data에 전달된 값을 저장한다.

데이터 저장

- `int LFirst(List * plist, LData * pdata);`
  - 첫 번째 데이터가 pdata가 가리키는 메모리에 저장된다.
  - 데이터의 참조를 위한 초기화가 진행된다.
  - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

저장된 데이터의 탐색 및 탐색 초기화

LData는 저장 대상의 자료형을 결정할 수 있도록 typedef로 선언된 자료형의 이름이다.

리스트 자료구조의 ADT 2

- `int LNext(List * plist, LData * pdata);`
  - 참조된 데이터의 다음 데이터가 pdata가 가리키는 메모리에 저장된다.
  - 순차적인 참조를 위해서 반복 호출이 가능하다.
  - 참조를 새로 시작하려면 먼저 LFirst 함수를 호출해야 한다.
  - 참조 성공 시 TRUE(1), 실패 시 FALSE(0) 반환

다음 데이터의 참조(반환)을 목적으로 호출

- `LData LRemove(List * plist);`
  - LFirst 또는 LNext 함수의 마지막 반환 데이터를 삭제한다.
  - 삭제된 데이터는 반환된다.
  - 마지막 반환 데이터를 삭제하므로 연이은 반복 호출을 허용하지 않는다.

바로 이전에 참조(반환)이 이뤄진 데이터의 삭제

- `int LCount(List * plist);`
  - 리스트에 저장되어 있는 데이터의 수를 반환한다.

현재 저장되어 있는 데이터의 수를 반환

ArrayList.h

```
#ifndef __ARRAY_LIST_H__
#define __ARRAY_LIST_H__

#define TRUE		1
#define FALSE		0

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

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


/*** ArrayList와 관련된 연산들 ****/
typedef ArrayList List;

void ListInit(List * plist);
void LInsert(List * plist, LData data);

int LFirst(List * plist, LData * pdata);
int LNext(List * plist, LData * pdata);

LData LRemove(List * plist);
int LCount(List * plist);

#endif
```

ArrayList.c

```
#include <stdio.h>
#include "ArrayList.h"

void [redacted](List * plist)
{
	(plist->numOfData) = 0;
	(plist->curPosition) = -1;
}

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

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

int [redacted](List * plist, LData * pdata)
{
	if(plist->numOfData == 0)
		return FALSE;

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

int LNext(List * plist, LData * pdata)
{
	if(plist->curPosition >= ([redacted]))
		return FALSE;

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

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->[redacted] = plist->[redacted];

	(plist->[redacted])--;
	(plist->[redacted])--;
	return rdata;
}

int LCount(List * plist)
{
	return plist->[redacted];
}
```

ListMain.c

```
#include <stdio.h>
#include "ArrayList.h"

int main(void)
{
	/*** ArrayList의 생성 및 초기화 ***/
	List list;
	int data;
	ListInit(&list);

	/*** 5개의 데이터 저장 ***/
	LInsert(&list, 11);  LInsert(&list, 11);
	LInsert(&list, 22);  LInsert(&list, 22);
	LInsert(&list, 33);

	/*** 저장된 데이터의 전체 출력 ***/
	printf("현재 데이터의 수: %d \n", LCount(&list));

	// 첫 번째 데이터 조회
	if(LFirst(&list, &data))
	{
		printf("%d ", data);
		// 두 번째 이후의 데이터 조회
		while(LNext(&list, &data))
			printf("%d ", data);
	}
	printf("\n\n");

	/*** 숫자 22을 탐색하여 모두 삭제 ***/
	if(LFirst(&list, &data))
	{
		if(data == 22)
			LRemove(&list);

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

	/*** 삭제 후 저장된 데이터 전체 출력 ***/
	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;
}
```

리스트의 초기화와 데이터 저장 과정

리스트의 초기화

```
int main(void)
{
    List list;              // 리스트의 생성
    . . . .
    ListInit(&list);        // 리스트의 초기화
    . . . .
}
```

```
int main(void)
{
    . . . .
    LInsert([redacted], 11);    // 리스트에 11을 저장
    LInsert([redacted], 22);    // 리스트에 22를 저장
    LInsert([redacted], 33);    // 리스트에 33을 저장
    . . . .
}
```

초기화된 리스트에 데이터 저장

리스트의 데이터 참조 과정

```
int main(void)
{
    . . . .
    if( LFirst(&list, &data) )      // 첫 번째 데이터 변수 data에 저장
    {
        printf("%d ", data);
        while( [redacted] )    // 두 번째 이후 데이터 변수 data에 저장
            printf("%d ", data);
    }
    . . . .
}
```

데이터 참조의 새로운 시작을 위해서

LFirst 함수의 호출을 요구한다!

√ 데이터 참조 일련의 과정

LFirst → LNext → LNext → LNext → LNext . . . .

리스트의 데이터 삭제 방법

```
int main(void)
{
    . . . .
    if( LFirst(&list, &data) )
    {
        if(data == 22)
            [redacted];    // 위의 LFirst 함수를 통해 참조한 데이터 삭제!

        while( [redacted] )
        {
            if(data == 22)
                LRemove(&list); // 위의 LNext 함수를 통해 참조한 데이터 삭제!
        }
    }
    . . . .
}
```

LRemove 함수는 연이은 호출을 허용하지 않는다!

프로그램의 논리상 LRemove 함수의 연이은 호출이 불필요함을 이해하는 것도 중요하다!

배열 기반 리스트의 헤더파일 정의 1

```
#ifndef __ARRAY_LIST_H__
#define __ARRAY_LIST_H__

#define TRUE        1       // '참'을 표현하기 위한 매크로 정의
#define FALSE       0       // '거짓'을 표현하기 위한 매크로 정의

#define LIST_LEN   100
typedef int LData;

typedef struct __ArrayList     // 배열기반 리스트를 정의한 구조체
{
    LData arr[LIST_LEN];       // 리스트의 저장소인 배열
    int numOfData;             // 저장된 데이터의 수
    int curPosition;           // 데이터 참조위치를 기록
} ArrayList;

typedef ArrayList List;
```

저장할 대상의 자료형을 변경을 위한 typedef 선언

배열 기반 리스트를 의미하는 구조체

리스트의 변경을 용이하게 하기 위한 typedef 선언

위의 내용 중 일부는 리스트를 배열 기반으로 구현하기 위한 선언을 담고 있다.

배열 기반 리스트의 헤더파일 정의 2

```
void ListInit(List * plist);                // 초기화
void LInsert(List * plist, LData data);     // 데이터 저장

int [redacted](List * plist, LData * pdata);    // 첫 데이터 참조
int [redacted](List * plist, LData * pdata);    // 두 번째 이후 데이터 참조

LData LRemove(List * plist);                // 참조한 데이터 삭제
int [redacted](List * plist);               // 저장된 데이터의 수 반환

#endif
```

위의 함수들은 리스트 ADT를 기반으로 선언된 함수들이다.

따라서 배열 기반 리스트로 선언된 함수들의 내용을 제한할 필요가 없다.

배열 기반 리스트의 초기화

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

리스트에 저장된 데이터의 수

마지막 참조 위치에 대한 정보 저장

배열 기반 리스트의 초기화

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

-1은 아무런 위치도 참조하지 않았음을 의미함!

실제로 초기화할 대상은 구조체 변수의 멤버이다. 따라서 초기화 함수의 구성은 구조체의 정의를 기반으로 한다.

배열 기반 리스트의 삽입

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

배열에 데이터 저장!

```
void LInsert(List * plist, LData data)
{
    if(plist->numOfData >          )     // 더 이상 저장할 공간이 없다면
    {
        puts("저장이 불가능합니다.");
        return;
    }

    plist->arr[                    ] = data;    // 데이터 저장
    (plist->          )++;          // 저장된 데이터의 수 증가
}
```

배열 기반 리스트의 조회

```
int [redacted](List * plist, LData * pdata)
{
    if(plist->numOfData == 0)      // 저장된 데이터가 하나도 없다면
        return FALSE;

    (plist->curPosition) = 0;      // 참조 위치 초기화! 첫 번째 데이터의 참조를 의미!
    *pdata = plist->[redacted];    // pdata가 가리키는 공간에 데이터 저장
    return TRUE;
}
```

초기화! 및 첫 번째 데이터 참조

만약에 중간 지점에서 다시 처음부터 참조하기 원한다면?

```
int LNext(List * plist, LData * pdata)
{
    if(plist->curPosition >= ([redacted])      // 더 이상 참조할 데이터가 없다면
        return FALSE;

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

그 다음 데이터 참조

값의 반환은 매개변수를 통해서!

함수의 반환은 성공여부를 알리기 위해서!

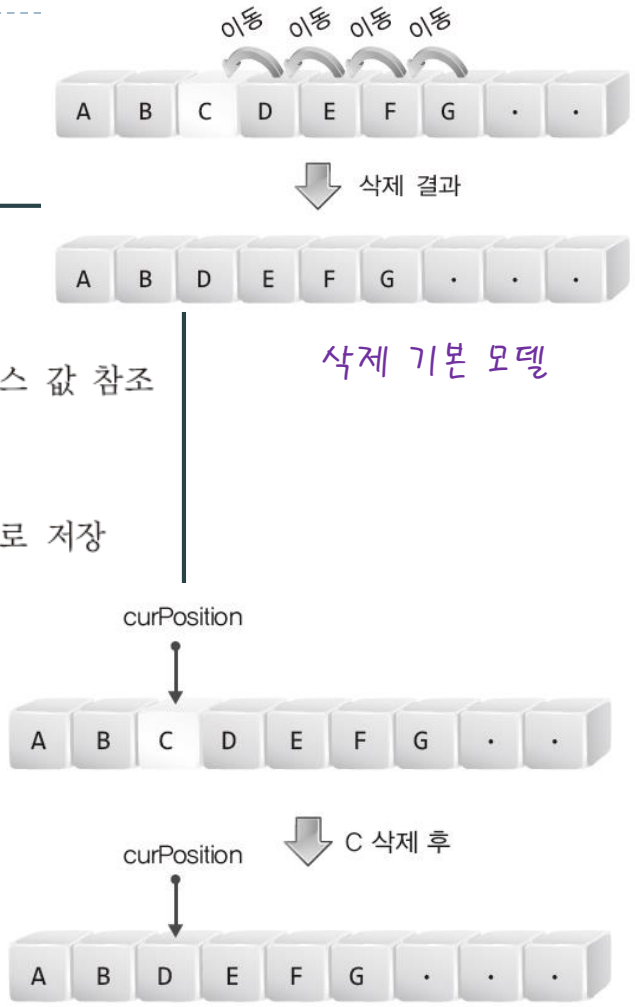
배열 기반 리스트의 삭제

삭제되는 데이터는 반환의 과정을 통해서 되돌려주는 것이 옳다!

```
LData LRemove(List * plist)
{
    int rpos = plist->curPosition;      // 삭제할 데이터의 인덱스 값 참조
    int num = plist->numOfData;
    int i;
    LData rdata = plist->arr[rpos];     // 삭제할 데이터를 임시로 저장

    // 삭제를 위한 데이터의 이동을 진행하는 반복문
    for(i=rpos; i<num-1; i++)
        plist->[redacted] = plist->[redacted];

    (plist->[redacted])--;              // 데이터의 수 감소
    (plist->[redacted])--;              // 참조위치를 하나 되돌린다.
    return rdata;                       // 삭제된 데이터의 반환
}
```



참조 위치를 하나 되돌려야 하는 이유