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.

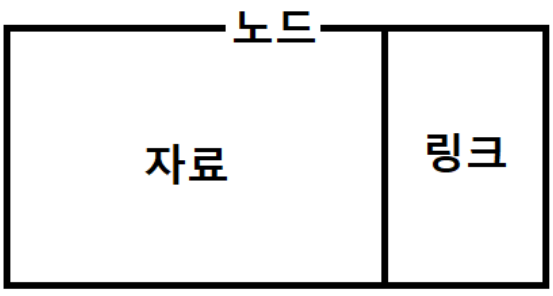
→ 배열리스트와 달리 저장 가능한 최대원소의 개수를 지정 할 필요가 없다.

→ 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**

스택

- **스택**

> **자료를 쌓아두는 기능**

> **특성**

→ Last In First Out

→ 선형 자료구조 : 저장된 자료들 사이의 선후관계가 모두 1대1

> **제약사항**

→ 자료의 추가 및 반환은 스택의 끝에서만 가능

→ 스택의 끝이란 스택의 제일 위→ 가장 최근에 추가된 자료가 있는곳

→ 후입선출의 특성을 지니는 이유

> **후입선출의 특성은 다양한 알고리즘에서 필수적인 요소로 사용**

→ 수식을 해석하고 이를 계산하는데 사용

→ 복잡한 미로에서 길을 찾는 알고리즘등으로 사용

→ 트리나 그래프등 다른 복잡한 자료구조에서도 내부적으로 사용

> **푸쉬(Push)**

→ 새로운 자료를 스택에 추가하는 것

→ 스택의 제일 위에서만 발생

→ 스택의 크기는 스택이 저장할 수 있는 최대 자료의 개수를 의미

→ 스택의 크기를 초과해 새로운 자료를 추가할 때 자료가 추가될 수 없는 현상을 넘침(Overflow)현상이라 한다.

→ 탑을 어떻게 변경해야 할지를 가장 주의깊게 고려해야 한다.

> **팝(Pop)**

→ 스택에 저장된 자료를 사용하기 위해 스택에서 자료를 꺼내는 것

→ 스택의 제일 위에서만 발생

→ 스택에 아무 자료가 저장되지 않은 상태를 공백(Empty)이라 한다.

→ 공백상태에서 팝연산이 호출되면 부족(Underflow)현상이 발생

> **피크(Peek)**

→ 팝 연산과 달리 자료를 제거하지 않고 해당자료에 접근만 하는 것

→ 최상위 자료를 반환하는 점에서 팝연산과 비슷하지만 스택에서 자료를 제거하지 않고 반환만 시킨다.

> **추상자료형**

이름		입력	출력	설명
스택 생성	createStack()	스택의 크기 n	스택 s	최대 n개의 원소를 가지는 공백(Empty)스택 s를 생성
스택 삭제	deleteStack()	스택 s	N/A	스택제거(메모리 해제)
원소 추가 가능여부판단	isFull()	스택 s	TRUE/FALSE	스택의 원소 개수가 최대 원소 개수와 같은지를 반환. 배열 스택인 경우에만 의미가 있음

공백스택인지 여부 판단	isEmpty()	스택 s	TRUE/FALSE	공백 스택인지 여부를 전달
푸시	Push()	스택 s 원소 e	성공/실패 여부	스택의 맨 위에 새로운 원소를 추가
팝	Pop()	스택 s	원소 e	스택의 맨 위에 있는 원소를 제거한 뒤 이를 반환
피크	Peek()	스택 s	원소 e	스택의 맨 위에 있는 원소를 반환 (스택에서 제거하지는 않음)

→ createStack()과 deleteStack()은 각각 스택을 생성 및 삭제하는 연산이다.

→ isFull은 스택의 크기를 초과하는 overflow가 발생할지에 따라 원소가 추가 가능한지를 알려준다.

→ 구현 시 최대 저장 원소 개수라는 제약조건이 필요하지만, 스택을 어떻게 구현할지에 따라 필요가 없을 수도 있다.

→ isEmpty는 현제 스택이 공백상태인지 확인하는 기능이다.

→ 스택의 가장 최근 자료를 반환하는 팝/피크 연산은 스택에 있어서 가장 기본적인 연산이다.

→ 스택의 구현 방법은 배열을 이용한 방법과 포인터를 이용한 방법으로 구분할 수 있다.

> **배열로 구현한 스택**

→ C언어에서 사용하는 배열을 이용하여 스택을 구현한 것

→ 물리적으로 연결된 C의 배열을 이용하기 때문에 스택을 생성할 때 스택의 크기를 반드시 지정해 주어야 한다.

→ 배열로 구현한 스택은 스택의 크기(최대노드개수)를 내부변수로 저장하고 있다.

→ 스택을 생성해 줄 때에 스택의 크기에 해당하는 개수만큼 노드의 배열을 생성해준다.

→ 스택에 저장된 현재 노드 개수 또한 내부변수에 저장

→ 이를 이용해 스택에서의 탑의 위치를 알수있다. → 탑의 위치 = 현재노드개수 – 1

> **배열스택 구현**

> **연결리스트로 구현한 스택**

→ 배열로 구현시에는 스택의 크기를 미리 지정해야 하고, 최대저장수보다 많은 자료를 저장하려고 할 때 오버플로우가 발생

→ 연결리스트로 스택을 구현할 때에는 노드 추가 때마다 동적으로 메모리를 할당하기에 스택크기를 미리 지정해 줄 필요가 없다.

→ 보다 효율적인 메모리 사용이 가능

→ 탑노드를 포인터 변수가 기리키고 있기 때문에 포인터 변수로 직접 접근 할 수 있다.

→ 노드 추가시에는 새로운 노드를 연결리스트에 추가하고 탑을 가리키는 포인터변수는 새로 추가된 노드를 가리키도록 수정해야 한다.

→ 또한 새로운 탑 노드의 다음노드가 기존의 탑 노드가 되도록 노드사이의 링크를 설정해줘야 한다.

→ 정리 : 연결리스트로 구현한 스택은 최대저장개수로 인한 오버플로우가 발생하지 않는 대신 구현이 어렵다

> **이전에 했던 연결리스트를 조금 변형시키면?**

→ 스택은 탑에서만 자료를 추가/제거 할 수 있기 때문에 범용적인 리스트를 직접 이용하는 것보다 스택에 맞춰 구현해 주는게 훨씬 효율적이다.

→ 기존의 연결리스트를 활용할 경우 가장 마지막에 추가된 노드에 접근하기 위해 리스트 전체를 순회해야 한다.

→ 연결리스트스택의 경우 가장 마지막에 추가된 노드가 스택구조체 내에서 탑포인터에 의해 링크되어 있어서 접근이 쉽다.

→ 기존의 연결리스트와 다르게 추가순서와 연결방향이 반대라는 것 또한 차이점이다.

→ 가장마지막에 추가된 자료의 다음 노드는 그 이전에 추가된 노드가 되고, 스택에 맨 처음 추가된 노드는 다음 노드값이 NULL이 된다.

> **연결리스트스택 구현**

- 스택응용1

> **역순문자열**

→ 입력받은 문자열의 순서를 뒤집은 문자열을 출력하는 함수

→ 스택을 활용해 비교적 간단하게 구현 가능

1) 문자열의 모든 문자를 순서대로 스택에 푸시

2) 공백스택이 될 때까지 스택의 모든 노드를 팝

3) 정반대 순서대로 문자들이 스택에서 추출

> **수식괄호검사**

→ 2가지를 점검해야한다.

1) 여는 괄호와 닫는 괄호가 서로 쌍을 이루는가?

여는 괄호를 만날 때 스택에 푸시 하다, 닫는 괄호를 만날 때마다 팝을 해주면 쉽게 구현 할 수 있다.

2) 여러 개의 괄호가 중첩 될 경우 차례대로 처리되는가?

스택의 후입선출 제약을 이용하면 검사할 수 있다.

> **스택응용1 구현**

- 수식계산과 표기법 변환

> **표기법**

→ 중위 표기법 : a * b

→ 후위표기법 : a b +

→ 변환1 : A*(B+C) -> A B C + *

→ 변환2 : A * B+ C -> A B * C +

→ 후위 표기수식 계산시 3가지 원칙

1) 피연산자를 마나면 스택에 푸시

2) 연산자를 만나면 연산에 필요한개수만큼 피연산자를 스택에서 팝

3) 연산한 결과는 다시 스택에 푸시

→ 스택에 저장된 유일한 노드에서 최종계산 결과를 팝시키면 정상종료

→ 결과 노드가 없다면 수식에 오류

> **지금부터 피연산자와 연산자를 묶어서 요소 혹은 토큰이라 칭한다.**

> **후위 표기수식은 중위 표기와는 달리 수식을 따라 단계별로 토큰을 처리**

> **토큰타입선언**

→ 연산자들의 종류를 열거형타입으로 정의

→ 숫기의 각 단위인 토큰을 표현하기 위해 구조체 선언

→ 데이터 값과 연산자 종류를 멤버변수로 가짐

> **중위에서 후위표기로의 전환**

→ 기본규칙 : 피연산자를 만나면 바로 출력, 연산자를 만나면 일단 스택에 보관

→ 연산자 우선순위 : 스택내부와 스택 외부에서의 연산자 우선순위에 차이가 있다.

→ 연산자 "("는 스택안에서는 우선순위가 가장 낮은 연산자이다.

→ 중위 표기에서 후위표기로의 변환 과정 도중에 연산자 ")"를 만나게 되면 스택에서 여는 괄호를 만날 때까지 스택에 저장된 모든 연산자를 팝시켜야 한다.

→ 중위->후위표기 변환시 적용되는 5가지 규칙

1) 피연산자를 만나면 바로 출력
2) 연산자를 만나게 되면 일단 스택에 보관
3) 단, 스택보관중인 연산자 중에서 우선순위가 높은 연산자는 출력
4) (주의)스택의 내부와 외부에서의 연산자 우선순위는 다르다.
5) 닫는 괄호연산자")"를 만나게 되면, 스택에서 여는 괄호"("를 만날 때까지 스택에 저장된 모든 연산자들을 팝 시키고 이를 출력

> **스택응용2 구현**

- 미로찾기

> **입구와 출구가 주어진 미로에서 입구부터 출구까지의 경로를 찾는 문제**

> **스택의 노드가 위치정보를 저장하도록 디자인**

> **알고리즘**

→ 스택에 기존에 방문한 위치 정보를 차례로 저장하는 방식을 통해 경로를 찾는 알고리즘

→ 입구 – 출구 까지의 경로를 출력하기 위해서는 어떤 경로를 통해 탐색했는지 정보가 저장되어야 한다. → 위치정보의 저장에 스택을 이용

→ 특정방향으로 탐색을 계속 진행하다가 더 이상 진행 할 수 없을 시, 이전의 이정표에 해당하는 위치로 돌아와서 다른 방향으로 탐색

→ 제약사항 : 무한루프가 발생하지 않기 위해서 미로의 위치별로 기존에 방문했던 곳인지를 저장할 필요가 있다.

→ 정리

1) 미로찾기 알고리즘에서는 기존 위치를 저장하고 다른 방향으로 재탐색 하기 위해 스택을 이용한다.

2) 미로의 탐색단계마다 방문위치 정보와 이동방향 정보를 스택에 푸시한다.

3) 출구 탐색에 실패한다면 다시 이전 위치로 돌아오기 위해 스택에 팝을 실시한다.

> **스택응용3 구현**

큐

- 큐의 개념

> **선입선출(First in First Out) 특성을 지니는 선형자료구조.**

- 특징

> **선입선출 : 추가되는 자료를 차례대로 저장 후 추가된 순서대로 반환시킨다.**

> **선형구조 : 큐에 있는 모든 자료는 각각 자신의 앞에도 1개의 자료만 존재하고 뒤에도 1개의 자료만 존재한다.**

> **큐의 제일 앞은 프런트 혹은 전단이라 부르고 제일 뒤는 리어 혹은 후단이라고 한다.**

- 스택과의 비교

> **스택의 탑과 큐의 리어는 자료가 추가되는 리스트의 제일 끝이라는 점에서 공통점을 보인다.**

> **스택은 탑(첫 부분)에서 자료를 반환하는 반면 큐는 리어(끝 부분)에서 자료를 반환한다.**

- 인큐

> **새로운 자료를 큐에 추가하는 것**

> **큐의 최대자료수를 넘어 추가하면 오버플로우 현상이 발생한다.**

> **추가 시 큐의 리어가 가장 나중에 삽입된 자료를 가리킨다.**

- 디큐

> **큐에서 자료를 꺼내는 것**

> **큐의 프런트가 가리키는 자료를 반환한 이후 그 다음 자료를 가리킨다.**

> **한 개의 자료만 저장된 상태에서는 프런트와 리어가 모두 한 자료를 가리킨다.**

→ 의미하는 바

1) 해당 자료가 현재 큐에서 유일한 원소이다.

2) 현재 큐에 있는 자료 중 가장 오래된 자료이다.

3) 현재 큐에 있는 자료 중 가장 최근 자료이다.

→ 반환 시 프런트와 리어는 공백자료를 가리키게 된다.

> **큐에 아무런 자료가 저장되어 있지 않은 상태를 공백상태라 하며, 이 경우에 디큐 연산 시 언더플로우 현상이 발생한다.**

- 피크

> **스택과 마찬가지로 기존 자료를 제거하지 않고 반환만 하는 특성이 있다.**

> **큐의 프런트 자료에 접근하는 점에서는 디큐와 같지만, 실제 큐에서 자료를 반환 시키지는 않는다는 차이점이 존재한다.**

- 추상자료형

이름		입력	출력	설명

큐 생성	createQueue()	스택의 크기 n	큐 q	최대 n개의 원소를 가지는 공백(Empty) 큐 q를 생성
큐 삭제	deleteQueue()	큐 q	N/A	큐 제거(메모리 해제)
원소 추가 가능여부판단	isFull()	큐 q	TRUE/FALSE	큐의 원소 개수가 최대 원소 개수와 같은지를 반환. 배열 큐인 경우에만 의미가 있음
공백 큐 여부 판단	isEmpty()	큐 q	TRUE/FALSE	공백 큐인지 여부를 전달
인큐	Enqueue()	큐 q 원소 e	성공/실패 여부	큐의 리어에 새로운 원소를 추가
디큐	Dequeue()	큐 q	원소 e	큐의 프런트에 있는 원소를 제거한 뒤 이를 반환
피크	Peek()	큐 q	원소 e	큐의 프런트에 있는 원소를 반환 (큐에서 제거하지는 않음)

> **구현 방법에 따른 구분**

1) 배열 : 구현의 복잡도가 낮은 대신 고정된 큐의 크기로 인해 불필요한 메모리 사용량이 증가

2) 리스트 : 구현의 복잡도가 높은 대신 큐의 크기를 가변적으로 추가/삭제할 수 있기에 메모리 사용 효율 면에서는 우수함

- 배열 선형 큐

> **배열을 이용해 큐를 구현한 것**

> **선형 큐는 인큐/디큐를 반복함에 따라 배열에 비어 있는 곳이 있음에도 이를 인식 하지 못하는 경우가 발생한다. → 이는 비효율적인 메모리 사용으로 이어진다.**

> **보통 배열로 구현한 원형 큐 또는 연결리스트로 구현한 큐를 많이 사용**

> **선형큐와 원형 큐모두 같은 구조체를 사용하지만 함수 내부의 알고리즘에서 차이가 발생한다.**

> **배열은 동적할당으로 생성되며 큐 구조체에서는 이를 가리키는 포인터변수를 가지고 있다.**

> **큐에 저장된 현재 노드의 개수를 내부변수로 저장하고, 배열에서의 프런트와 리어 위치를 각각 내부변수로 저장한다.**

> **구현**

- 배열 원형 큐

> **선형 큐에 최대개수까지 인큐하고, 이후 디큐한후 다시 인큐를 하면 오버플로우 현상이 발생**

> **프런트 쪽의 빈노드가 있음에도 리어가 이동할 오른쪽에 빈 노드가 없기 때문에 넘침 현상이 발생한 것**

> **대안으로 배열을 이동시키는 방법을 생각 할 수는 있으나 배열의 크기에 비례해 시간복잡도가 O(n)만큼 증가한다. → 더욱 효과적인 방법이 필요**

> **배열의 오른쪽을 배열의 왼쪽과 연결시킨다면 효율적일 것**

> **리어의 오른쪽과 프런트의 왼쪽을 연결 시킨 것이 원형 큐**

◆ **리어와 프런트의 이동에 mod연산자를 이용하면 구현 가능**

> **인큐**

→ 리어 변수 증가시 : (rear + 1)%maxElementCount

> **디큐**

→ 프런트 변수 증가시 : (front +1)%maxElementCount

> **display함수**

→ mod연산자 때문에 rear의 값이 front보다 작을 수 있기 때문에 리어를 인덱스 계산에 쓰는 것이 아니라 프런트를 기준으로 계산한다.

> **구현**

- 연결리스트 큐

> **단순연결리스트의 제일 앞부분이 프런트, 제일 뒷부분이 리어**

> **새로운 자료 추가 시 리어에 추가하고, 기존 자료를 반환할 때는 가장 앞 프런트에 있는 노드를 제거한 뒤 이를 반환한다.**

> **각각의 노드는 자신의 앞, 뒤 노드와 1대1의 선형관계로 서로 연결되어 있다.**

> **자료의 방향은 프런트에서 리어 순서로 이어진다.**

> **배열큐와의 차이점**

→ 저장 노드 개수 초과로 인한 넘침현상이 발생하지 않는다는 장점이 있다.

→ 하지만 동적으로 메모리를 할당하기 때문에 구현 시 난이도가 있다.

> **구현시 주의점**

→ 연결큐의 제일 마지막인 리어노드의 경우 pLink값이 NULL이다.

> **인큐**

→ 기존의 연결리스트와 유사하게 이미 저장된 자료가 있는 지에 따라 인큐로직에 다소 차이가 있다.

1) 공백일때 : 기존의 프런트와 리어모두 NULL을 가리키고 있다. 이 상태에서 새로운 자료를 추가하면 프런트 노드와 리어노드가 모두 새로 추가된 노드를 가리키도록 수정해야 한다.

2) 공백이 아닐 때 : 새로추가되는 노드를 리어의 다음 노드로 설정한다. 그 다음 리어노드의 포인터를 새로추가된 노드를 가리키도록 재설정한다.

> **디큐와 피크**

→ 디큐를 통해 전달 받은 노드는 기존 큐의 연결리스트에서 제거된 노드이므로 전달받은 쪽(main)에서 노드의 메모리 해제(free)를 책임지지만, 피크는 큐에서 해당노드를 제거한 것이 아니기 때문에 전달받은 쪽에서의 해제에 대한 책임이 없다.

> **디큐**

→ 기존 연결리스트에서 제거하는 연산이 필요하기 때문에 조금 복잡하다.

1) 일반적인 경우 : 기존 큐의 프런트 노드가 연결리스트에서 제거되어 포인터 변수로 반환된다. 또한 기존 프런트노드의 다음 노드가 큐의 새로운 프런트 노드로 설정된다. 반환 되는 포인터 변수 또한 기존의 연결을 NULL로 초기화 시켜줘야 한다.

2) 노드가 1개 남은 경우 : 새로운 노드에 대한 처리는 일반적인 경우와 동일하나, 리어노드의 경우 마지막 남은 노드를 처리하는 과정이라 NULL을 가리키도록 해야 한다.

> **연결리스트 큐 구현**

- 연결리스트 덱

> **덱(Deque)이란?**

→ 두 개의 끝을 가지는 큐(Double-Ended Queue)

→ 양쪽 끝에서 자료의 삽입과 반환이 모두 가능한 선형자료구조

→ 이런 특성 때문에 큐와 스택의 기능을 합친 자료구조로도 설명될 수 있다.

> **덱과 큐, 스택 비교**

연산위치	연산	덱	스택	큐
앞, 프런트	추가	InsertFront	X	X
	반환	DeleteFront	X	Dequeue
뒤, 리어	추가	InsertRear	Push	Enqueue
	반환	DeleteRear	Pop	X

→ 스택의 두 연산 Push/Pop은 덱의 InsertRear/DeleteRear와 매칭된다.

→ 큐의 두 연산 Enqueue/Dequeue또한 InsertRear/DeleteFront로 매칭된다.

> **덱은 기존의 스택 혹은 큐에서 제공하지 않던 기능을 제공한다.**

→ InsertFront

> **덱은 스택과 큐를 일반화 시킨 포괄적인 자료구조로 해설 될 수 있다.**

→ A-Steal스케줄링 알고리즘과 같이 몇몇 알고리즘에서 덱의 이러한 포괄적인 기능이 요구되기도 한다.

> **추상자료형**

이름		Input	Output	설명
덱 생성	CreateDeque()	덱의 크기n	덱 d	최대 n개의 원소를 가질 수 있는 공백(Empty) 덱 d생성

덱 삭제	DeleteDeque()	덱 d	N/A	덱을 제거(메모리 해제)
원소추가 가능 여부 판단	isFull()	덱 d	TRUE/FALSE	덱의 원소 개수가 최대 언소 개수와 같은지를 반환
공백 판단	isEmpty()	덱 d	TRUE/FALSE	공백 덱인지를 전달
앞 추가	InsertFront()	덱 d 원소e	성공/실패여부	덱의 맨 앞(Front)에 새로운 원소를 추가
뒤 추가	InsertRear()	덱 d 원소e	성공/실패여부	덱의 맨 뒤(Rear)에 새로운 원소를 추가
앞 제거	DeleteFront()	덱 d	원소e	덱의 맨 앞(Front)에 있는 원소를 제거 한 뒤 반환
뒤 제거	DeleteRear()	덱 d	원소e	덱의 맨 뒤(Rear)에 있는 원소를 제거한 뒤 반환
앞 반환	PeekFront()	덱 d	원소e	덱의 맨 앞(Front)에 있는 원소를 반환(제거X)
뒤 반환	PeekRear()	덱 d	원소e	덱의 맨 뒤(Rear)의 원소를 반환(제거X)
덱 표시	displayDeque()	덱 d	N/A	덱에 저장된 원소를 화면에 표시

> **덱은 구현 방법에 따라 배열을 이용하거나 연결리스트를 이용하는 두가지 방법으로 구현 될 수 있음**

> **지금은 이중연결리스트를 이용해서 구현**

→ 이중연결리스트를 사용한면서 헤드포인터를 사용해 구현

> **선언**

→ 헤드포인터를 이용하기 때문에 두개의 해드포인터(Front, Rear)에 대한 변수를 가지고 있다.

→ "헤더노드"를 이용하는 이중연결 리스트는 임의의 위치에 있는 노드 pNode에 대해 항상 다음식이 성립한다.

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

→ 헤드 포인터를 이용하여 구현하는 이중 연결리스트는 위 식이 성립하지 않는다.

→ 따라서 헤드포인터를 이용하여 이중연결리스트를 구현하면, 노드의 추가와 삭제 시에 고려해야 할 경우의 수가 많아짐을 유념해야 한다.

> **InsertFront/Rear**

→ 덱의 상태가 공백인지에 따라 처리 과정에서 차이가 있다.

1) 공백일때 : 프런트노드와 제일 뒤쪽 리어노드가 모두 이 새로 추가된 노드를 가리키도록 한다.

2) 공백이 아닐 때 :

2-1) 새로운 노드가 기존 프런트 노드의 왼쪽 노드로 설정된다.

2-2) 새로운 노드의 오른쪽 노드는 기존의 프런트 노드가 차지한다.

2-3) 이후 새로 추가된 노드를 덱의 프런트 노드로 설정해 줘야 한다.

→ Rear의 경우 Front를 Rear로 대체하고 반대방향으로 연결해준다.

> **DeleteFront/Rear**

→ delete연산을 통해 전달되는 노드들은 사용이 끝난 후 반드시 메모리를 해제 시켜 줘야 한다.

→ 덱의 노드가 1개인지 1개이상(일반적인 경우)인지에 따라 차이가 발생한다.

1) 일반적인 경우

1-1) 반환되는 노드에 덱의 프런트노드를 대입한다.

1-2) 덱의 새로은 프런트 노드는 기존 프런트 노드의 오른쪽 노드로 대체한다.

1-3) 반환되는 노드의 오른쪽링크 정보는 NULL로 초기화 시킨다.

1-4) 덱의 새 프런트 노드는 왼쪽 링크정보를 NULL로 설정해준다.

2) 노드가 1개인경우 : 남은 노드 1개를 제거했을 때, 프런트 노드는 일반적인 경우와 차이기 없지만 리어 노드의 경우 NULL로 설정해 줘야 한다.

→ Rear의 경우 Front를 Rear로 대체하고 반대방향으로 연결해준다.

> **연결리스트 덱 구현**

- ## 큐의 응용_시뮬레이션

  > **시뮬레이션이란? : 현실 시스템에 대한 모델을 설계하고, 컴퓨터 기반의 실험을 통해, 시스템의 현상을 예측하는 일련의 과정**

  > **시뮬레이션을 구현하는 방법에는 많은 방법이 존재하지만, 시스템에서 시간을 일정하게 증가시키면서, 각각의 시각마다 발생하는 이벤트들을 큐에 저장하고 이를 처리하는 방식을 채택 할것이다.**

  > **하나의 노드를 고객으로 잡고, 노드의 멤버로 도착시간과 서비스시간을 정의한다.**

  > **큐를 통해 대기열, 서비스를 제공할 은행원, 고객의 방문순서를 표현한다.**

  > **시뮬레이션 시작시 도착시각이 미리 정의된 고객들은 시간 순서에 의해 고객의 방문순서를 기록한 큐에 미리 삽입되어 있다고 가정한다.**

  > **구현**

- ## 연습문제

  > **시뮬레이션에서 은행원이 1명 더 추가되었을 때를 구현하자**

  > **구현**

재귀호출

- ## 개념

  > **자기 자신을 호출하는 것**

  > **비선형 자료구조에 기반을 둔 다양한 알고리즘에 사용된다.**

  → 트리 혹은 그래프에 있는 모든 노드를 탐색하거나 다원탐색트리에서 노드를 삭제하거나 병합할 때 사용한다.

  → 문제 해결을 위한 접근방식으로 사용되기도 한다.

  ◆ 재귀적 접근방법은 일단 큰 문제를 여러 개의 작은 문제들로 나누는 것으로 시작

  ◆ 재귀적 접근 방법은 이처럼 하나의 큰 문제를 여러 개의 작은 문제들로 나누는 방

법인 것

- ◆ 이때 나누어진 작은 문제는 모두 같은 타입이어야 한다는 제약이 존재
- ◆ 종료조건 또한 재귀적 접근방법의 다른 특성으로 더 이상 나눌 수 없는 상태에 이를 때 재귀 호출을 종료하고 값을 반환한다.
- ◆ 재귀적 접근방법은 여러 개의 문제들로 쪼개어 해결한다는 점에서 분할정복의 한 방법으로 볼 수 있다.
- ◆ 단순히 자기자신을 호출 할 뿐 아니라 호출 될때마다 문제가 더 작아지는 특징이 존재

→ 성능상의 몇가지 제약사항 때문에 문제해결방법이 어느정도 고정된 후에는 비 재귀적인 방법(반복적인 방법)으로 다시 수정되기도 한다.

> **재귀호출의 내부적 구현**

→ 실제 운영체제에서는 스택을 이용해 재귀호출을 가능케 한다.

→ 호출 시 함수에서 사용되는 모든 지역변수와 전달인자를 저장하는 공간을 활성 레코드라 하는대, 운영체제에서는 하나의 함수가 호출되면, 함수별로 이러한 활성화 레코드에 함수관련 정보를 저장한다.

→ 재귀호출 종료시 활성레코드를 팝하여 처리하는 과정은 문맥변경(ContextSwitch)가 필요하며, 이는 운영체제가 기존에 수행되던 함수의 지역변수와 전달된 인자를 대신하여 새로운 함수의 지역변수와 인자를 적재하기 때문에, 이를 처리하는 과정에서 시간이 소요될 수 있다.

→ 따라서 같은 기능을 수행하는 함수의 경우 반복호출을 사용하는 함수가 재귀호출을 사용하는 함수보다 수행시간이 더 빠를 수 있다.

> **재귀호출의 종료조건**

→ 기본경우 혹은 최소한계라 부른다.

→ 만약 재귀호출에서 문제범위의 최소한계를 정하는 부분이 없다면 시스템 스택이 허용할 대까지 계속 순환적으로 호출 하다가 메모리 에러를 발생시키며 끝날 것이다.

> **재귀호출과 반복호출**

→ 재귀호출과 비교되는 방법으로 반복호출이 존재

→ 정해진 반복 횟수 혹은 완료조건을 만족할 때까지 반복적으로 계산을 실시하는 것이 반복 호출의 기본 개념이다.

→ 재귀호출이 반복호출과 비교하여 가지는 장점은 알고리즘의 간결성과 명화성이다.

→ 그러나 시스템 스택을 사용하므로 수행시간이 더 오래걸리며, 스택 메모리 문제가 발생할 수 있다.

> **재귀 호출 의 예**

→ 팩토리얼

→ 피보나치

→ 하노이탑

트리

- 트리의 개념

> **트리는 노드와 간선의 집합니다.**

→ 노드 : 일반적으로 모델링하려는 시스템의 객체를 나타낸다.

→ 간선 : 모델링한 객체들 사이의 부모-자식 관계를 정의한다.

> **노드의 구분**

구분	용어	내용
트리에서의 위치	루트(Root)노드	트리의 첫번째 노드
	단말(Leaf or Terminal)노드	자식노드가 없는 노드
	내부(Internal)노드	자식노드가 있는 노드
노드사이의 관계	부모(parent)노드	부모노드와 자식노드는 간선으로 연결되어 있음
	자식(Child)노드	
	선조(Ancestor)노드	루트 노드로부터 부모노드까지의 경로상에 있는 모든 노드
	후손(Descendant)노드	특정 노드의 아래에 있는 모든 노드
	형제(Sibling)노드	같은 부모노드의 자식노드

> **노드의 속성**

용어	내용

레벨(Level)	루트 노드로부터의 거리
높이(Height)	루트노드부터 가장 먼 거리에 있는 자식 노드 높이에 1을 더한 값
차수(Degree)	한 노드가 가지는 자식 노드의 수

> **트리의 특징**

→ 계층구조로 자료를 저장

◆ 계층구조란 트리를 구성하는 노드가 부모-자식관계라는 의미 → 특정부모 노드 하나에 여러 개의 자식 노드들이 연결되는 구조

→ 계층구조(하나의 노드에 여러 자식노드가 연결된 구조)인 점에서 비선형 자료임을 알 수 있다.

→ 노드 하나가 여러 자식을 가질 수 는 있지만, 반대로 각 노드의 부모노드는 모두 1개라는 점을 주의해야 한다.

> **트리의 기타 용어**

→ 서브트리 : 트리에 속한 노드들의 부분집합을 서브트리라 한다.

→ 포리스트(Forest) : 트리의 집합 → 여러 개의 루트 노드가 존재하기도 한다.

> **일반적인 트리의 경우 자식노드에 개수제한이 없는 반면, 최대 2개까지 제한을 가진 트리가 존재 → 이진트리**

- 이진트리

> **이진트리란?**

→ 모든 노드의 차수가 2이하인 트리를 말한다.

> **특성**

→ 이진트리의 각 노드는 3가지 경우가 존재한다.

1) 차수가 2(왼쪽, 오른쪽 자식 모두 가지는 경우)이거나

2) 차수가 1(왼쪽 오른쪽 중 하나만 가지는 경우)이거나

3) 차수가 0(단말노드인 경우)

→ 이진트리의 모든 서브트리들은 이진트리이다.

◆ n개의 노드를 가지는 이진트리는 모두 (n-1)개의 간선을 가지는데, 루트노드를 제외

한 나머지 노드들이 모두 부모-자식 관계에 의해 간선이 있기 때문이다.

◆ 정리하면 모든 노드개수에서 루트노드 1개를 제외한 수만큼의 간선이 존재한다.

◆ n-1개 보다 작은 간선을 가진다면 트리가 아니라 포리스트인 경우이다.

> **이진트리의 종류**

→ 트리의 형태는 레벨과 노드 수에 따라 결정된다.

1) 포화이진트리 : 모든레벨의 노드가 꽉 차있는 형태의 이진트리로 노드개수의 일반식은 $n = 2^h-1$이다.

2) 완전이진트리 : 노드가 왼쪽부터 차례로 채워진, 중간에 빈 노드가 없는 트리로 노드개수는 $n < 2^h-1$이다.

3) 편향이진트리 : 같은 높이의 이진트리 중에서 최소개수의 노드개수를 가지면서 왼쪽 혹은 오른쪽 서브트리만을 가지는 이진트리로 노드의 최대개수 범위는 $h \leq n \leq (2^h - 1)$가 된다.

> **이진트리의 추상자료형**

이름		Input	Output	설명
이진트리생성	makeBinTree()	원소e	이진트리bt	원소 e를 데이터로 갖는 루트노드로 구성된 이진트리bt를 생성
루트노드 반환	getRootNode()	이진트리 bt	루트노드rn	이진트리bt의 루트노드 반환
왼쪽 자식노드 추가	insertLeftChildNode()	부모노드 pt 원소e	생성된 노드 n	부모 노드pt의 오른쪽 자식 노드 n추가
오른쪽 자식 노드 추가	insertRightChildNode()	부모노드 pt 원소e	생성된 노드 n	부모노드pt의 왼쪽 자식 노드 n추가
왼쪽 자식 노드 반환	getLeftChildNode()	노드n	왼쪽 자식노드 ln	노드n의 오른쪽 자식노드 반환
오른쪽 자식 노드 반환	getRightChildNode()	노드n	오른쪽 자식 노드 rn	노드n의 왼쪽 자식노드 반환

이진트리루트노드의 데이터	getData()	이진트리 bt	원소e	이진트리bt의 루트노드 데이터를 반환
이진트리 삭제	deleteBinTree()	이진트리 bt		이진트리를 제거(메모리 해제)

> **배열을 이용한 이진트리 구현**

→ 배열을 이용한 트리는 노드의 번호가 인덱스 역할을 하는 점이 중요

> **특징**

→ 장점 : 배열 인덱스를 사용하여 노드접근이 편리

→ 단점 : 편향이진트리와 같이 빈 노드가 많을 경우에 메모리 낭비가 심하다.

- 높이가 h인 트리에 대해서는 미리 최대 저장노드개수보다 한 개 더 많게($2^h - 1 + 1$) 메모리를 할당 해야 한다.

→ 트리의 크기가 증가하는 경우 필요한 메모리 양이 급속히 증가하게 된다.

> **포인터를 이용한 이진트리 구현**

→ 장점 : 필요한 만큼만 메모리를 할당하기 때문에 메모리 효율성 측면에서 우수하다.

→ 단점 : 포인터를 이용해 구현하기 때문에 노드의 탐색과 메모리 관리측면에서 구현이 다소 복잡하다.

> **포인터를 이용한 구현**

- 이진트리 순회

> **순회**

→ 트리의 모든 노드를 한번씩 방문하는 것을 말한다.

→ 트리의 내용을 출력하기 위해서는 트리의 모든 노드를 한번씩 방문하는 순회가 필요하다.

→ 트리를 대상으로 한 다양한 알고리즘은 기본적으로 이러한 순회를 기반으로 한다는 점에서 순회 알고리즘을 정확히 이해하고 구현할 수 있다는 것은 매우 중요하다.

→ 이진트리에서의 순회는 크게 현재 노드방문(V), 왼쪽서브트리 방문(L), 오른쪽 서브트

리 방문(R)으로 구성된다.

→ 트리는 계층구조를 가지기 때문에 순회알고리즘에 따라 여러가지 순회 방법이 가능한대, 대표적으로 전위 중위 후위 순회와 레벨 순회가 있다.

	종류	방문순서
1	전위순회	V-L-R (현재 노드 방문 – 왼쪽 서브트리 이동 – 오른쪽 서브트리 방문)
2	중위순회	L-V-R (왼쪽 서브트리 방문 – 현재 노드 방문 – 오른쪽 서브트리 방문)
3	후위순회	L-R-V (왼쪽 서브트리 방문 – 오른쪽 서브트리 방문 – 현재 노드 방문)

→ 레벨순회는 앞의 3가지 순회가 현재노드방문과 서브트리 방문으로 구성된 것에 비해, 형제노드 방문으로 구성되어 있다.

◆ 같은 레벨에서는 왼쪽에서 오른쪽으로 노드를 방문한다.

> **전위 순회**

→ 노드 방문순서는 VLR로 순회의 목적이 내용출력이라면, 현재 노드 방문은 현재 노드의 내용을 출력하는 것이다. 반면 서브트리 이동은 해당 서브트리의 루트 노드로의 이동을 의미한다.

→ 만약 오른쪽 자식노드만 있다면 다음 이동노드가 오른쪽 자식노드가 된다.

→ 아울러 자식노드가 없는 단말노드의 경우 부모노드로 올라가서 마찬가지로 부모노드의 왼쪽 혹은 오른쪽 노드가 다음 이동노드가 된다.

> **중위순회**

→ 노드 방문순서는 LVR로 왼쪽 자식노드를 방문한 다음 현재노드를 방문하고 오른쪽 서브트리로 이동한다.

> **후위순회**

→ 노드의 방문순서는 LRV로 왼쪽 자식노드를 방문한 다음 바로 오른쪽 서브트리로 이동한다.

> **레벨순회**

→ 레벨의 크기에 따라 낮은 레벨에서 높은 레벨의 노드로 방문하는 순회방법이다.

→ 만약 같은 레벨이면 왼쪽 노드에서 오른쪽 노드로 이동한다는 특성이 있다.

> **순회의 구현**

→ 순회는 크게 재귀호출에 의한 구현과 반복에 의한 구현이 있다.

> **재귀 호출로 구현한 순회**

> **반복으로 구현한 순회**

→ 반복으로 순회를 구현하기 위해서 연결리스트 스택과 연결리스트큐를 사용했다.

→ 구조체 BinTreeNode에 내부변수 Visited를 추가했다.

◆ 이 변수는 반복 후위 순회 함수 구현을 위해 추가 되었다.

→ 전위 중위 후위 순회를 반복으로 구현하려면 스택이 필요하다.

→ 또한 레벨순회 구현에는 큐가 필요하다.

> **반복전위순회**

→ 스택에서 팝된 노드가 현재노드라고 한다면, 현재 노드를 먼저 방문처리하고 난 뒤 현재노드의 오른쪽 자식 노드와 왼쪽 자식 노드를 차례로 푸시한다.

→ 이런 반복적 처리를 스택에 저장된 노드가 없을 때까지 계속한다.

> **반복중위 순회**

→ 스택에 현재 노드부터 왼쪽 서브트리 노드들을 차례로 푸시한다.

→ 팝된 현재노드를 방문 처리한 이후, 현재 노드의 오른쪽 서브트리를 대상으로 다시 앞서 작업을 반복한다.

→ 반복적 처리를 스택에 저장된 노드가 더 없을 때까지 계속한다.

> **반복레벨 순회**

→ 루트노드를 시작노드로 각 노드의 왼쪽 서브트리와 오른쪽 서브트리를 큐에 인큐한다.

→ 이는 큐의 선입선출특성을 이용한 방법이다.

- 이진트리연산

> **앞서 다루지 않은 6개의 함수를 구현했다.**

1) 이진트리 복사 : 현재노드를 먼저 복사한 다음, 왼쪽 서브트리 및 오른쪽 서브트리를 재귀 호출을 통해 복사한다.

2) 이진트리 동일성 검사 : 앞서 소개한 재귀호출을 이용한 전위순회를 이용해 구현할 수 있다. 현재 노드의 자료를 비교한 뒤 왼쪽서브트리와 오른쪽 서브트리를 각각 재귀 호출을 통해 비교한다.

3) 이진트리 노드개수 반환 : 이진트리의 노드개수 반환은 재귀호출을 이용한 후위순회를 이용하면 구현할 수 있다. 왼쪽 서브트리의 노드수와 오른쪽의 수를 더한 후 현재 노드개수를 더하면 구할 수 있다.

4) 이진트리 단말노드개수 반환 : 노드개수 반환 함수와 비슷한 구조이나 단말 노드인 경우에만 노드 개수를 반환하는 점에서 차이가 있다.

5) 이진트리 높이 반환 : 트리의 높이는 루트노드부터 가장 먼거리에 잇는 자식노드의 높이에 1을 더한 값이기 때문에, 왼쪽 서브트리와 오른쪽 서브트리의 높이를 비교해, 더 큰값에 1을 더한 값으로 계산될 수 있다.

6) 이진트리 구조 및 내용 출력 : 전위 순회를 통해 출력할 수 있다.

- 힙(Heap)

> **힙은 루트노드가 언제나 그 트리의 최댓값 혹은 최솟값을 가진다는 특성이 있다.**

> **루트노드가 트리의 최댓값을 가지는 힙을 최대힙이라 하고 최솟값을 가지면 최소힙이라 부른다.**

> **최대힙은 최대트리의 속성을 가진다.**

◆ 이처럼 최소트리 혹은 최대트리의 조건을 만족하는 이진트리라고 해서 반드시 힙인 것은 아니다.

> **힙의 또다른 조건은 힙은 완전이진트리여야 한다는 점이다.**

◆ 힙은 앞서 언급했던 완전이진트리처럼 노드사이에 공백이 없어야 한다.

> **정리 : 힙이란 완전이진트리이면서 동시에 최대/최소트리를 만족하는 트리를 말한다.**

◆ 최대힙과 최소힙은 루트노드를 제외하면 느슨한 정렬상태를 유지한다.

◆ 각 노드별로 부모노드가 자식 노드의 키값보다 최대힙은 크고 같을 경우, 최소힙은 작거나 같을 경우면 된다.

◆ 이는 특정노드의 키값이 하위 레벨의 모든 노드보다 크거나 작다는 의미는 아니다.

> **힙의 추상자료형**

이름		Input	Output	설명
힙 생성	CreateHeap()	힙의 크기n	힙의 크기 d	최대 n개의 원소를 가질 수 있는 공백힙 d생성
힙 삭제	DeleteHeap()	힙h	N/A	힙제거(메모리 해제)
자료추가	InsertHeap()	힙h 원소e	성공/실패여부	힙에 새로운 원소 추가
자료제거	DeleteHeap()	힙h	원소e	힙의 루트노드의 원소를 제거한 뒤 반환

→ 힙은 구현 방법에 따라 배열을 이용하는 경우와 연결리스트를 이용하는 경우로 구분되며 자료 저장방식에 따라 최대힙과 최소힙으로 구분된다.

> **최대힙에서의 삽입연산**

1) 트리의 마지막 자리에 임시저장

■ 힙에 새로운 노드가 추가되면 먼저 트리의 가장 마지막 자리에 임시로 저장한다.

■ 포화 이진트리의 경우 마지막 레벨의 다음레벨에서 제일 왼쪽에 추가한다.

2) 부모노드와 키값비교 및 이동

■ 최대트리 조건을 만족시키기 위해 새로 추가된 노드와 이 노드의 부모노드 사이의 키값을 비교해야 한다.

> **최대힙에서의 삭제연산**

→ 힙에서 노드의삭제는 오직 루트노드만 가능하다.

→ 삭제 연산을 실행하면 힙 내에서 가장 큰 값을 가져가는 루트노드가 반환된다.

1) 루트노드의 삭제

■ 삭제 연산은 루트노드를 대상으로 한다.

2) 트리 마지막 자리 노드의 임시이동

- 트리의 가장 마지막자리에 있는 노드를 임시로 루트노드위치로 이동한다.

3) 자식노드와 키값 비교 및 이동

- 한번의 노드 이동만으로는 최대트리의 조건을 만족시키지 못할 때도 있으므로 최대트리의 조건을 만족할 때까지 부모노드와 자식노드사이의 이동연산을 계속 실행해 줘야한다.

> **힙에 저장되는 자료의 개수를 n으로 놓았을 때 새로운 자료를 추가하는 함수의 시간 복잡도는 $O(log_2 n)$이다.**

> **이런 시간복잡도를 가지는 이유는 힙은 추가되는 자료를 완전이진트리 형태로 저장하기 때문이다.**

- **이로인해 자료의 개수가 n개인 경우에는 트리의 높이가 $log_2 n$이 된다.**
- **따라서 새로운 자료를 추가하거나 삭제할 경우 $O(log_2 n)$번의 접근을 통해 자료의 추가 제거가 가능하다.**

> **배열로 만들어진 힙은 각 노드별로 부여된 배열 인덱스 값에 아래와 같은 규칙이 적용된다.**

1) 노드 i의 부모노드 인덱스 → (int)i/2 (단 i > 1)

2) 노드 i의 왼쪽 자식 노드 인덱스 = 2*i

3) 노드 i의 오른쪽 자식노드 인덱스 = (2*i)+1

> **힙 구현**

- 이진탐색트리

> **이름에 탐색이 들어간 이진트리 답게 자료의 검색이 주된 기능이다.**

> **효율적인 자료검색을 목적으로 기존 이진트리에 몇가지 제약사항을 추가한 이진트리를 말한다.**

> **특정 키 값에 해당하는 노드를 신속하게 찾는 것이 기본적인 기능이다.**

> **특성**

1) 트리의 모든 노드는 유일한 키를 가진다.

2) 왼쪽 서브트리에 있는 모든 노드의 키는 루트의 키보다 작다.

3) 오른쪽 서브트리에 있는 모든 노드의 키는 루트의 키보다 크다.

4) 왼쪽 서브트리와 오른쪽 서브트리도 모두 이진탐색트리이다.

> **추상자료형**

이름		Input	Output	설명
이진탐색트리 생성	createBinSearchTree()		이진탐색트리d	공백 이진탐색트리d를 생성
검색	Search()	이진탐색트리d 키값k		이진탐색트리에서 키값k를 가지는 노드를 반환
데이터 추가	InsertElement()	이진탐색트리d 원소e	성공/실패여부	이진탐색트리에 새로운 원소를 추가
데이터 제거	DeleteElement()	이진탐색트리d 키값k		이진탐색트리에서 키값k를 가지는 노드를 제거
이진탐색트리 삭제	DeleteBinSearchTree()	이진탐색트리d	N/A	이진탐색트리를 제거(메모리 해제)

→ 이진탐색트리는 자료의 검색이 일차적인 목적이기 때문에 자료를 검색하는 search연산이 필요하다.

> **검색 연산**

→ 검색연산은 이진탐색트리의 루트노드를 시작으로 하여, 단계마다 3가지 경우에 따라 연산이 지속된다.

1) 입력 키 값과 현재노드의 키 값이 같은 경우 : 검색종료(성공)

2) 입력 키 값보다 현재노드의 키 값이 큰경우 : 왼쪽 서브트리로 이동

3) 입력 키 값보다 현재노드의 키 값이 작은경우 : 오른쪽 서브트리로 이동

> **추가 연산**

→ 새로운 노드를 추가하기 위해서는 추가하려는 노드의 키 값이 삽입되기 적절한 위치를 먼저 찾아야 한다.

◆ 추가 하려는 노드의 키 값을 입력값으로 해서 먼저 검색연산을 실시한 다음 같은 키 값을 가진 노드를 찾았다면 추가연산이 실패하도록, 찾지 못했다면 검색연산의 최종위치에 새로운 노드가 삽입 될 수 있도록 한다.

> **삭제 연산**

→ 삭제할 노드의 자식 노드 수에 따라 3가지 경우를 고려해야 한다.

1) 삭제하려는 노드의 자식노드가 없는 경우(단말노드인 경우)

- 대상노드를 메모리 해제 하고 부모노드의 연결에 NULL을 대입한다.

2) 삭제하려는 노드의 자식노드가 1개인 경우

- 해당노드를 삭제하고, 삭제노드의 자식노드를 삭제노드의 위치에 이동시켜 주면 된다.

3) 삭제하려는 노드의 자식노드가 2개인 경우

- 해당 노드 제거 후 대체되는 노드는 이진탐색트리의 특성을 해치지 않게 왼쪽 서브트리의 모든 노드키값보다는 큰, 오른쪽 서브트리의 모든 키값보다는 작은 노드를 선택해 대체한다.
  - 이런 조건의 노드는 아래의 경우와 같이서브트리 노드 키 값의 중간이 되는 노드여야 한다.
    - ◆ 왼쪽 서브트리의 가장 큰 키값
    - ◆ 오른쪽 서브트리의 가장 작은 키값.

> **이진탐색트리 구현**

그래프

- 그래프 개념

> **그래프는 노드(Node혹은 정점Vertex)와 간선(Edge)의 집합이다.**

> **그래프는 객체사이의 연결 관계를 표현하는 자료구조이다.**

> **지금껏 나온 자료구조 중에서 표현 능력이 가장 우수하여 현실세계의 다양한 문제를 효과적으로 모델링하는 목적으로 사용되고 있다.**

> **그래프의 정의는 노드V와 간선E로 구성된 그래프 G에 대한 식 $\mathbf{G} = \big(\mathbf{V}(\mathbf{G}), \mathbf{E}(\mathbf{G})\big) = (\boldsymbol{V}, \boldsymbol{E})$ 으로 나타낼 수 있다.**

◆ **해당 식은 "그래프G는 노드들의 집합V(G)와 간선들의 집합 E(G)로 구성된다."라는 의미이다.**

> **그래프의 종류**

구분	종류	설명
간선의 방향성	무방향 그래프	간선에 방향이 없는 그래프
	방향 그래프	간선에 방향이 있는 그래프
간선의 가중치	가중 그래프	간선에 가중치가 할당된 그래프
구조적 특징	완전 그래프	연결 가능한 최대 간선 수를 가진 그래프
	부분 그래프	원래의 그래프에서 일부의 노드나 간선을 제외하여 만든 그래프
	다중 그래프	중복된 간선을 포함하는 그래프

◆ 무방향 그래프 : 노드르 ㄹ연결하는 간선에 방향이 없는 그래프로 A, B노드를 잇는 간선(A,B)는 간선(B, A)와 같은 간선이다.

- 식 : $\begin{cases} \text{그래프 G의 노드} - \mathrm{V(G)} = \{A, B\} \\ \text{그래프 G의 간선} - \mathrm{E(G)} = \{(A, B)\} \end{cases}$

◆ 방향 그래프 : 노드를 연결하는 간선에 방향이 있는 그래프이며, 다이그래프(Digraph)로 불린다.

- 간선<A, B> 는 간선<B, A>와 다른 간선이다.
- 식 : $\begin{cases} \text{그래프 G의 노드} - \mathrm{V(G)} = \{A, B\} \\ \text{그래프 G의 간선} - \mathrm{E(G)} = \{< A, B >, < B, A >\} \end{cases}$
- 방향 그래프는 모델링 하려는 시스템이 구성 객체들 사이에 연결 관계가 대칭적이지 않을 때 사용된다.
- 덧붙여, 방향 그래프의 간선<A, B>에서 시작노드 A는 꼬리(Tail)라 하고 종료노드B는 머리(Head)라 한다.
  - 즉 방향그래프의간선은 꼬리에서 머리로 그려진 화살표이다.

◆ 가중그래프 : 노드를 연결하는 간선에 가중치가 할당된 그래프로, 가중치는 간선 사이의 비용(Cost)혹은 거리(Distance)등 다양한 속성으로 사용될 수 있다.

◆ 무방향 그래프에 가중치가 존재한다면 해당 그래프는 무방향 가중그래프라 하고 방향그래프에 가중치가 있으면 이는 방향 가중그래프라 한다.

- 일반적으로 방향 가중그래프르 네트워크라고도 부른다.
- 노드사이의 연결관계 이외에 비용 혹은 거리 등의 추가 속성을 정의 할 수 있어서 응용분야가 포괄적인 그래프 모델이다.

◆ 완전 그래프 : 그래프 내의 모든 노드가 1:1 간선을 연결된 경우로, 연결 가능한 최대 간선 수를 가진 그래프를 말한다.

- 노드의 개수를 n, 간선의 개수를 m이라 할 때 무방향그래프와 방향그래프는 다음과 같은 식으로 정의 할 수 있다.
- 식 : $\begin{cases} \text{무방향 그래프} & \mathrm{m} = \frac{n(n-1)}{2} \\ \text{방향 그래프} & \mathrm{m} = \mathrm{n}(\mathrm{n}-1) \end{cases}$

◆ 부분 그래프 : 일부의 노드나 간선을 제외하여 만든 그래프를 부분 그래프이다.

- 그래프G의 부분 그래프G′는 G의 노드 V(G)와 간선의 집합E(G)의 부분집합으로 만들어졌으며 이를 식으로 표현하면 다음과 같다.
- 식 : $\begin{cases} G' = (V(G'), E(G')) \\ V(G') \subseteq V(G) \\ E(G') \subseteq E(G) \end{cases}$

◆ 다중 그래프 : 중복된 간선을 포함하는 그래프를 말한다.

> **그래프 관련 용어**

→ 인접 : 두개의 노드를 연결하는 간선이 존재하는 경우 인접(Adjacent)되었다 표현한다.

→ 부속 : 두 노드를 연결하는 간선이 존재하는 경우 해당 간선은 두 노드에 각각 부속(Incident)되었다고 한다.

→ 차수 : 노드에 부속된 간선의 개수를 차수(Degree)라 한다.

- 단 방향그래프의 경우 노드로 들어오는 간선의 개수를 진입차수(In-Degree)라 하고, 나가는 간선의 개수를 (Out-Degree)라 한다.

- ◆ 경로 : 노드 A에서 B까지의 경로는 A에서 B에 이르는 간선들의 인접(Adjacent)노드를 순서대로 나열한 리스트를 말한다.
- ◆ 경로를 구성하는 간선의 개수를 경로 길이(Path Length)라 하며 특히 경로 중에 같은 노드가 존재하지 않는 경우를 단순경로(Simple Path)라 한다.
- ◆ 보통 방향그래프에서의 단순 경로는 단순 방향경로(Simple Direted Path)라 한다.
- ◆ 아울러 단순경로 중에서 경로의 시작노드와 마지막 노드가 같은 경로를 사이클(Cycle)이라 한다.
- ◆ 그래프 내의 모든 노드사이에 경로가 있을 때, 그래프가 연결(Connected) 되었다고 한다.

> **그래프의 동일성**

→ 같은 그래프라 할지라도 시각적 표현은 다를 수 있다.

- ◆ 노드와 간선의 집합이 같으면 같은 그래프이다.

> **루프**

→ 임의의 노드에서 자기 자신으로 이어지는 간선을 루프라고 한다.

- 추상자료형

이름		Input	Output	설명
그래프생성	createGraph()	최대노드개수 n	그래프g	최대n개의 노드를 가지는 공백 그래프 생성
그래프 삭제	deleteGraph()	그래프 g		그래프 g의 모든 노드 및 간선제거
노드추가	addVertex()	그래프 g 노드v		그래프g에 노드 추가
간선추가	addEdge()	그래프g 노드u 노드v		그래프g에 해당 노드들을 잇는 간선추가
노드제거	removeVertex()	그래프g 노드v		그래프g의 노드 및 해당 노드에

				연결된 간선제거
간선제거	removeEdge()	그래프g 노드u 노드v		그래프g의 간선을 제거
인접노드반환	adjacentNode()	그래프g 노드v	노드의 목록	그래프 g의 노드v에 인접한 모든 노드를 반환

- 그래프 구현

> **일반적으로 인접행렬을 이용하는 방법과 인접리스트로 구현하는 2가지 방법이 사용된다.**

> **인접행렬 : 데이터를 2차원배열로 저장**

→ 각 노드 사이의 간선을 2차원 배열에 저장하는 방법

→ 간선이 있을 때는 값이 1 없을 때는 0

→ 각 행은 나가는 간선 열은 들어오는 간선을 의미한다

→ 무방향그래프는 대칭인 배열의 값이 같은 대 이러한 성질을 대칭성이라 한다.

→ 각 노드의 차수는 인접행렬의 각 행의 합 또는 열의 합으로 구할 수 있다.

→ 구현

> **인접리스트 : 연결리스트로 저장**

→ 정점 별 인접정점을 연결리스트로 저장하는 방식

→ 인접행렬이 노드 사이의 연결 여부와 상관없이 모든 간선의 정보를 2차원배열에 저장한 반면, 인접리스트는 실제 연결된 노드만을 저장한다.

→ 간선에 대한 접근이 연결리스트의 순회가 필요하다.

→ 구현

- 그래프 탐색

> **그래프의 탐색은 긴선을 이용하여 그래프 상의 모든 노드를 한번 씩 방문하는 것을 말한다.**

> **대표적으로 깊이 우선탐색(DFS)와 너비우선탐색(BFS)이 있다.**

> **깊이 우선 탐색**

→ 깊이 우선은 현재 선택된 노드와 연결된 노드를 먼저 선택하는 것

→ 스택을 통해 구현

1) 선택된 노드인 0과 연결된 노드들을 찾고, 이들 중에서 아직 탐색되지 않은 노드들이 있는지 점검한 후 차례대로 푸시한 뒤, 다음에 이동할 노드를 팝을 해 반환한다.

2) 선택된 노드와 연결된 노드들을 각각 스택에 푸시시키고 다음으로 이동할 노드를 팝해 선택한다.

3) 만약 이미 스택에 추가된 인접노드는 넘어간다.

> **넓이 우선 탐색**

→ 현재 선택된 노드의 이전 노드에서 연결된 다른 노드를 먼저 탐색 하는 것

→ 큐를 통해 구현

1) 시작노드와 연결된 노드들을 인큐 한 후 다음 디큐로 반환

2) 큐의 선입선출로 자연스럽게 넓이 우선 탐색이 가능해 졌음을 파악 할 수 있다.

3) 인큐시킬 데이터는 방문한 적 없고 이미 큐에 삽입된 것이 아니어야 한다.

- 신장트리와 최소비용신장트리

> **신장트리는 모든 노드를 포함하는 트리이다.**

> **트리의 일종이기 때문에 순환이 없어야 한다는 제약사항이 있다.**

> **기존 그래프가 가진 모든 노드를 순환 없이 서로 연결시킨 트리이다.**

> **그래프G의 간선 중에 신장트리에도 포함된 간선을 트리간선이라 하고, 신장트리에 포함되지 않는 간선을 비 트리 간선이라고 한다.**

> **신장트리는 앞서 설명한 그래프 탐색을 이용하여 생성 할 수 있고, 앞에서 설명한 탐색 방법을 활용해 깊이 우선 신장트리, 넓이 우선 신장트리를 구할 수 있다.**

> **최소비용 신장트리는 가중 그래프의 신장트리중에서 가중치의 합이 최소인 신장트리를 말하낟.**

> **최소 신장트리를 구하는 방법은 대표적으로 Prim알고리즘과 Kruskal알고리즘이 있다.**

> **크루스칼 알고리즘 : 모든 간선을 비용순으로 정렬한 다음 낮은 비용을 가지는 간선을 차례대로 선택해서 신장트리를 완성해가는 방법.**

> **알고리즘**

1) 간선을 가중치 값에 따라 오름차순으로 정렬한다.
2) 간선 중 가중치가 가장 작은 간선을 추출
3) 신장트리에 연결된 노드개수와 그래프에 연결된 노드개수를 비교한 뒤 신장트리가 작으면 다시추출
4) 어떤 간선을 선택했을 때, 순환이 발생한다면 다음으로 작은 가중치의 간선을 선택

> **프림알고리즘 : 임의의 시작노드1개만을 추가해서 현재 신장트리와 연결된 간선 중에서 가장 적은 비용을 가지는 간선을 선택한다.**

> **알고리즘**

1) 시작노드 선택
2) 해당 노드에 부속된 간선들은 모두 2개이며 이중 가장 가중치가 작은 간선이 선택되어 해당 간선의 진출노드가 새로운 노드로 추가
3) 추가된 노드들에 부속된 간선들 중 가중치가 가장 작은 노드를 선택하되, 사이클이 생성되지 않는 간선을 선택한다.
4) 신장트리의 노드수와 그래프의 노드수가 같을 때까지 반복

- 최단경로

> **두 노드 사이의 경로들 중에서 최소의 비용으로 이동할 수 있는 경로를 찾는 문제**

> **세분화 하면 크게 세가지 종류로 난ㄹ 수 있다.**

1) 단일 시작점에서 최단경로 구하기

→ 첫 번째 문제는 특정 노드에서 다른 모든 노드 사이의 최단 경로를 구하는 문제로 다른 최단경로 문제의 가장 기본이 되는 문제이다.

→ 간선의 가중치가 양수인 경우와 음수인 경우로 나눌 수 있다.

2) 모든 최단경로 구하기

→ 그래프 상의 모든 노드에서 다른 모든 노드사이의 최단경로를 구하는 문제로, 첫번째 문제의 확장된 경태이다.

3) 도달 가능성 구하기

→ 방향그래프인 경우에 특정 노드에서 다른 노드 사이에 경로가 존재하는지를 판단하는 문제이다.

> **단일 시작점에서의 최단경로 – Dijkstra**

→ 노드 사이의 거리가 음수인 경우에는 구할 수 없다.

→ 시작노드부터 노드V까지는 최단 거리를 알고 있다고 가정한 뒤 시작노드부터 노드W까지의 최단거리를 알고 싶을 때 사용한다.

→ 다익스트라 알고리즘의 단계

1) 시작노드에서 최단거리를 알고 있는 노드 V선택
2) 최단거리 노드 V와 연결된노드 W에 대해 다음 조건을 검사
3) 조건 : 만약 노드W의 기존거리보다 노드V에 간선(v,w)를 연결한 새로운 경로가 더 짧다면, 노드 w의 경로를 수정한다.

→ 다익스트라 알고리즘은 단일시작점이 주어진 경우 최대n번수행한다.

> **알고리즘**

◆ **초기화**

- 시작노드와 연결된 노드들에 대해서만 거리를 설정하고, 남은 노드들은 도달하지 못함(INF)을 표시한다.
- 집합S는 시작노드를 제외한 그래프의 모든 노드를 포함한다.

◆ **루프**

- 집합S 중 최단 거리를 가지는 노드를 선택 후 해당 노드를 S에서 제거한다.
- 또한 앞서 제거한 노드에 인접한 노드에 대해 앞서 나온 조건을 검사한다.
- S에 포함되지 않은 노드는 검사대상에서 제외된다.

- S집합에 포함된 노드들의 거리정보를 갱신한다.
- S가 공백상태가 될 때까지 반복한다.

> **모든 최단경로 구하기 : Floyd 알고리즘**

→ 그래프 상의 모든 노드에서 다른 모든 노드 사이의 최단 경로를 구하는 알고리즘

→ 기본적으로 다익스트라 알고리즘과 동일한 접근 방법을 사용하고 있다.

◆ 최단경로를 수정하는 전략이 동일 한대, 시작노드에서 임의의 노드까지의 최단 거리를 구하는 방법으로 기존의 최단경로를 알고있는 노드를 이용하는 방법이다.

◆ 대신 시작노드가 그래프의 특정노드가 아니라 그래프의 모든 노드 집합에 대해서 수행한다.

→ 플루이드 알고리즘은 시작노드와 도착노드로 구성된 2차원 배열의 모든 원소에 대해 루프를 돌면서 최단경로를 구한다.

→ 이때 그래프의 모든 노드를 중간노드로 놓고 최단경로를 변경한다.

→ 또한 그래프의 시작노드와 도착노드 사이의 거리를 저장하는 2차원배열을 A라고 가정했을 때, 행렬의 각 원소[시작노드][종료노드]는 해당 경로의 길이를 저장하게 된다.

→ 초깃값으로는 해당 간선이 존재한다면 간선의 가중치가 되고, 서로 인접하지 않은 노드의 경우라면 무한대가 저장된다.

> **알고리즘**

◆ **초기화**

- 서로 인접한 두 노드에 대해서 해당 간선의 가중치로 조기화되며, 직접적으로 연결되지 않는 노드 사이의 거리는 무한대가 된다.
- 직접적으로 연결된 노드들에 대해서만 설정된다.

◆ **루프**

- **거리가 단축되는 경우**
  - 값을 대입하고 다음 노드 확인
- **단축 되지 않는 경우**

- 다음노드 확인

> **도달 가능성 구하기**

→ 그래프 상의 모든노드에서 다른 모든 노드 사이로 이동할 수 있는 지를 구하는 문제

→ 방향그래프를 대상으로 하며, 플로이드 알고리즘과 마찬가지로 시작노드와 도착노드로 구성된 2차원 배열의 모든 원소에 대해 루프를 돌면서 이동할 수 있을지를 확인한다.

→ 단순히 경로가 존재하는지만 특정값으로 설정한다는 점이 플로이드 알고리즘과 다른 점이다.

→ 핵심은 도달 가능성을 점검하는 부분에 있다.

→ 시작노드에서 중간노드까지 경로가 있는지를 점검한 후, 시작노드에서 중간노드까지 경로 가 있다면, 중간노드부터 도착노드까지 경로가 있는지 파악한 후, 만약 경로가 존재한다면 이는 시작노드부터 도착노드까지 경로가 존재한다는 뜻으로 도달 가능하다는 표시를 남긴다.

→ 즉 도달 가능성 문제는 플로이드 알고리즘을 적절히 응용한 형태이다.

정렬

- 정렬

> **정렬이란?**

→ 순서없이 배열되 있는 자료들을 그 값 및 순서에 따라 재배열 하는 것

> **키 값**

→ 자료를 정렬하는 데 사용하는 자료의 값

→ 오름차순 : 작은 값부터 시작해 값이 증가하는 순서대로 배열하는 것

→ 내림차순 : 오름차순 반대

> **효율성**

→ 얼마만큼 빠르게 정렬을 실시하는지를 나타내는 것

→ 대상이 되는 자료가 n개일 때 필요한 비교 연산과 자료 이동연산의 횟수에 의해 결

정

→ 연산의 횟수가 같다 하더라도 알고리즘에 따라 비교 연산이 많이 수행되는 경우와 이동 연산이 많이 되는 경우로 나눌 수 있다.

→ 저장된 자료의 상태에 따라 평균 효율성과 최악인 경우의 효율성이 차이가 있다면 이 또한 알고리즘의 선택 시 고려해야 할 관점 중 하나이다.

> **안정성**

→ 같은 키 값을 가지는 자료들을 입력 순서 그대로 정렬하는지를 확인

→ 결과가 원본의 정렬순서와 일치된다면 이는 안정성이 있는 안정 정렬이다.

→ 일치 하지 않는다면 이는 불안정 정렬이다.

→ 안정성 여부는 정렬에 사용되는 키가 2개 이상인 경우에 의미가 존재

→ 정렬에 사용되는 키가 여러 개라면 이는 다중키 정렬이라 부른다.

→ 안정성 여부는 다중 키 정렬에서 중요한 차이를 발생시키는데, 안정정렬이라면 기존 정렬 내용이 유지되기 때문에 단순히 우선순위가 낮은 키부터 높은 순으로 여러 번 정렬만 한다면 자동으로 다중키 정렬이 가능 할 것인 반면, 불안정 정렬의 경우 다중 키 정렬의 키값 비교를 위한 별도의 비교 로직이 추가되기에 비효율 적일 것이다.

→ 일반적으로 효율성이 높은 정렬알고리즘은 불안정 정렬인 경우가 많다.

- 정렬의 종류

> **정렬이 수행되는 방법에 따라 두가지로 구분 가능**

1) 정렬 수행 장소

→ 메모리 내부에서 정렬될 경우 내부정렬

◆ 자료가 컴퓨터 메모리에 로딩되기 때문에 빠르게 정렬 할 수 있지만, 대용량데이터는 정렬할 수 없다.

→ 외부보조기억장치에서 정렬하는 경우 외부정렬

◆ 외부보조기억장치를 활용하기 때문에 대용량 데이터를 정렬할 수 있지만, 수행속도가 느리다.

2) 정렬 실행 방법

→ 크게 다섯가지로 분류가능

1) 교환 : 키 값을 비교 후 자료 교환
    - 버블 정렬이나 퀵 정렬이 이에 해당
2) 삽입 : 키 값 비교 후 자료 삽입
    - 삽입정렬과 셸정렬이 여기 해당
3) 병합 : 키 값 비교 후 자료 병합
    - 2 – way 병합과 n - way병합이 이에 해당
4) 분배 : 키 값을 여러 개의 부분집합으로 분배 한 후, 이를 이용해 자료를 정렬
    - 기수정렬이 여기에 해당
5) 선택 : 트리 자료구조 등 특정 자료구조를 통해 정렬하는 방식
    - 힙 정렬이 여기에 해당

- 선택정렬

> **정렬되지 않은 전체 자료 중 해당 위치에 맞는 자료를 선택하여 위치를 교환 하는 정렬 방식**

> **알고리즘**

1) 최솟값 탐색
2) 가장 처음 인덱스의 값과 교환
3) 교환된 다음번지의 인덱스부터 최소값 탐색
4) 다음 번지의 값과 교환
5) 반복

> **특성**

→ 비교연산의 시간복잡도 : O(n(n-1) / 2) = O(n^2)

→ 이동 연산의 시간복잡도 : O(3(n-1)) = O(n)

→ 전체 : O(n^2 + n) = O(n^2)

→ 자료의 이동연산횟수가 O(n)이기 때문에 자료의 크기가 큰 정렬에 유리

→ 또한 비교 및 이동연산의 횟수가 미리 정해져서 최악의 경우와 평균인 경우의 시간 복잡도에 차이가 없다.

→ 하지만 전체효율성이 낮고, 정렬의 안정성이 불안정하다.

> **구현**

- 버블정렬

> **정렬되지 않은 전체 자료들을 대상으로 인접한 두개 자료의 키 값을 비교하여 위치를 교환하는 정렬방식**

> **알고리즘**

1) 가장 처음 인덱스와 다음 인덱스를 비교
2) 기준에 따라 부합될 경우 교환
3) 자료의 끝까지 반복
4) 가장 마지막으로 확인한 인덱스를 제외시키며 전체 반복

> **특성**

→ 비교연산의 시간 복잡도 : O(n(n-1)/2) = O(n^2)

→ 이동 연산의 시간 복잡도 : O(0) ~ O(n(n-1)) + O(n^2)

- 버블정렬은 정렬 전 상태에 따라 이동연산의 차이가 있다.

→ 전체 : O(n^2)

→ 전체 효율성을 볼 때 느린 알고리즘이다.

→ 또한 이동 연산의 횟수가 크기 때문에 자료의 크기가 클 때는 사용하기 어렵다는 단점이 존재한다.

→ 하지만 정렬의 안정성은 유지되기 때문에 같은 키 값을 가지는 자료형의 위치는 변경되지 않는다는 장점이 존재한다.

> **구현**

- 퀵정렬

> **중심값을 기준으로 두 자료의 키 값을 비교하여 위치를 교환하는 정렬방식**

> **피봇기준의 위치 교환이 끝난 다음, 기존 자료 집합을 피봇을 기준으로 2개의 부분집합으로 나눈 뒤, 분할된 부분 집합에 대해 다시 퀵 정렬을 실행하는 "분할정복 기법"에 바탕을 두고 있다.**

> **알고리즘**

1) 첫 인덱스 또는 마지막 인덱스를 피봇기준으로 설정
2) 조건에 따라 참이면 피봇과 가까운 인덱스를 이동 거짓일 경우 먼 인덱스를 이동
3) 두 인덱스가 만나면 피봇과 교환
4) 피봇을 기준으로 두 집합으로 나눈 뒤 재귀호출을 통한 분할정복
5) 분할 할 수 없는 집합의 수 까지 반복

> **특징**

→ 피봇을 기준으로 두 개의 부분집합으로 나누어 자료의 위치를 교환하기 때문에 n개의 자료를 평균O(nlogn)번 만에 정렬하는 효율성을 가진다.

→ 정렬 횟수가 n/2, n/2^2, n/2^3과 같이 평균 logn의 횟수가 되며, 각 정렬 마다 모두 n번의 비교가 필요하기 때문에 평균 비교 횟수는 nlogn이 된다.

→ 또한 이동연산은 비교 연산보다 상대적으로 적게 발생하지만, 피봇을 기준으로 나누어지는 두 개의 부분집합에 지속적인 불균형이 발생한다면 최악의 경우 O(n^2)의 효율성을 가진다.

→ 퀵정렬은 정렬 전 자료의 상태에 따라 효율성에 차이가 있기는 하지만, 전체적인 효율성은 우수한 편에 속한다.

→ 단 최악의 경우 효율성이 떨어진 다는 단점이 존재해, 이를 보완하기 위해 중간값을 피봇으로 선택해서 퀵연산을 수행하는 방법이 사용된다.

→ 또한 퀵정렬은 정렬의 안정성이 유지되지 않는다.

> **구현**

- 삽입정렬

> **기존에 정렬된 부분집합에 정렬할 자료의 위치를 찾아 삽입하는 방식이다.**

> **알고리즘**

1) 비교 대상 인덱스의 값을 임시변수에 저장한 후 해당 인덱스의 이전 인덱스와 비교
2) 반복하며 비교한 인덱스의 값을 오른쪽으로 이동
3) 반복 탈출조건에 부합할 경우 탈출 후 임시 변수에 저장한 값을 대상인덱스에 저장
4) 마지막 인덱스까지 반복

> **특성**

→ 정렬 전 자료의 상태에 따라 효율성의 차이가 무척 큰 알고리즘이다.

→ 기존에 정렬 되어 있는 경우 각 정렬단계에서 1번씩의 비교만 필요하고 이동연산은 필요 없으므로, O(n)의 효율성을 가진다.

→ 반면 모든 원소가 역순으로 되어있는 경우 비교연산과 이동연산의 횟수가 O(n^2)의 효율성을 가지는 특성이 있다.

→ 따라서 효율성은 O(n) ~ O(n^2)을 가진다.

→ 전체 효율성은 그렇게 빠르진 않아도, 알고리즘 자체가 간단하며, 기존자료가 어느정도 정렬된 경우에는 효율적일 수 있다는 장점이 있다.

→ 또한 정렬의 안정성도 유지된다.

> **구현**

- 셸정렬

> **셸정렬은 기존자료를 일정한 간격(interval or gap)에 의해 여러 개의 부분집합으로 나눈 다음, 각 부분 집합에 대해 삽입 정렬을 수행하여 전체 자료를 정렬하는 방식이다.**

> **기존 삽입 정렬이 어느정도 정렬된 집합에서는 상당히 빠른 정렬 알고리즘이라는 특징에서 착안해 고안된 알고리즘 이다.**

> **알고리즘**

1) 초기 간격을 전체 자료개수의 1/2로 설정한 후 조건에 따라 삽입정렬 시행

2) 삽입 정렬 시에는 처음+간격부터 마지막 까지 간격만큼 증가하며 탐색한다.

3) 간격이 1보다 작거나 같아질 때 까지 반복

> **특성**

→ 기존 자료를 일정한 간격으로 여러 부분집합으로 나눈 다음 각 부분집합에 대해 삽입정렬을 수행하는 정렬방식이다.

→ 기존의 삽입정렬에 비해 우수한 효율성을 가진다.

→ 단계별로 미리 정렬을 수행하기 때문에 최종적으로 아주 빠른 삽입정렬이 가능하도록 했기 때문이며, 최선의 경우에는 O(n)의 시간복잡도를 가지지만, 평균적으로 O(n^1.25)의 효율성을 가지며, 최악의 경우 O(n^2)의 효율성을 가지는 특성이 있다.

→ 셸정렬은 삽입정렬과 달리 정렬의 안정성이 유지되지 않는다.

> **구현**

- 병합정렬

> **같은 개수의 원소를 가지는 부분 집합으로 기존 자료를 분할하고 분할된 각 부분집합을 병합 하면서 정렬작업을 완성하는 분할정복기법에 의한 정렬방식이다.**

> **2개의 정렬된 자료집합을 병합하는 경우 2-way병합이라 하고, n개의 자료집합을 병합하는 경우 n-way집합이라 한다.**

> **알고리즘**

1) 분할단계 : 재귀 호출을 활용해 더 이상 나눌 수 없을 때까지 부분집합을 나눈다.

2) 병합단계 : 나눠지 부분집합들을 병합하며 조건에 따라 먼저 저장할 자료를 선택해 정렬하며 병합

3) 분할되었던 모든 집합들을 병합하면 종료

> **특징**

→ 이동 및 비교연산이 평균적으로O(nlogn)번 필요한 비교적 우수한 효율성을 가진다.

→ 따라서 정렬전 자료의 상태에 영향이 적을 뿐만 아니라, 효율이 매우 우수한 정렬방식이다.

→ 단, 추가 메모리 공간이 필요하며, 자료의 이동 횟수가 많다는 단점이 존재한다.

→ 따라서 정렬대상의 자료의 크기가 큰 경우에는 시간적 낭비가 심해지므로 이 경우에는 배열보단 연결리스트를 사용해 실제자료의 이동 대신 연결 포인터만을 변경하는 방식을 통해 물리적 이동 양을 줄여서 성능을 향상시키는 최적화 기법이 필요하다.

→ 또한 이 정렬은 안정성이 유지되는 방식이다.

> **구현**

- 기수정렬

> **기수(Radix)는 숫자의 자리수를 말한다.**

> **키 값의 자릿수에 따라 자료를 분배하는 방식을 통해 정렬하는 알고리즘이다.**

> **즉 앞서 나온 알고리즘들이 각 자료의 키값을 비교하여 이동을 수행하는 방식이었던 것에 비해 기수정렬은 키값끼리의 비교연산이 필요 없으며, 버킷이라 불리는 자료 보관 큐에 자료를 분배하고 다시 이를 꺼내는 연산을 통해 정렬이 이루어진다.**

> **알고리즘**

1) 일정한 크기의 버킷배열을 생성한다.
2) 키값을 버킷배열의 수만큼 나머지 연산한 후 도출되는 기수를 인덱스 삼아 해당 버킷에 삽입한다.
3) 이후 버킷 배열의 전체를 순차적으로 출력해 일차적으로 정렬을 시킨 후 다음자릿수의 기수를 구해 반복한다.

> **특징**

→ 자릿수가 d인 자료의 정렬의 경우 효율성은 O(d*n)이 되며, 이때 자릿수는 보통 32비트 기준 10정도의 상수에 해당하기에 실질적인 효율성은 O(n)으로 볼 수 있다.

→ 즉, 기존의 정렬알고리즘과 비교해 상당히 우수한 효율성을 가진다.

→ 다만, 기수정렬을 위해 자릿수의 개념이 적용될 수 있어야 하기에, 보통 숫자키의 경우에만 적용 될 수 있으며, 한글과 같은 문자키의 경우 상당히 많은 버킷이 필요하다.

→ 또한 자료의 분배 저장을 위해 버킷의 메모리가 필요하다는 특성이 있다.

→ 기수정렬은 안정성이 유지되는 정렬이다.

> **구현**

- **힙 정렬**

  > **힙 자료구조를 이용한 정렬로 힙은 완전이진트리이면서 동시에 최대트리 혹은 최소트리를 말한다.**

  > **힙의 루트는 항상 최대 혹은 최솟값을 가지며, 이를 이용해 정렬을 하는 알고리즘이다.**

  > **알고리즘**

    1) 힙을 통해 전체 자료들을 저장

    2) 힙의 원소들을 제거하는 것으로 정렬 완료

  > **특징**

  → 힙 정렬은 정렬하고자 하는 자료를 힙에 삽입하고, 삭제하는 것으로 정렬한다.

  → 힙을 유지하는데 있어서 힙 재구성이 필수적이기 때문에 평균 O(logn)의 시간이 소요된다.

  → 때문에 전체 평균 효율성은 O(nlogn)으로 다른 방식에 비해 상당히 우수하다.

  → 하지만 힙 생성에 필요한 추가적인 메모리가 필요하고 정렬의 안정성이 유지되지 못하는 단점이 존재한다.

검색

- **검색**

  > **정보검색의 가장 기본이 되는 분야로, 기존에 저장된 자료 중에서 원하는 자료를 찾는 것을 말한다.**

  > **이때 찾고자 하는 자료를 같이 저장되어 있는 자료들과 구별 시켜주는 키를 검색 키라 하며 검색은 이러한 검색키를 가지는 자료를 찾는 것을 말한다.**

  > **정렬과 마찬가지로 검색이 수행되는 위치에 따라 내부검색과 외부검색으로 나눌 수 있다.**

  → 내부검색 : 검색연산이 모두 메모리 내에서 수행되는 것

  → 외부검색 : 디스크 등의 보조 기억장치를 이용해 자료를 수행하는 것

> **검색 방법에 따라 나누기도 한다.**

→ 비교 검색 방법 : 검색키를 비교하여 검색하는 방법

- 대표적으로 순차, 이진, 트리검색등으로 나누어진다.

→ 계산 검색 방법 : 검색키를 계수적으로 계산하여 검색하는 방법

- 해싱이 가장 대표적인 계산방법이다.

- 순차검색

> **일렬로 되어있는 자료들의 검색 키값을 차례대로 비교하는 검색방법이다.**

> **일렬로 되어있는 자료를 검색하기 때문에 선형 검색이라고도 한다.**

> **순차 검색은 가장 단순하면서도 직관적인 검색방법으로 구현이 쉽다는 장점이 있지만, 자료의 양이 증가하면 효율성이 상당히 감소하기 때문에 자료가 적은 경우에만 사용할 수 있다는 단점이 있다.**

> **자료가 정렬 되지 않은 경우의 순차검색**

→ 전체자료를 전부 비교하기 전까지 검색실패여부를 알 수 없기 때문에 전체를 비교해야 하며, n개의 자료를 가지는 경우 평균적으로 O(n+1/2)만큼의 비교연산을 한다.

> **자료가 정렬된 경우의 순차검색**

→ 정렬되지 않은 경우와의 차이점은 검색 실패여부를 자료의 마지막 원소까지 찾지 않아도 된다는 점이다.

→ 검색 실패시 평균 비교횟수가 반으로 감소하지만 평균적인 시간복잡도는 여전히 O(n)이므로 자료의 개수가 증가함에따라 검색에 소요되는 시간또한 비례해 증가함을 알 수 있다.

> **순차검색구현**

> **색인 순차 검색**

→ 자료가 미리 정렬된 경우에 인덱스 테이블을 이용해 검색의 효율성을 높이는 방법이다.

→ 인덱스 테이블은 정렬된 자료를 일정 간격으로 저장하기 때문에, 찾아야 하는 전체 자료의 개수를 감소시키는 역할을 한다.

→ 검색 대상이 되는 전체 자료의 개수가 n개이고, 인덱스 테이블의 개수가 m개라면, 키 값을 비교해야 하는 자료의 개수는

- 인덱스 테이블에서 비교해야 하는 평균 m개와
- 실제 자료집합에서 비교해야 하는 평균 n/m의 합이므로

◆ 평균 O(m+n/m)의 시간복잡도를 가진다.

→ 시간복잡도에서 알 수 있듯 색인검색의 성능향상은 최적의 n과 m의 결정에 있다.

→ 또한 인덱스 테이블을 사용하기 위해서는 자료의 정렬이 필요한데, 만약 중간에 새로운 자료가 삽입되거나 기존 자료의 삭제가 발생한 경우, 기존의 생성된 테이블을 변경해야 하는 문제 또한 발생한다.

> **색인 순차 검색 구현**

- 이진 검색

> **미리 정렬된 자료를 대상으로 검색범위를 반으로 감소시키는 과정을 반복해 검색키를 탐색**

> **가운데 위치한 자료의 키값을 비교해 자료가 배열의 왼쪽에 있는지 오른쪽에 있는지를 판단한다.**

> **구현**

- 해싱

> **검색키에 대한 산술연산을 이용한 검색방식**

> **검색키 값을 입력값으로 계산을 실시하면 검색하려는 자료의 위치를 알 수 있다는 것이 해싱의 기본개념이다.**

> **앞서 순차검색이나 이진검색은 검색 키 값과 실제 자료들의 키값을 하나하나 비교하여 자료를 찾는 비교검색인 반면, 해싱은 검색 키 값만으로 원하는 자료의 위치를 직접 계산할 수 있다는 점에서 차이가 있다.**

- 해싱함수

> **검색 키값을 이용해 저장된 자료의 주소를 변환하는 함수이다.**

> **어떤 해싱함수를 사용하느냐에 따라 검색속도와 효율성이 결정되기 때문에 해싱함수가 굉장히 중요해진다.**

- 해싱테이블

> **해싱 함수를 통해 계산된 주소에 따라 자료를 저장하는 자료구조를 해시테이블이라 한다.**

> **보통 배열이 선호되며, 버킷을 통해 생성되기도 하는데, 버킷이란 해시 테이블에 저장되는 각각의 노드를 칭한다.**

- ◆ **버킷을 사용할 경우 버킷이 실제 자료가 저장되는 주소가 된다.**
- ◆ **1개의 버킷에 여러 개의 자료가 저장 될 수 있으며, 이때 버킷 내부에는 여러 개의 슬롯이 존재하게 된다.**
- ◆ **서로 다른 검색 키값을 가지지만, 같은 버킷에 저장된 키값들을 동거자라 칭한다.**

- 해싱특징

> **해싱은 검색키를 계산해 찾으려는 자료의 위치를 바로 알 수 있기 때문에 검색에 필요한 평균 시간복잡도가 O(1)이 가능하다는 특징이 있다.**

> **단순히 키 값에 대한 산술적 연산만 하면 찾을 수 있기에 상수시간내에 검색할 수 있다.**

> **단, 이런 최적의 해싱이 가능하려면 해싱함수와 해싱테이블의 설계시에 고려해야 할 몇 가지 제약사항이 존재한다.**

> **제약사항 1. 해싱검색**

→ 해싱 검색은 먼저 전달받은 검색 키 값으로 버킷의 주소를 계산한다.

→ 즉 해싱함수를 이용해 검색 키에 대한 주소를 계산한 후 만약 계산된 주소에 자료가 있으면 검색이 성공한 것이고, 계산된 주소에 자료가 없다면 검색은 실패로 종료된다.

→ 해싱을 이용한 검색의 순서

1) **검색 자료의 주소계산** : 검색 키값을 입력 값으로 하여 해싱 함수로 계산
2) **해시 테이블의 자료점검** : 앞단계에서 계산된 주소로 해시 테이블에서 실제 자료가 있는지 점검
3) **검색 완료** : 검색 대상항목이 있으면 검색성공, 없다면 실패

> **제약사항 2. 자료의 저장**

→ 앞서 검색과 마찬가지로 전달받은 검색 키값에 대해 해싱함수를 이용하여 주소를 계산한다.

→ 만약 계산된 주소에 자료가 없으면, 해당 위치에 자료를 저장하면 되고, 있다면 충돌이 발생했다는 뜻이다.

→ 충돌이 발생하는 가장 큰 이유는 이상적인 해싱이 현실적으로 힘들기 때문이다.

→ 이상적인 해싱은 발생가능한 모든 검색 키 값에 대해 해시테이블에 저장공간을 만들어 두는 경우인대, 이럴경우 불필요하게 저장공간이 낭비 될 수 있다.

→ 실제 해싱에서는 여러 개의 키값에 대해 하나의 공간을 할당하는 것이 일반적이다.

- 해싱함수 상세

> **해싱검색의 효율성을 측정하는 지표는 충돌발생빈도, 해시테이블 사용률, 해싱함수 계산속도이다.**

> **해싱함수는 충돌이 드물게 발생하고 해시 테이블을 고르게 사용할 수 있도록 주소를 계산할 수 있어야 하며, 동시에 계산속도 또한 빨라야 한다.**

> **대표적인 해싱함수**

1) 나머지(제산) 함수

- 가장 일반적으로 사용되는 함수 중 하나로 검색 키 값 k를 해시테이블의 크기 m으로 나눈 나머지를 해시 주소로 사용한다.
- 이 함수는 어떤 크기의 해시테이블에서도 사용할 수 있지만, 테이블의 크기 m이 소수(prime number)인 경우 충돌 발생 빈도가 낮아지고 해시 테이블의 사용률이 증가하는 특성이 있다.
    - 그래서 보통 해시테이블의 크기를 소수로 하는 것이 권장된다.

2) 접기(접지) 함수

2-1) 이동 접기 함수

- 키 값을 일정한 자릿수 만큼 자른 후 오른쪽 끝자리가 일치하도록 맞춘 다음 더하는 방법

2-2) 경계 접기 함수

- ■ 키 값을 일정한 자릿수 만큼 자른 후 분할된 부분들 사이의 경계를 기준으로 반전시켜 더하는 방법

3) 중간-제곱 함수

- ● 검색 키를 제곱한 값 중에서 중간부분을 해시테이블 주소로 이용하는 방법

4) 숫자 분석 방법

- ● 숫자로 이루어진 검색 키 값의 각 자릿수의 분포를 미리 분석하여 해시 주소로 사용하는 방법
- ● 검색 키 값의 각 자릿수 중 가장 편중된 분산을 가지는 자릿수를 생략하고 가장 고르게 분포된(충돌 가능성이 적은) 자릿수만을 추출하여 이를 주소로 사용한다.

> **그 밖에 진법 변환, 비트 추출, 가상 난수 기법 등이 있으니 한번 찾아보도록 하자.**

> **검색 키가 문자열인 경우**

1) 첫번째 문자를 키값으로 사용 : 문자열이 긴 경우에는 문제가 발생할 수 있다.
2) 문자열의 모든 코드 값 더하기 : 문자열을 이루는 모든 문자에 대해 코드값을 더하는 방법으로 글자들의 순서가 바뀐 문자열에 대해서도 같은 값을 가진다는 문제가 있다.
3) 문자의 위치를 고려한 코드 값 더하기 : 문자열을 이루는 각 문자의 코드값을 자신의 위치에 해당하는 값을 곱하여 더하는 방법으로, 보통 "호너의 방법"을 사용한다.
    - ■ 이 방법을 사용할 경우 오버플로우가 발생하는 등 해시테이블의 크기를 벗어나는 값이 계산될 수 있다. 따라서 산출된 검색 키값을 검사하여 정상범위를 벗어나지 않도록 조절 해야 한다.

- 충돌 해결

> **해싱 함수를 잘 선택해 충돌을 예방해야 하지만 충돌이 발생했다면 효율적으로 문제를 해결해야 한다.**

> **해싱함수에 따라 해싱테이블에 저장되는 자료들이 해시테이블에 고르게 분포되지 않고**

특정 주소들 주위로 뭉치는 현상을 군집화 현상이라 한다.

> **군집화는 보통 충돌에 의해 나타나는대 한 테이블 내에서 군집화 현상이 발생하면 효율이 급감한다.**

> **때문에 충돌을 해결하는 기법은 군집화를 최소화 하도록 설계되어야 한다.**

> **해결 기법으로 개방주소법과 체이닝 기법이 있다.**

> **첫번째 해결방법 : 개방주소법**

→ 조사법이라고도 하며, 해싱함수로 산출된 주소가 비어있지 않은 경우 그 다음 주소가 비었는 지 조사하는 방법

→ 대표적인 방법으로 선형 조사법, 제곱 조사법, 이중해싱등이 있다.

1) 선형 조사법

- 충돌이 발생한 경우 주소를 일정한 상수만큼 증가시켜 다시 조사하는 방법
- 특성상 군집화 가능성이 크다는 단점이 있다.

2) 제곱 조사법

- 출동이 발생한 경우 주로를 조사 횟수의 제곱만큼 증가시켜 다시 조사해 보는 방법
- 1차 충돌에서는 1을 더하고, 2차 충돌에서는 2*2를 n차 충돌에서는 n*n을 주소에 더해주는 방식이다.
- 1차 군집화 현상은 어려우나 군집화 자체의 가능성은 여전히 존재한다.
- 해시 테이블의 모든 주소를 조사하기 위해서는 테이블의 크기가 반드시 소수여야 한다는 제약사항이 존재한다.

3) 이중 해싱

- 충돌 발생시 주소를 원래의 해싱함수와는 다른 추가적인 해싱 함수를 통해 주소를 증가시켜 다시 조사하는 방법
- 다양한 이중해싱기법이 존재하나 가장 간단한 방법으로 조사간격을 이용하는 방법이 있다.
- K는 검색키이고, m은 해시테이블의 크기라 가정할 때 조사간격 = m - (k mod m)

의 식을 기존의 주소에 더해 계산하는 방식이다.

- 이 방식은 같은 해싱함수값을 가지더라도 다른 조사순서를 가지도록 해 2차 집중을 줄인다는 장점이 있다.

> **해시테이블 개방주소법 추상자료형**

이름		Input	Output
HT생성	CreateHT	HT크기	HT
HT삭제	DeleteHT	HT	
데이터 추가	addElement	HT, element	TRUE/FALSE
데이터 제거	deleteElement	HT, key	TRUE/FALSE
자료 검색	Search	HT, key	Element
자료 개수	getElementCount	HT	저장된 자료 개수

> **구현**