자료구조의 정의

- 컴퓨터 프로그램

> **데이터 + 명령어**

- 자료구조

> **데이터를 효율적으로 처리할 수 있도록 만들어진 데이터의 처리 방식**

- 알고리즘

> **데이터를 효율적으로 처리할 수 있는 방법**

→ 프로그램의 목적에 따라 올바른 자료구조를 사용할 경우 얻을 수 있는 이득

1) 메모리의 **효율적 사용**

2) 프로그램 **수행 시간의 단축**

◆ 효율적인 알고리즘 적용하기 위해서는 알맞은 자료구조가 선행 되어야 함

- 자료구조의 분류

1. 단순구조 : 기본적인 데이터타입들이 해당

   → EX) int, float, double, ...

2. 선형구조 : 각각의 자료들이 1:1로 연결된 형태

   → EX) 스택, 큐, 리스트, 덱, ...

3. 비선형구조 : 각각의 자료들이 1:다 또는 다:다로 연결된 형태

   → EX) 트리, 그래프, ...

4. 파일구조 : 일반적으로 메모리에 로드하기 어려운 큰 자료들이 대상

   → 보조기억장치에 저장하는 것을 전제로 만들어졌다.

   → 구성 방식에 따라 순차적, 상대적, 색인, 다중 키 파일구조등이 있다.

- 추상자료형

> **자료형이란?**

- 자료(데이터) + 명령어
- EX) int 자료형 = int + 사칙연산을 필두로 한 명령어들

> **정보은닉**

- 사용자로 하여금 핵심적인 정보만 확인할 수 있게 하고, 내부적 기능이나, 형태는 공개하지 않는 것 → 백조를 생각하면 편할 듯 하다.

> **추상자료형이란?**

- 정보은닉 + 자료형
- 데이터와 명령어로 구성된 자료형에 정보은닉의 개념을 쓰까 만든 것
- 사용에 있어서 핵심적인 데이터나 명령어들의 정의만 공개하고 내부 로직 같은 필수적이지 않은 부분들은 비공개로 정의한 형태

- 알고리즘

> **알고리즘이란?**

- 문제해결을 위한 일련의 절차
- 컴퓨터가 이해할 수 있는 명백한 명령어의 집합이면서 어떤 문제를 해결하기 위한 절차
- 알고리즘은 문제 해결을 위한 5가지 조건이 필요

1) 입력 : 외부에서 제공되는 자료가 0개이상 존재해야 한다.
2) 출력 : 적어도 1개 이상의 결과를 만들어야 한다.
3) 유한성 : 각 명령어는 의미가 모호해서는 안된다.
4) 명백성 : 한정된 수의 단계 뒤에는 반드시 종료되어야 한다.
5) 유효성 : 모든 명령은 실행 가능한 연산이어야 한다.

> **알고리즘의 표현법**

- 대표적으로 4가지의 표현 방법이 존재

  1) 자연어

  2) 순서도

  3) 의사코드

  4) 프로그래밍 언어

> **알고리즘의 분석기준**

- 공간 복잡도
- 시간 복잡도

  → 임베디드 환경을 제외한 대부분의 환경에서 시간 복잡도를 우선시 함

> **시간 복잡도**

- 계산방법
  - 한번의 연산을 1로 잡아 명령어들의 구성을 통해 함수를 도출
  - 반복문의 경우 명령어 동작시 적용되는 연산 마다 1씩 증가 후 * 반복횟수
- 빅오 표기법
  - 시간 복잡도 함수에서 가장 큰 차수의 항만 도출해 표기
  - O(n)의 형태
  - n < logn < nlogn < n^2 < n^3 < 2^n < n!

C프로그래밍

- 컴파일 및 실행 프로그램 작성

→ 빌드 시 출력되는 메시지에서 구성:DebugWin32가 출력 되는 것을 확인 가능

→ 일반적으로 visual Studio의 프로젝트는 2개의 빌드환경을 가짐

1) Debug : C 소스파일을 컴파일 할 때 디버깅 정보를 실행 프로그램에 자동 추가

→ 솔루션 파일이 저장되는 폴더의 하위Debug폴더에 저장

2) Release : 개발이 완료된다음에 실행 프로그램 배포 시에 Release환경에서 빌드 된 실행 파일을 배포

→ Release폴더에 저장

- 단순 자료형

> **정수**

- 소수점이하는 저장하지 않음
- unsigned추가시 음수 값에 할당된 비트까지 가져와서 표현 가능

> **실수**

- 소수를 포함하는 수
- 수의 범위에서 E라는 문자가 표현되는대, E다음에 오는 수가 지수(Exponential)이라는 의미이다.

> **문자**

- 문자를 나타내는대 사용

- 배열

> **같은 자료 형의 데이터를 메모리상에 연속적으로 저장하는 자료형**

> **연속된 각각의 값을 배열의 원소(Element)라고 한다.**

→ 어떤 자료형의 배열수를 추출하는 방법 : Sizeof(배열)/sizeof(배열의 자료형)

- 다차원 배열

> **배열의 원소가 배열 자료형인 배열**

> **2차원 배열에서는 열에 대한 인덱스가 행에 대한 인덱스보다 먼저 나온다**

> **물리적인 실제 메모리 구조는 순차적으로 할당된다.**

- 구조체

> **다른 자료형의 데이터를 하나의 그룹으로 묶은 자료형**

> **사용하려면 먼저 구조체에 대한 선언이 필요**

> **자료형 정의를 통해 하나의 독립된 구조를 가지도록 선언해야 함**

> **자료형 선언이라 후에 변수선언도 따로 해줘야 한다.**

> **Typedef를 통해 하나의 데이터타입으로 정의 하기도 한다.**

- 포인터

> **포인터 변수는 메모리주소값을 저장하는 변수이다.**

> **프로그램의 안정성 차원에서 가능한 변수 선언에 항상 NULL값을 넣어 초기화를 시켜 줘야 한다.**

> **포인터 연산**

1) 주소연산자(&) : 변수의 주소값을 얻기 위한 연산자

2) 참조연산자(*) : 포인터 변수에 저장된 주소를 이용해 해당 주소에 있는 값을 얻는 연산자

> **동적 메모리 할당**

- 프로그램 실행중(런타임)j에 임의의 크기로 메모리를 할당 할 수 있다.
- Void* Malloc(int size)을 사용해 할당 가능
  - 파라미터인 size바이트 만큼 메모리를 할당 후 할당된 메모리 블록의 시작 주소를 반환한다.
  - 할당 불가능시 NULL을 반환 → malloc호출후에는 반드시 NULL이 아닌지 검사를 해야 한다.
  - 할당시킨 메모리는 사용 후 반드시 void free(void *ptr)로 할당을 해제해 줘야 한다.
  - free함수의 경우 파라미터로 NULL또는 쓰레기값이 전달 될 경우 ㅈ 될수 있다.
- 초기화 : void *Memset(void *ptr_start, int value, size_t count)를 사용해 초기화 가능
  - String.h를 선언 시켜 줘야 한다.
  - Ptr_start가 가리키는 메모리 영역을 시작으로 Count크기만큼 value로 설정한다.

- **더블 포인터**

  > **다른 변수의 메모리 주소를 저장하는 포인터 변수의 주소를 저장**

  > **포인터와 배열**

  - 배열을 포인터로 나타내는 것 또한 가능하다.
  - 2차 배열의 경우 더블 포인터를 활용해 표현 가능하다.
  - 배열의 이름 = 포인터변수

  Ex) int arr[10]일 때, arr = 포인터 변수 → 값을 더해주는 것으로 다른 인덱스에 접근 가능

  - 2차배열의 이름 = 더블 포인터 변수

  Ex) int arr[a][b]일 때, arr = 더블 포인터 변수

  → 값을 더해주는 것으로 a의 인덱스에 접근 가능

  → 이후 도출된 값에 수를 더해주는 것으로 b인덱스에 접근 가능

  > **구조체에서도 포인터 사용가능**

  - 포인터 참조연산으로 값 도출 시 (.)연산자가 (*)연산자보다 우선순위가 높으니 주의

리스트

- **리스트**

  > **여러 개의 자료가 일직선으로 서로 연결된 선형 구조를 의미**

  > **리스트의 활용 예시**

  1. 문자리스트 : 문자들이 각각 차례대로 저장
     - 리스트에 저장되는 자료는 각각의 문자가 되고, 문자들은 순서대로 선형 구조를 이루게 된다.
  2. 문자열 리스트 : 저장되는 자료가 문자열
     - 제일 마지막 문자는 널문자('₩0')가 오게된다.

3. 행렬 : 행렬은 각각의 열이 행개수만큼의 자료를 가지는 리스트라 볼 수 있다.

4. 다항식 : 리스트를 활용하여 다항식 또한 저장 할 수 있다.

→ 다항식의 계수값을 저장하는 리스트로 표현 가능하다.

→ 계수가 0인 경우도 저장을 해야 한다.

- 리스트 추상자료형

> **리스트 사용에 필요한 추상자료형**

이름		입력	출력	설명
리스트 생성	createList()	최대 원소 개수 n	리스트 l	최대 n개의 원소를 가지는 공백(Empty)리스트 l을 생성
리스트 삭제	deleteList()	리스트 l	N/A	리스트의 모든 원소를 제거
원소 추가 가능여부판단	isFull()	리스트 l	TRUE/FALSE	리스트의 원소 개수가 최대 원소 개수와 같은지를 반환. 배열 리스트인 경우에만 의미가 있음
원소 추가	addElement()	리스트 l 원소 위치 p 원소 e	성공/실패 여부	원소 e를 리스트의 특정 위치 p에 추가
원소 제거	removeElement()	리스트 l 원소 위치 p	성공/실패 여부	리스트의 위치 p에 있는 원소를 제거
리스트 초기화	clearList()	리스트 l	N/A	리스트의 모든 원소를 제거
원소 개수	getListLength	리스트 l	원소의 개수	리스트의 모든 원소개수를 반환
원소 반환	getElement()	리스트 l 원소 위치 p	원소	리스트의 위치 p에 있는 원소를 반환

→ 경우에 따라서 디버깅 용도로 모든 노드를 출력하는 displayList()가 필요 할 수 있다.

- 배열리스트

> **배열을 통해 리스트를 구현한 것**

> **논리적 순서와 물리적 순서가 같다는 특징이 있다.**

> **원소 추가**

→ 배열리스트의 중간에 새로운 원소를 추가하려면 기존의 원소들을 이동 시켜 줘야 한다.

→ 원소의 논리적 순서를 보장하기 위해 물리적인 순서에 제약을 준 것

→ 배열의 가장 오른쪽 원소부터 왼쪽으로 진행

> **원소 제거**

→ 제거하려는 원소의 다음 인덱스부터 시작해 오른쪽으로 진행

> **구현**

→ github.com/Snack1511/DataStructure/tree/master/ArrayList

- 연결리스트

> **포인터를 이용하여 구현**

> **자료의 순서상으로는 연결된 듯 하지만 물리적으로는 서로 떨어져 있다.**

→ 링크에 의해 논리적으로만 연결되어 있음을 시사한다.

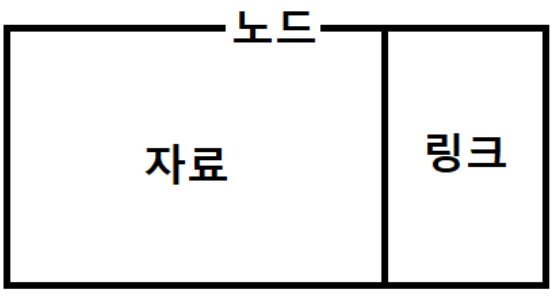
→ 배열리스트와 달리 저장 가능한 최대원소의 개수를 지정 할 필요가 없다.

→ 새로운 원소를 추가할 경우 동적으로 원소를 생성하고 포인터로 이어주면 된다.

> **연결 리스트의 노드구조**

→ 원소 = 노드인 배열리스트와 다르게 노드 = 자료 + 링크 형태로 구성되어 있다.

→ 이는 노드가 원소를 포함하는 개념이다.



→ 위 그림과 같이 노드는 자료를 저장하는 부분과 링크(다음 주소지)를 저장하는 부분으로 나뉘어 있다.

1) 자료 : 정수와 실수 같은 단순데이터 뿐만 아니라 복잡한 구조체 정보도 저장 할 수 있다.

2) 링크 : 포인터를 이용하여 현재 노드와 연결된 다음 노드를 가리킨다.

→ → 제일 마지막 노드는 다음 노드가 더 없으므로 NULL값을 가리킨다.

> **노드추가**

→ 기존에 연결된 링크를 제거하고 새로 추가되는 노드의 링크를 기존의 노드들과 연결시켜 준다.

> **노드제거**

→ 기존의 링크를 제거하고 다음 노드를 연결시켜준다

→ 연결이 끊긴 노드는 메모리를 해제해서 메모리 누수가 발생 하지 않도록 하는 것이 중요하다.

> **연결리스트의 장점**

1) 추가 원소 이동 연산 불필요

→ 배열리스트의 경우 모든 원소를 이동시키는 연산이 반드시 필요하다.

→ 연결리스트는 링크만 새로 연결 하면 되므로 이동연산의 제약에서 자유로워 진다.

2) 최대 원소 개수 지정 불필요

→ 배열리스트는 생성 시 반드시 최대 원소 개수를 지정해줘야 한다.

→ 연결리스트는 동적으로 메모리를 할당하므로 제약에서 자유롭다.

> **연결리스트의 단점**

1) 구현이 어렵다.

→ 동적인 메모리 할당 및 포인터 연산등으로 배열리스트보다 구현의 비용이 높다.

→ 메모리 관리와 관련해서 메모리 누수오류의 발생 가능성이 높다.

2) 탐색연산의 비용이 높다.

→ 배열리스트는 특정 원소에 대한 탐색은 O(1)의 시간복잡도를 가지는 반면 연결리스트는 O(n)의 시간복잡도를 가진다.

→ 이는 연결리스트의 경우 원하는 원소를 찾을 때까지 포인터로 노드를 탐색해야 하기 때문에 시간 비용이 높아질 수 밖에 없다.

> **연결리스트의 종류**

1) 단순 연결리스트

- 연결리스트의 가장 기본이다.
- 첫 노드부터 끝 노드까지 일직선으로 구현하기에 간단한 구조를 지닌다.
- 이전노드로의 링크가 없으므로 이전 노드 탐색 시 새로운 순회를 시작해야 한다.

2) 원형 연결리스트

- 연결리스트의 마지막 노드가 첫 노드를 가리키는 원형의 형태를 가진다.
- 이전노드를 탐색하려면 순회를 지속하면 된다.
  - 이 경우 전체 리스트를 한번 순회해야 한다.

→ 단순 연결리스트와 원형 연결리스트는 링크가 단 방향이라는 것에 주의하자.

3) 이중 연결리스트

- 노드 사이의 링크가 양방향으로 구성되어 있다.
- 이로 인해 이전 노드에 대한 직접 접근이 가능하다.

- 단순연결리스트

> **노드의 구조체 = 현재 노드 개수 + 헤더노드**

> 구현

- 원형연결리스트

> **리스트의 마지막 노드가 리스트의 첫 번째 노드와 연결된 단순연결리스트**

> **마지막 노드가 첫 번째 노드와 연결되어 순환 구조를 지님**

> **원형 연결리스트는 링크를 따라 이동하면 이전 노드에 접근 할 수 있다.**

→ 현재 노드의 이전노드가 현재노드의 다음 노드인지 확인하면 이전노드인지 판별 가

능

> **헤더포인터를 사용한 구현**

→ 헤더 포인터를 사용한 경우 추가 제거 기능에서 여러가지 고려해야 할 사항이 발생할 수 있다.

> **추가**

→ 헤드포인터를 이용하여 고려 시 3가지 경우를 생각해야 한다.

1) 리스트의 첫 번째 노드로 추가하는 경우

1-1) 공백리스트인 경우

1-2) 공백리스트가 아닌 경우

2) 리스트의 중간에 노드를 추가하는 경우

> **제거**

→ 이 또한 3가지 경우를 고려 해야 한다.

1) 리스트의 첫 번째 노드를 제거하는 경우

1-1) 제거하려는 노드가 리스트의 마지막 노드인 경우

1-2) 마지막 노드가 아닌 경우

2) 중간 노드를 제거하는 경우

> **구현**

- 이중연결리스트

> **각각의 노드마다 2개의 링크가 있기 때문에 특정 노드의 다음 노드뿐 아니라 이전 노드에도 직접 접근 가능**

> **이전노드에 직접적인 접근은 장점이나, 이 때문에 추가 메모리 공간을 더 사용한다는 단점이 존재**

> **마지막 노드가 첫 번째 노드를 가리키는 원형 연결 리스트의 속성을 지닌다.**

> **헤더노드를 포함하는 이중연결리스트는 임의의 위치에 있는 노드pNode에 대해 다음식이 성립한다.**

→ pNode == pNode->pLLink->pRLink == pNode->pRLink->pLLink

→ 이 식은 헤드포인터로는 성립될 수 없다.

> **생성**

→ 생성 시 헤더 노드에서 좌우 링크 모두 자기자신을 참조하도록 초기화시켜줘야 한다.

> **추가**

→ 노드가 추가될 적절한 위치를 찾아서 새로 생성한 노드를 삽입만 하면 된다.

> **제거**

→ 삭제하려는 노드의 이전 노드를 찾고 삭제하려는 노드의 좌우 링크를 재설정 해준다.

> **구현**

- 연결리스트의 응용

> **순회**

→ 노드 사이에 연결된 노드의 링크를 이용해, 보다 효율적으로 리스트 순회 가능

> **다른 리스트 연결**

→ 연결 주체 리스트에 연결하려는 리스트를 연결한 후 연결하려는 리스트의 링크를 해제해줘야 한다.

→ 각 리스트들의 노드개수 또한 수정해줘야 한다.

> **역순**

→ 현재 노드와 현재노드의 이전노드가 있다면 현재노드의 다음노드로의 링크를 이전노드로의 링크로 설정해주고, 현재노드의 다음노드로 이동한다.

1) B = A

2) A = C

3) C = C->link

4) A->link = B

> **다항식**

→ 각 항을 노드로 나타내고, 이런 항들의 집합인 다항식을 단순연결리스트로 구현

→ 각 항의 계수와 차수를 멤버변수를 이용해 저장하되, 차수의 내림차순으로 저장하고, 저장공간을 절약하기 위해 계수가 0인 항은 노드를 추가하지 않는다.

→ 다항식의 덧셈연산 : 3가지 경우를 고려

1) 다항식들 주체식의 차수가 높은경우 : 주체식에 대해서만 다음노드로 이동

2) 다항식들 중 주체식의 차수가 낮은 경우 : 더해주려는 식만 다음노드로 이동

3) 같은 경우 : 다항식들 전체의 노드 이동

→ 항들 끼리 더해준 후 남은 항들은 마지막에 연산

→ 연산 후 식들은 모두 메모리를 해제해줘야 한다.

> **옵션함수들 구현**

> **다항식 구현**