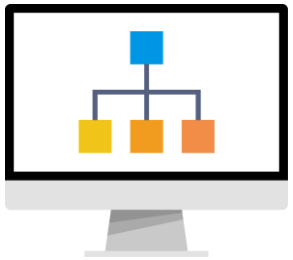


Data Structure & Algorithm
자료구조 및 알고리즘

5. 연결 리스트 (Chapter3, Linked List)

추상화 (abstraction)

- 컴퓨터 공학에서 복잡도를 관리하는 방법

- 더욱 중요한 것에 집중하기 위해 물리적, 공간적, 시간적 세부 사항을 없애는 행위

- 어떤 복잡한 기능을 모두 구현하고 이해하는 것은 힘들다.
- 복잡한 기능을 계층으로 나누고 각 계층에서는 1) 하위 계층에서 구현한 것을 활용하여 2) 현재 계층에서 어떻게 해야 하는지만 생각하자.

사진 찍어서 클라우드에 올리기

1) 전기적으로 정보를 저장하는 장치를 만든다.
2) 사진의 색상정보를 0 또는 1로 나타내는 방법을 고안한다.
3) 2에서 표현한 정보를 1을 이용해 저장하고 관리하는 소프트웨어를 만든다.
4) 컴퓨터 간 정보를 전달할 수 있는 방법을 개발한다.
5) 다른 컴퓨터에 접속해 컴퓨터에서 제공하는 정보를 그래픽으로 보여줄 수 있는 소프트웨어를 만든다.
6) 5의 소프트웨어를 활용하여 다른 컴퓨터에 사진의 정보를 전송한다.

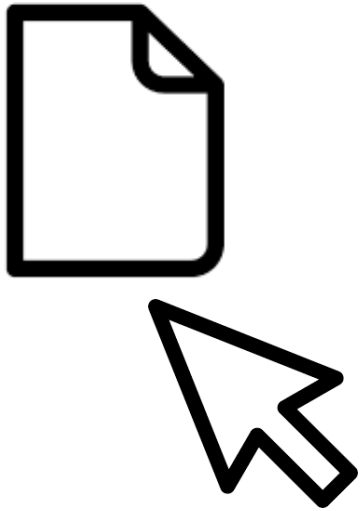
사진 찍어서 클라우드에 올리기

1) 전기적으로 정보를 저장하는 장치를 만든다 **(SSD)**.
2) 사진의 색상정보를 0 또는 1로 나타내는 방법을 고안한다 **(이미지 인코딩 포맷, JPEG)**.
3) 2에서 표현한 정보를 1을 이용해 저장하고 관리하는 소프트웨어를 만든다 **(운영체제, Windows)**.
4) 컴퓨터 간 정보를 전달할 수 있는 방법을 개발한다 **(인터넷, TCP/IP)**.
5) 다른 컴퓨터에 접속해 컴퓨터에서 제공하는 정보를 그래픽으로 보여줄 수 있는 소프트웨어를 만든다 **(웹 브라우저, Chrome)**.
6) 5의 소프트웨어를 활용하여 다른 컴퓨터에 사진의 정보를 전송한다.

Drag&Drop files here

or

Browse Files



추상화의 장점

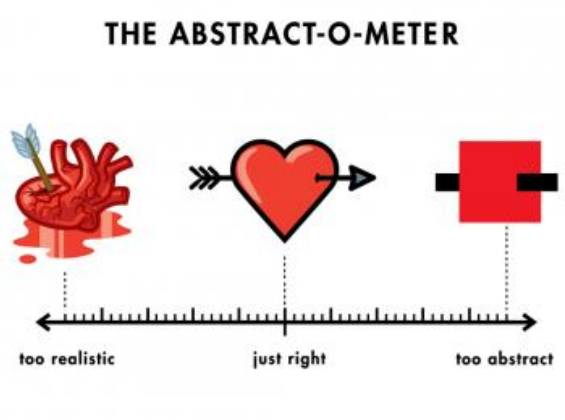
- 사용자는 아래에서 어떤 일이 벌어지는지 알 필요가 없다.
  - 어떤 작업이 가능하고 (파일 전송), 이를 어떻게 하면 되는지 (드래그 앤 드랍)만 배우면 충분하다.

- 입출금기계
  - 어떤 작업이 가능합니까? 현금을 넣거나 뺄 수 있습니다.
  - 어떻게 하면 됩니까? 현금 카드를 넣고, 비밀번호를 입력하고, 원하는 금액을 입력하면 됩니다.

  - 어떻게 구현된 거지요? 몰라도 됩니다.

추상화의 예

- 휴대전화
  - 어떤 작업이 가능합니까? 멀리 있는 사람과 대화를 할 수 있습니다.
  - 어떻게 하면 됩니까? 전화번호를 누르고 통화 버튼을 누르면 됩니다.



- 모든 사용자 매뉴얼은 어떤 작업이 가능하고 어떻게 하면 되는지를 설명하지, 구현을 설명하지 않는다.

자료 구조에서 추상화

- 입출금기계
  - 어떤 작업이 가능합니까? 현금을 넣거나 뺄 수 있습니다.
  - 어떻게 하면 됩니까? 현금 카드를 넣고, 비밀번호를 입력하고, 원하는 금액을 입력하면 됩니다.
  - 어떻게 구현된 거지요? 몰라도 됩니다.

- 배열
  - 어떤 작업이 가능합니까? i 번째 데이터 읽기 및 쓰기
  - 어떻게 하면 됩니까? a[i]로 i 번째 데이터에 접근
  - 어떻게 구현된 거지요? 몰라도 됩니다.
    - **수강생분들은 아셔야 합니다.**

추상 자료형(ADT)의 이해

추상 자료형이란?

구체적인 기능의 완성과정을 언급하지 않고, 순수하게 기능이 무엇인지를 나열한 것

지갑

지갑의 추상 자료형

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

```
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

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.

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

지갑을 의미하는 구조체 Wallet의 정의

자료형 Wallet의 정의

구현은 감춘다.

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

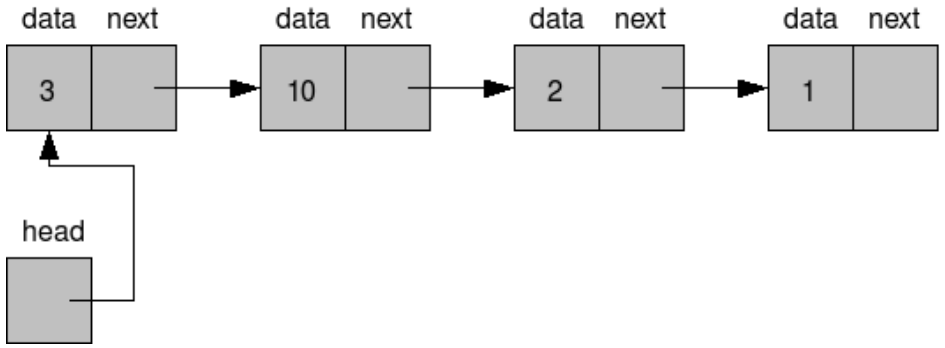
기능은 보인다.

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

Application Programming Interface (API)

배열과 리스트

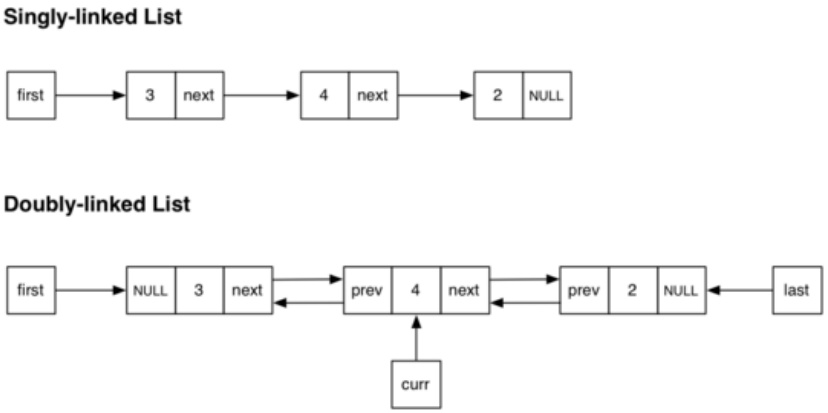
	배열 (Array)	리스트 (List)
길이	고정	가변
i 번째 값 읽기	O(1) Random Access	O(N) Sequential Access
삽입/삭제	O(N)	O(1)



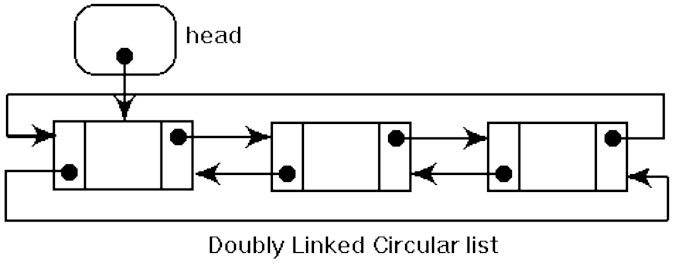
리스트의 종류

- 구현 방법에 따라서
  - 배열 리스트(ArrayList): 길이가 정해진 배열
  - 연결 리스트(LinkedList): 동적 할당을 활용

- 형태에 따라서
  - Singly Linked List
  - Doubly Linked List



- 원형 여부
  - Doubly Linked Circular List



리스트의 기능

- `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.

[ArrayList.h 참고] typedef int LData;

리스트의 기능

- `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

리스트를 사용하는 main 함수

예제 ListMain.c를 봅니다.

- ArrayList.h 리스트 자료구조의 헤더파일
- ArrayList.c 리스트 자료구조의 소스파일
- ListMain.c 리스트 관련 main 함수가 담긴 소스파일

실행을 위해 필요한 파일들

실행결과

```
현재 데이터의 수: 5
11 11 22 22 33

현재 데이터의 수: 3
11 11 33
```

리스트의 초기화와 데이터 저장

리스트의 초기화

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

```
int main(void)
{
    . . . .
    LInsert(&list, 11);     // 리스트에 11을 저장
    LInsert(&list, 22);     // 리스트에 22를 저장
    LInsert(&list, 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( LNext(&list, &data) )    // 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)
            LRemove(&list);     // 위의 LFirst 함수를 통해 참조한 데이터 삭제!

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

LRemove 함수는 연이은 호출을 허용하지 않는다!

프로그램의 논리상 LRemove 함수의 연이은 호출이 불필요함!

배열 기반 리스트의 헤더파일 정의

```
#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

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

int LFirst(List * plist, LData * pdata);    // 첫 데이터 참조
int LNext(List * plist, LData * pdata);     // 두 번째 이후 데이터 참조

LData LRemove(List * plist);                // 참조한 데이터 삭제
int LCount(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[LIST_LEN];
    int numOfData;
    int curPosition;
} ArrayList;
```

리스트에 저장된 데이터의 수

마지막 참조 위치에 대한 정보 저장

배열 기반 리스트의 초기화

```
void ListInit(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[LIST_LEN];
    int numOfData;
    int curPosition;
} ArrayList;
```

배열에 데이터 저장!

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

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

배열 기반 리스트의 조회

```
int LFirst(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->arr[0];        // pdata가 가리키는 공간에 데이터 저장
    return TRUE;
}
```

초기화! 및 첫 번째 데이터 참조

만약에 중간 지점에서 다시 처음부터 참조하기 원한다면?

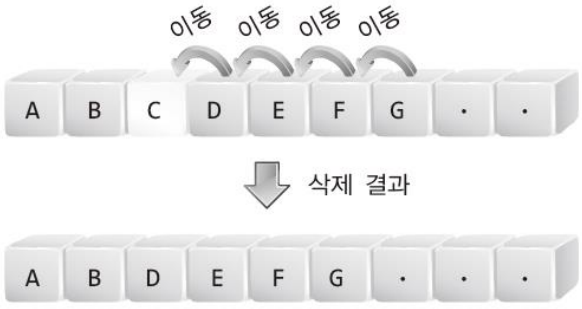
```
int LNext(List * plist, LData * pdata)
{
    if(plist->curPosition >= (plist->numOfData)-1)      // 더 이상 참조할 데이터가 없다면
        return FALSE;

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

그 다음 데이터 참조

값의 반환은 매개변수를 통해서!

함수의 반환은 성공여부를 알리기 위해서!

배열 기반 리스트의 삭제



삭제 기본 모델

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->arr[i] = plist->arr[i+1];

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

리스트에 구조체 변수 저장하기

리스트에 저장할 Point 구조체 변수의 헤더파일 선언 (Point.h)

```
typedef struct _point
{
    int xpos;
    int ypos;
} Point;

// Point 변수의 xpos, ypos 값 설정
void SetPointPos(Point * ppos, int xpos, int ypos);

// Point 변수의 xpos, ypos 정보 출력
void ShowPointPos(Point * ppos);

// 두 Point 변수의 비교
int PointComp(Point * pos1, Point * pos2);
```

리스트에 구조체 변수 저장하기

구조체 Point 관련 소스파일

```
void SetPointPos(Point * ppos, int xpos, int ypos) {
    ppos->xpos = xpos;
    ppos->ypos = ypos;
}

void ShowPointPos(Point * ppos) {
    printf("[%d, %d] ₩n", ppos->xpos, ppos->ypos);
}

int PointComp(Point * pos1, Point * pos2) {
    if(pos1->xpos == pos2->xpos && pos1->ypos == pos2->ypos)
        return 0;
    else if(pos1->xpos == pos2->xpos)
        return 1;
    else if(pos1->ypos == pos2->ypos)
        return 2;
    else
        return -1;
}
```

리스트에 구조체 변수 저장하기

ArrayList.h의 변경 사항 두 가지

```
typedef int LData;        (typedef 선언변경)▶      typedef Point * LData;

#include "Point.h"        // ArrayList.h에 추가할 헤더파일 선언문
```

typedef 선언에서 보이듯이 실제로 저장을 하는 것은 Point 구조체 변수의 주소 값이다!

실행을 위한 파일 구성

Point.h, Point.c	구조체 Point를 위한 파일
ArrayList.h, ArrayList.c	배열 기반 리스트
PointListMain.c	구조체 Point의 변수 저장

리스트에 구조체 변수 저장하기

예제 PointListMain.c를 봅니다.

실행결과

```
현재 데이터의 수: 4
[2, 1]
[2, 2]
[3, 1]
[3, 2]

현재 데이터의 수: 2
[3, 1]
[3, 2]
```

PointListMain.c의 일부: 초기화 및 저장

```
int main(void)
{
	List list;
	Point compPos;
	Point * ppos;

	ListInit(&list);

	/*** 4개의 데이터 저장 ***/
	ppos = (Point*)malloc(sizeof(Point));
	SetPointPos(ppos, 2, 1);
	LInsert(&list, ppos);

	ppos = (Point*)malloc(sizeof(Point));
	SetPointPos(ppos, 2, 2);
	LInsert(&list, ppos);
	. . . . . .
}
```

PointListMain.c의 일부: 참조 및 조회

```
int main(void)
{
	. . . . . .
	/*** 저장된 데이터의 출력 ***/
	printf("현재 데이터의 수: %d ₩n", LCount(&list));

	if(LFirst(&list, &ppos))
	{
		ShowPointPos(ppos);

		while(LNext(&list, &ppos))
			ShowPointPos(ppos);
	}
	printf("₩n");
	. . . . . .
}
```

PointListMain.c의 일부: 삭제

```
int main(void)
{
    . . . . . .
    /*** xpos가 2인 모든 데이터 삭제 ***/
    compPos.xpos=2;
    compPos.ypos=0;

    if(LFirst(&list, &ppos)) {
        if(PointComp(ppos, &compPos)==1) {
            ppos=LRemove(&list);
            free(ppos);
        }
        while(LNext(&list, &ppos)) {
            if(PointComp(ppos, &compPos)==1) {
                ppos=LRemove(&list);
                free(ppos);
            }
    }
    . . . . . .
}
```

요약

- 컴퓨터공학에서는 **추상화**를 통해 복잡도를 분산하여 관리한다.

- 추상 자료형(Abstract Data Type)에서는 데이터에 어떤 작업이 가능한지를 기술하고 구현은 감춘다.
  - 사용자는 구현을 모르고 어떻게 사용하는 지만 안다.
  - Application Programming Interface (API)

- 리스트는 데이터를 순차적으로 저장/관리하는 ADT이다.
  - 배열과 비교하여 가변 길이이고 삽입/삭제가 빠르다는 장점이 있다.