CS

Created	@March 26, 2023 2:31 PM
Tags	

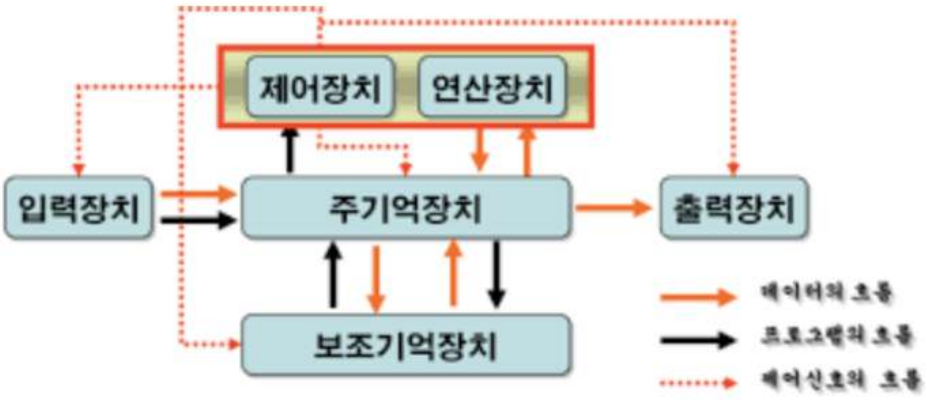
https://gyoogle.dev/blog/computer-science/data-structure/Array vs ArrayList vs LinkedList.html

▼ Computer Architecture

▼ 컴퓨터 시스템

▼ 하드웨어

중앙처리장치(CPU), 기억장치, 입출력장치로 구성되어 있으며,
각 구성 요소는 **시스템버스**(:데이터와 명령 제어 신호를 각 장치로 실어나르는 역할)
로 연결되어 있음



▼ **중앙처리장치(CPU)**

- 역할

  컴퓨터의 가장 핵심적인 역할 수행 (두뇌).
  주기억장치에서 프로그램 명령어와 데이터를 읽어와 처리하며, 명령어의 수행 순서 제어함

- 구성 요소

  ▼ 크게 연산장치, 제어장치, 레지스터로 구성

  - 연산 장치 (=산술논리연산장치, ALU)
    - 산술연산과 논리연산 수행 (비교와 연산 담당)
    - 연산에 필요한 데이터를 레지스터에서 가져오고, 결과를 다시 레지스터로 보냄
  - 제어 장치
    - 명령어를 순서대로 실행할 수 있도록 제어 (명령어 해석과 실행 담당)

        - 주기억장치에서 프로그램 명령어 꺼내 해독 →
          결과에 따라 명령어 실행에 필요한 제어 신호를 기억/연산/입출력 장치로 보냄 →
          장치들이 보낸 신호를 받아, 다음에 수행할 동작 결정
    - 레지스터
        - 고속 기억장치
        - 명령어 주소, 코드, 연산에 필요한 데이터, 연산 결과 등을 임시로 저장
        - 용도에 따라 범용/특수목적 레지스터로 구분
            - 범용 레지스터 : 연산에 필요한 데이터나 연산 결과를 임시로 저장
            - 특수목적 레지스터 : 특별한 용도로 사용하는 레지스터
                - MAR(메모리 주소 레지스터) : 읽기/쓰기 연산을 수행할 주기억장치 주소 저장
                - PC(프로그램 카운터) : 다음에 수행할 명령어 주소 저장
                - IR(명령어 레지스터) : 현재 실행 중인 명령어 저장
                - MBR(메모리 버퍼 레지스터) : 주기억장치에서 읽어온 데이터 or 저장할 데이터 임시 저장
                - AC(누산기) : 연산 결과 임시 저장

- 동작 과정
    1. 주기억장치가 데이터(입력장치에서 입력받은) or 프로그램(보조기억장치에 저장된) 읽어옴
    2. 프로그램 실행을 위해 CPU가 주기억장치에 저장된 프로그램 명령어와 데이터를 읽기 → 처리 → 결과를 다시 주기억장치에 저장
    3. 주기억장치는 처리 결과를 보조기억장치 or 출력장치로 보냄
    4. 제어장치는 1~3 과정에서 명령어가 순서대로 실행되도록 각 장치를 제어함

    - CPU 관련

        ▼ 명령어 세트 : CPU가 실행할 명령어 집합

        - 연산 코드와 피연산자로 이루어짐
            - 연산 코드 : 실행할 연산 (연산, 제어, 데이터 전달, 입출력 기능 가짐)
            - 피연산자 : 필요한 데이터 or 저장 위치 (주소, 숫자/문자, 논리 데이터 등을 저장)
        - 명령어 사이클
            - CPU는 프로그램 실행을 위해 주기억장치에서 명령어를 순차적으로 인출 → 해독 → 실행 반복
              이렇게 CPU가 기억장치에서 한번에 하나의 명령어를 인출해 실행하는데 필요한 일련의 활동을 뜻함
            - 인출/실행/간접/인터럽트 사이클로 나누어짐
            - 주기억장치의 지정된 주소에서 하나의 명령어를 가져오고, 실행 사이클에서는 명령어를 실행함
              하나의 명령어 실행이 완료되면 그 다음 명령어에 대한 인출 사이클 시작

- 인출 사이클과 실행 사이클에 의한 명령어 처리 과정

```
T0 : MAR ← PC
T1 : MBR ← M[MAR], PC ← PC+1
T2 : IR ← MBR

1. PC에 저장된 주소를 MAR(메모리 주소 레지스터)로 전달
2. 저장된 내용을 토대로 주기억장치의 해당 주소에서 명령어 인출
3. 인출한 명령어를 MBR(메모리 버퍼 레지스터)에 저장
4. 다음 명령어 인출을 위해 PC값 증가 시킴 ***
5. MBR에 저장된 내용을 IR(명령어 레지스터)에 전달
```

- 인출한 이후, 명령어를 실행하는 과정

```
T0 : MAR ← IR(Addr)
T1 : MBR ← M[MAR]
T2 : AC ← AC + MBR

이미 인출이 진행되고 명령어만 실행하면 되기 때문에 PC 증가 필요 X
IR에 MBR의 값이 이미 저장된 상태이므로 AC(누산기)에 MBR(메모리 버퍼 레지스터)을 더해주기만 하면 됨
```

▼ 캐시 메모리 : 속도가 빠른 장치와 느린 장치의 속도 차이에 따른 병목 현상을 줄이기 위한 메모리

- 역할
  - CPU가 주기억장치에서 저장된 데이터를 읽어올 때, 자주 사용하는 데이터를 캐시 메모리에 저장 →
    다음 이용 시 캐시 메모리에서 먼저 데이터를 가져오면서 속도를 향상시킴
  - 속도가 빠르지만 용량이 적고 비용이 비쌈
    (캐시 메모리 크기가 작은 이유는 SRAM 가격이 매우 비싸기 때문)
  - CPU에는 이런 캐시 메모리가 2-3개 정도 사용됨 (L1, L2, L3 캐시 메모리)
    L1, L2, L3는 속도와 크기에 따라 분류한 것으로 일반적으로 L1 캐시부터 먼저 사용됨
    (CPU에서 가장 빠르게 접근 → 데이터 찾지 못하면 L2로 이동)
  - 디스크 캐시 : 주기억장치(RAM)와 보조기억장치(하드디스크) 사이에 존재하는 캐시
  - 듀얼 코어 프로세서에서의 캐시 메모리
    - 각 코어마다 독립된 L1 캐시 메모리가 있으며, 두 코어가 공유하는 L2 캐시 메모리가 내장됨
      ex. 만약 L1 캐시가 128kb일 시, 64/64로 나눠 64kb에 명령어를 처리하기 직전의 명령어를 임시 저장,
      나머지 64kb에는 실행 후 명령어를 임시 저장함 (명령어 세트로 구성. I-Cache - D-Cache)
      - L1 : CPU 내부에 존재
      - L2 : CPU와 RAM 사이에 존재

        - L3 : 보통 메인보드에 존재
- 작동 원리
    - 참조 지역성의 원리
        - 시간 지역성 : for, while 같은 반복문에 사용되는 조건 변수처럼 한번 참조된 데이터는 잠시 후 또 참조될 가능성 높음
        - 공간 지역성 : A[0], A[1]과 같은 연속 접근 시, 참조된 데이터 근처에 있는 데이터가 잠시 후 또 사용될 가능성 높음
    - 캐시에 데이터 저장 시, 이런 참조 지역성(공간)을 최대한 활용하기 위해 필요한 데이터 뿐 아니라 옆 주소의 데이터도 같이 가져옴
    - Cache Hit : CPU가 요청한 데이터가 캐시에 있을 때
    - Cache Miss : 캐시에 없어서 DRAM에서 가져왔을 때
        - Cold miss : 해당 메모리 주소를 처음 불러서 나는 miss
        - Conflict miss : 캐시 메모리에 A,B 데이터를 저장해야 하는데, A,B가 같은 캐시 메모리 주소에 할당되어 있어서 나는 miss
          (direct mapped cache에서 많이 발생함. 주소 할당 문제)
        - Capacity miss : 캐시 메모리 공간 부족해서 나는 miss

      ⇒ 캐시 크기를 키워 문제를 해결하려하면 접근 속도가 느려지고 파워를 많이 먹는 단점 생김
- 구조 및 작동 방식
    - Direct Mapped Cache
        - 가장 기본적인 구조. DRAM의 여러 주소가 캐시 메모리 한 주소에 대응되는 다대일 방식
          (00000,01000,10000,11000인 메모리 주소는 000 캐시 메모리 주소에 맵핑)
        - 캐시 메모리는 “인덱스 필드 + 태그 필드 + 데이터 필드”로 구성됨
        - 간단하고 빠르지만, Conflict Miss가 발생할 수 있음
    - Fully Associative Cache
        - 비어있는 캐시 메모리가 있으면 마음대로 주소를 저장하는 방식
        - 저장할 때는 매우 간단하지만 찾기가 어려움
          조건/규칙 없이 저장하므로 특정 캐시 set 안에 있는 모든 블럭을 한번에 찾아 데이터를 검색해야 함
        - CAM이라는 특수 메모리 구조를 사용해야 하지만 가격이 매우 비쌈
    - Set Associative Cache
        - Direct + Fully 방식
        - 특정 행을 지정하고, 그 행안의 어떤 열이든 비었을 때 저장하는 방식
        - Direct에 비해 검색 속도는 느리지만 저장이 빠르고,
          Fully에 비해 저장이 느린 대신 검색이 빠른 중간형

- 기타
  - 소형 컴퓨터에서는 CPU = 마이크로프로세서라고 부르기도 함

▼ 기억 장치

- 프로그램, 데이터, 연산의 중간결과를 저장하는 장치
  실행중인 프로그램과 같은 프로그램에 필요한 데이터를 일시적으로 저장함
- RAM(주기억장치), ROM도 기억장치에 해당
- 보조기억장치 : ex. 하드디스크
  주기억장치에 비해 속도는 느리지만 많은 자료를 영구적으로 보관 가능

▼ 입출력 장치

- 입력 장치 : 컴퓨터 내부로 자료를 입력하는 장치 (키보드, 마우스 등)
- 출력 장치 : 컴퓨터에서 외부로 표현하는 장치 (프린터, 모니터, 스피커 등)

▼ 시스템 버스

- 하드웨어 구성 요소를 물리적으로 연결하는 선
- 각 구성 요소가 다른 구성 요소로 데이터를 보낼 수 있도록 통로 역할

▼ 용도에 따라 데이터 버스, 주소 버스, 제어 버스로 나뉨

- **데이터 버스**
  - 중앙처리장치와 기타 장치 사이에서 데이터를 전달하는 통로
  - 기억장치와 입출력장치의 명령어와 데이터를 중앙처리장치로 보내거나,
    중앙처리장치의 연산결과를 기억장치와 입출력 장치로 보내는 통로
  - “**양방향**” 버스
- **주소 버스**
  - 데이터를 정확히 실어나르기 위해서는 기억장치 ‘주소’를 정해줘야 함
  - 중앙처리장치가 주기억장치나 입출력장치로 기억장치 주소를 전달하는 통로
  - **“단방향”** 버스
- **제어 버스**
  - 중앙처리장치가 기억장치나 입출력장치에 제어 신호를 전달하는 통로
  - 제어 신호 종류 : 기억장치 읽기 및 쓰기, 버스 요청 및 승인, 인터럽트 요청 및 승인, 클락, 리셋 등
  - 읽기 동작과 쓰기 동작을 모두 수행 ⇒ **“양방향”** 버스

- 기타
  - 컴퓨터는 기본적으로 읽고 처리한 뒤 저장하는 과정으로 이루어짐
    (READ → PROCESS → WRITE)

이 과정을 진행하면서 끊임없이 주기억장치와 소통함
(운영체제가 64bit일 경우 CPU는 RAM으로부터 데이터를 한번에 64비트씩 읽어옴)

▼ 소프트웨어

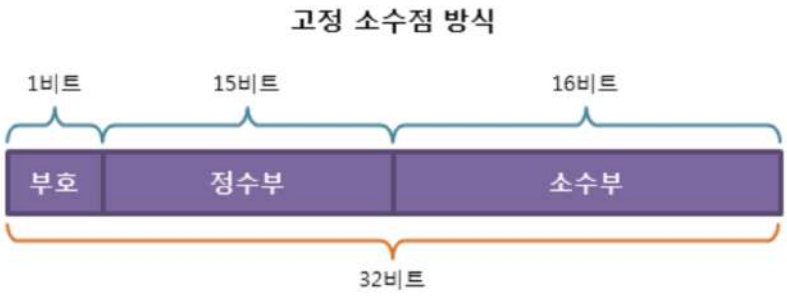
- 시스템 소프트웨어
- 응용 소프트웨어

▼ 데이터 표현

▼ 실수 표현

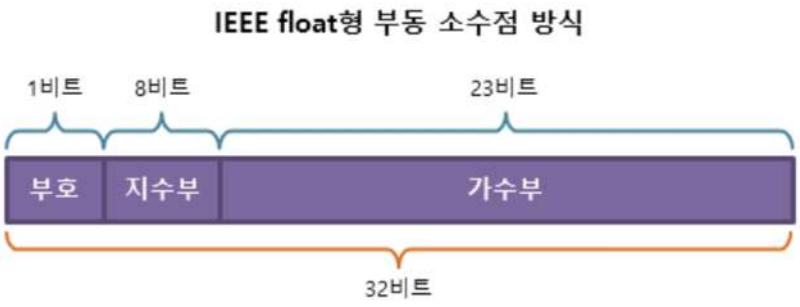
컴퓨터에서 실수를 표현하는 방법은 고정 소수점 / 부동 소수점 두가지 방식 존재

- 고정 소수점



  - 소수점이 찍힐 위치를 미리 정해놓고 소수를 표현하는 방식 (정수 + 소수)
    ex) -3.141592 : 부호(-), 정수부(3), 소수부(0.141592) = 3가지 요소 필요
  - 장점 : 실수를 정수부와 소수부로 표현해 단순함
    단점 : 표현의 범위가 너무 적어 활용 힘듬 (정수부 : 15bit, 수소부 : 16bit)

- 부동 소수점



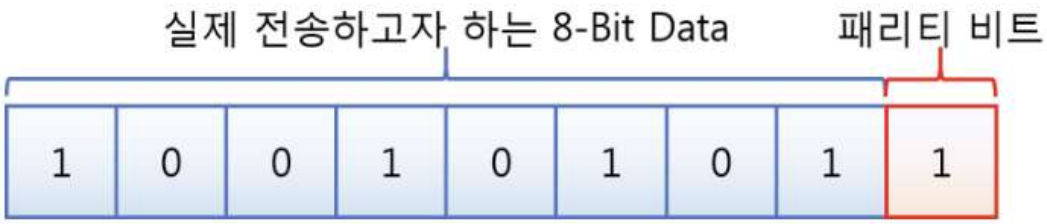
  - 지수의 값에 따라 소수점이 움직이는 방식을 활용한 실수 표현 방법
  - 현재 대부분의 시스템에서 사용하는 방식

- 실수를 가수부 + 지수부로 표현
  - 가수 : 실수의 실제값 표현
  - 지수 : 크기를 표현. 가수의 어디쯤에 소수점이 있는지 나타냄
    즉, 소수점의 위치가 고정되어 있지 않음
- 장점 : 표현할 수 있는 수의 범위 넓음
  단점 : 오차 발생 가능성 있음 (부동소수점으로 표현할 수 있는 방법 매우 다양함)

▼ 패리티 비트 & 해밍 코드

- 패리티 비트와 해밍 코드는 모두 오류 정정 코드의 일종
  - 패리티 비트 : 단일 오류를 감지할 수 있는 간단한 오류 검출 코드
  - 해밍 코드 : 패리티 비트를 포함해 더욱 강력한 오류 정정 코드

▼ 패리티 비트



- 정보 전달 과정에서 오류가 생겼는지 검사하기 위해 추가하는 비트
- 전송하고자 하는 데이터 끝에 +1 비트를 더해 전송하는 방법.
  데이터 블록의 모든 비트를 더해 그 결과가 홀수인지 짝수인지에 따라 1 또는 0로 설정됨
- 이를 통해 전송된 데이터 블록에 하나의 비트가 변경된 단일 오류 감지 가능
  하지만 두 개 이상의 오류나 오류의 위치를 찾아내지는 못함
- 2가지 종류의 패리티 비트(홀수/짝수) 존재

  - 짝수 패리티

실제 송신하고자 하는 데이터의 각 비트의 값 중, 1의 개수가 짝수가 되도록 패리티 비트를 정하는 것
즉, 데이터 비트에서 1의 개수가 홀수면 패리티 비트를 1로 정함
1의 개수가 짝수일 경우 패리티 비트를 0으로 정함

- 홀수 패리티

  전체 비트에서 1의 개수가 홀수가 되도록 패리티 비트를 정하는 방법
  송신하고자 하는 데이터의 각 비트 값 중 1의 개수가 짝수인 경우는 홀수로 맞추기 위해 홀수 패리티 비트의 값을 1로 설정

- 그래서 왜 중요?
  - 데이터를 송수신하는 과정에서 각 비틀르 단위 시간 당 하나씩 보내게 되어있는데,
    이때 알 수 없는 요인에 의해 비트의 값이 틀어져 1이 0으로 변경되거나 0이 1로 변경되면 이를 확인할 수 있게 하기 위함.
  - 즉, 패리티 비트를 정해 데이터를 보내면 받는 쪽에서는 수신된 데이터의 전체 비트를 계산해 패리티 비트를 다시 계산함으로써 데이터 오류 발생 여부 알 수 있음
  - 패리티 비트는 오류 발생 여부만 알 수 있으며 오류를 수정할 수는 없음
  - 시리얼 통신을 하며 데이터가 손실되어 패리티에 의한 데이터 손실 발생을 인지했을 경우
    수신지에서 다시 데이터를 보내달라는 재송신 요청을 할 수 있는 등의 안정적인 통신을 위한 하나의 보호장치
  - 시리얼 통신의 거리가 상당히 멀 경우 주로 적용되며,
    송수신 거리가 짧을 경우 보통 패리티 비트는 사용하지 않고 Checksum 데이터를 추가하는 방법으로 데이터 오류를 검출함

▼ 해밍 코드

- 디지털 통신에서 사용되는 오류 정정 코드 중 하나
- 패리티 비트를 이용해 단일 오류의 정정과 다중 오류의 감지에 모두 사용됨
- 데이터 블록의 비트 수와 패리티 비트의 수를 조합하여 전송된 데이터 블록에서 발견한 오류를 검출하고, 필요한 경우 정정함
  패리티 비트 방식보다 더욱 강력한 오류 검출 및 정정 기능을 제공함
- 단일 오류 정정과 이중 오류 감지를 모두 지원함
  - 단일 오류 정정 : 하나의 오류만을 찾아내 수정 가능
  - 이중 오류 감지 : 두개의 오류를 검출할 수 있지만 어떤 비트에 오류가 발생했는지는 파악 어려움
- 해밍 코드는 전송 중 오류가 발생한 경우에도 메시지의 정확성을 보장함
- 2의 n승 번째 자리인 1,2,4번째 자릿수가 패리티 비트라는 것.
  이 숫자로부터 시작하는 세개의 패리티 비트가 짝수인지 홀수인지 기준으로 판별함
- ex) 짝수 패리티 해밍 코드가 0011011일때 오류가 수정된 코드
  1. 1, 3, 5, 7번째 비트 확인 : 0101로 짝수이므로 '0'
  2. 2, 3, 6, 7번째 비트 확인 : 0111로 홀수이므로 '1'

3. 4, 5, 6, 7번째 비트 확인 : 1011로 홀수이므로 '1'

역순으로 패리티비트 '110'을 도출했다. 10진법으로 바꾸면 '6'으로, 6번째 비트를 수정하면 됨
따라서 **정답은 00110'0'1**

▼ 프로세서

"메모리에 저장된 명령들을 실행하는 유한 상태 오토마톤"

▼ 프로세서란?

- 컴퓨터 시스템에서 중앙 처리 장치(CPU)를 의미
- CPU는 컴퓨터에서 모든 명령어를 처리하고 데이터를 처리하는 핵심적인 요소.
  CPU는 ALU(산술 논리 장치), 제어 유닛, 레지스터 등의 하드웨어 컴포넌트로 구성되어 있음

▼ ARM 프로세서

- ARM 프로세서란?
  - 영국의 ARM Holdings가 개발한 저전력 소비와 높은 성능을 동시에 갖춘 CPU 아키텍처
  - ARM : Advanced RISC Machine (진보된 RISC 기기)
  - RISC : Reduced Instruction Set Computing (감소된 명령 집합 컴퓨팅)
- 특징
  - 주로 모바일 기기, 임베디드 시스템, IoT 기기 등에서 사용됨
    ⇒ 저전력 소비와 높은 효율성 때문. 작은 디바이스에 매우 중요한 요소
  - RISC(Reduced Instruction Set Computing) 아키텍처를 기반으로 하며,
    명령어의 수를 줄이고 단순화하여 명령어의 실행 속도를 높임
  - CPU의 설계에 대한 명확하고 간결한 명령어 세트와 재사용 가능한 논리 블록을 갖추고 있음
    대부분의 명령어를 32비트(또는 64비트)로 처리하며, 다양한 명령어 세트를 제공함
    이로 인해 ARM 프로세서는 높은 성능과 저전력 소비 동시에 실현 가능
  - 모바일 기기를 비롯한 다양한 분야에서 사용되고 있으며, 최근에는 클라우드 컴퓨팅 분야에서도 활용되고 있음
  - ARM 프로세서 기반의 서버가 등장함으로써, 대규모 데이터 처리와 같은 고성능 컴퓨팅 분야에서도 사용될 수 있게 됨
  - 쉬운 프로그래밍 및 작은 크기, 낮은 전력 소비 및 높은 성능을 특징으로 하기 때문에
    이를 위해 ARM 프로세서는 명령어의 길이가 고정되어 있고, 매우 단순한 명령어 세트를 사용하며,
    프로세서 내부에서 파이프라이닝과 슈퍼스칼라 기술 등을 사용하여 명령어 수행 속도를 높임
- 장단점
  - 장점
    - 에너지 효율적이며, 적은 전력으로 작동하여 배터리 수명이 늘어남
    - 저전력이기 때문에 작은 디바이스에서도 사용이 가능

        - 저렴한 가격으로 대량 생산이 가능
        - 소형화 기술에 능숙하여 모바일 기기에서 주로 사용됨
        - 많은 수의 입출력(I/O) 포트와 메모리 칩과의 직접 연결이 가능하여, 빠르고 안정적인 처리가 가능
        - 성능이 우수하며, 병렬 처리와 멀티 태스킹이 가능
        - 널리 사용되기 때문에 다양한 소프트웨어와 하드웨어가 이용 가능
    - 단점
        - 데스크탑 컴퓨터에서의 성능이나 기능면에서는 x86 프로세서에 비해 떨어짐
        - 높은 속도와 성능을 요구하는 응용프로그램에는 적합하지 않음
        - x86 아키텍처와는 다르게 ARM 아키텍처 기반 시스템은 대부분 32비트로 동작하며, 64비트와 같은 고성능 처리를 요구하는 응용프로그램에서는 적합하지 않을 수 있음
        - 다른 하드웨어와의 호환성이 낮아, 하드웨어 디바이스 드라이버 개발 등의 작업이 더 복잡할 수 있음

▼ Data Structure

▼ Array (배열)

- 배열이란?
    - 동일한 자료형의 데이터를 일렬로 나열한 것 (=메모리 상에 원소를 연속으로 배치한 자료구조)
    - 각각의 데이터는 index라는 번호를 가지고 있으며, 해당 index를 통해 배열 특정 위치에 접근 가능
    - 크기를 미리 지정하고, 지정한 크기만큼 데이터 저장 가능
      배열의 크기를 변경해야 할 경우, 보통 새로운 배열을 생성하고 기존 배열의 데이터를 복사해 새로운 배열에 저장함
    - 대부분의 프로그래밍 언어에서 기본적으로 제공되는 자료구조
- 장단점
    - 장점
        - index를 이용해 원하는 위치에 빠르게 접근 가능
        - 데이터를 일렬로 저장하기 때문에, 데이터를 순차적으로 처리하는 경우 빠른 처리 가능
        - 다른 자료구조와 달리 추가적으로 소모되는 메모리 양 거의 없음
        - 메모리 상에 데이터들이 붙어있기 때문에 Cache hit rate가 높음
          (CPU가 값을 가져오려할 때 캐시 메모리에 해당하는 값이 있는 것)
    - 단점
        - 배열의 크기를 변경하는 경우 데이터를 복사해야 하므로 비효율적일 수 있음
        - 배열 중간에 새로운 데이터를 삽입/삭제하는 경우,
          해당 위치 이후의 모든 데이터를 이동해야하므로 비효율적일 수 있음

- 시간복잡도
  - 원소 확인/변경 = O(1)
  - 원소 끝에 추가 / 마지막 원소 제거 = O(1)
  - 임의의 위치에 원소 추가 / 삭제 = O(n)

▼ 구현

▼ 회전 알고리즘

배열 arr의 원소들을 왼쪽으로 한 칸씩 이동시키는 함수

- temp 변수에 배열의 첫 번째 원소 저장
- 반복문을 이용해 배열의 첫 번째 원소를 제외한 나머지 원소를 왼쪽으로 한 칸씩 이동시킴
- 마지막으로 배열의 마지막 원소에 temp값을 대입해 첫 번째 원소를 마지막으로 이동시킴

```
function leftRotatebyOne(arr, n) {
  const temp = arr[0];
  for(let i = 0; i < n-1; i++) {
    arr[i] = arr[i+1];
  }
  arr[n-1] = temp;
}
```

▼ 저글링 알고리즘

배열 arr을 왼쪽으로 d칸씩 회전시키는 함수

- gcd 함수는 최대공약수를 계산하는 함수
- 배열의 인덱스 i부터 시작해 gcd(d,n)만큼 반복 → temp 변수에 현재 index 값 저장
- while문을 이용해 현재 위치에서 d칸씩 이동한 인덱스 k를 계산
- 이동한 인덱스의 값을 현재 인덱스의 값으로 대체
- 이 과정을 k가 처음 인덱스인 i가 될 때까지 반복
- temp값을 마지막 인덱스에 대입해 회전을 완성함

```
function gcd(a, b) {
  if (b === 0) return a;
  else return gcd(b, a % b);
}

function leftRotate(arr, d, n) {
  for (let i = 0; i < gcd(d, n); i++) {
    let temp = arr[i];
    let j = i;

    while (true) {
      let k = j + d;
      if (k >= n) k = k - n;
      if (k === i) break;
      arr[j] = arr[k];
      j = k;
    }

    arr[j] = temp;
```

```
  }
}

function printArray(arr) {
  console.log(arr.join(' '));
}

let arr = [1, 2, 3, 4, 5, 6, 7];
let n = arr.length;

leftRotate(arr, 2, n);
printArray(arr);
```

▼ 역전 알고리즘

- rotateLeft 함수는 먼저 배열의 0번째부터 d-1번째까지의 요소를 reverseArr 함수를 호출하여 뒤집음
- d번째 요소부터 배열의 마지막 요소까지 다시 reverseArr 함수를 호출하여 뒤집음
- 마지막으로 배열 전체를 reverseArr 함수를 호출하여 뒤집음
- 이렇게 되면 원래의 배열에서 왼쪽으로 d번째 위치에 있던 요소가 배열의 처음으로 옮겨지면서 배열이 왼쪽으로 d번 회전하게 됨
- printArray 함수는 배열의 요소들을 차례로 출력함
- 마지막으로 main 함수에서는 초기 배열을 생성하고 rotateLeft 함수를 호출하여 배열을 회전시킨 후, 결과를 printArray 함수를 호출하여 출력함

```
//배열의 요소들을 거꾸로 뒤집는 함수
function reverseArr(arr, start, end){
    while (start < end){
        let temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;

        start++;
        end--;
    }
}

//배열을 d만큼 회전시키는 함수
function rotateLeft(arr, d, n){
    reverseArr(arr, 0, d-1);
    reverseArr(arr, d, n-1);
    reverseArr(arr, 0, n-1);
}

//결과를 출력하는 함수
function printArray(arr, n){
    for(let i = 0; i < n; i++){
        console.log(arr[i] + " ");
    }
}

let arr = [1,2,3,4,5,6,7,8,9,10];
let n = arr.length;
let d = 3;

rotateLeft(arr, d, n);
printArray(arr, n);
```

▼ 배열의 특정 최대 합 구하기

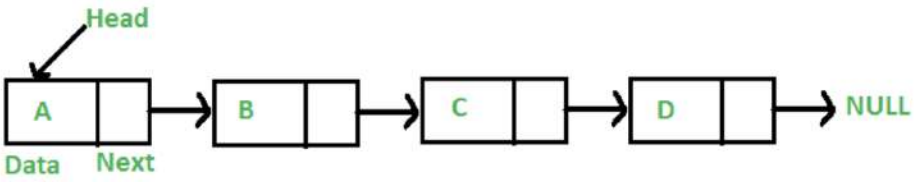
```
function maxVal(arr) {
  let n = arr.length;
  let arrSum = arr.reduce((acc, val) => acc + val, 0); // arr[i]의 전체 합
  let curSum = arr.reduce((acc, val, idx) => acc + idx * val, 0); // i*arr[i]의 전체 합
  let maxSum = curSum;

  for (let j = 1; j < n; j++) {
    curSum = curSum + arrSum - n * arr[n - j];

    if (curSum > maxSum) {
      maxSum = curSum;
    }
  }

  return maxSum;
}

let arr = [1, 20, 2, 10];
console.log(maxVal(arr)); // 72 출력
```

▼ 특정 배열 재배열

배열 A의 요소 중 -1이 아닌 값이 자신의 인덱스 값과 다를 경우,
해당 값이 가리키는 요소를 찾아 해당 요소가 가리키는 다른 요소를 찾는 과정을 반복해
각 요소의 값이 자신의 인덱스와 일치하도록 변경하는 코드

```
function fix(A, len) {
  for (let i = 0; i < len; i++) {
    if (A[i] !== -1 && A[i] !== i) {
      let x = A[i];
      while (A[x] !== -1 && A[x] !== x) {
        let y = A[x];
        A[x] = x;
        x = y;
      }
      A[x] = x;
      if (A[i] !== i) {
        A[i] = -1;
      }
    }
  }
}

function printArray(A, len) {
  for (let i = 0; i < len; i++) {
    console.log(A[i] + " ");
  }
}

let A = [-1, -1, 6, 1, 9, 3, 2, -1, 4, -1];
let len = A.length;
fix(A, len);
printArray(A, len);
```

▼ Linked List (연결 리스트)



- 연결 리스트란?
  - 연속적인 메모리 위치에 저장되지 않는 선형 구조
  - 각각의 노드가 데이터와 다음 노드의 주소를 가리키는 포인터로 이루어진 구조를 갖음
  - 배열과 달리 데이터의 저장 순서가 물리적인 주소와 일치하지 않으며,
    필요한 만큼 동적으로 메모리를 할당하여 사용 가능
    따라, 데이터의 추가나 삭제가 빈번한 경우에 유용하게 사용됨
- 종류
  - 단일 연결 리스트
    - 각 노드가 다음 노드의 주소를 가리키며, 마지막 노드는 NULL을 가리킴
  - 이중 연결 리스트
    - 각 노드가 이전 노드와 다음 노드의 주소를 가리키며, 첫 번째 노드와 마지막 노드는 NULL을 가리킴
  - 원형 연결 리스트
    - 단일 연결 리스트 또는 이중 연결 리스트의 마지막 노드가 첫 번째 노드를 가리키는 구조를 가짐
- 장단점
  - 장점
    - 노드의 추가와 삭제 용이
    - 크기를 동적으로 조정 가능
  - 단점
    - 데이터에 직접 접근 불가능
    - 특정 위치의 노드 탐색을 위해서는 순차적으로 검색 필요
    - 각 요소에 포인터 여분의 메모리 공간 필요

▼ 구현

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
    this.size = 0;
  }

  // 맨 앞에 노드 추가
```

```
  addFirst(data) {
    const newNode = new Node(data);

    if (this.head === null) {
      this.head = newNode;
    } else {
      newNode.next = this.head;
      this.head = newNode;
    }

    this.size++;
  }

  // 특정 노드 다음에 노드 추가
  addAfter(data, prevNodeData) {
    const newNode = new Node(data);

    let current = this.head;

    while (current !== null) {
      if (current.data === prevNodeData) {
        newNode.next = current.next;
        current.next = newNode;
        this.size++;
        return;
      }

      current = current.next;
    }

    console.log(`${prevNodeData}를 찾을 수 없습니다.`);
  }

  // 끝에 노드 추가
  addLast(data) {
    const newNode = new Node(data);

    if (this.head === null) {
      this.head = newNode;
    } else {
      let current = this.head;

      while (current.next !== null) {
        current = current.next;
      }

      current.next = newNode;
    }

    this.size++;
  }

  // 리스트 출력
  printList() {
    let current = this.head;
    let list = '';

    while (current !== null) {
      list += `${current.data} -> `;
      current = current.next;
    }

    list += 'null';
    console.log(list);
  }
}

//앞쪽에 노드 추가
const list = new LinkedList();
```

```
list.addFirst(3);
list.addFirst(2);
list.addFirst(1);
list.printList(); // 1 -> 2 -> 3 -> null


//특정 노드 다음에 추가
const list = new LinkedList();
list.addAfter(3, null); // null을 찾을 수 없습니다.
list.addAfter(2, 1);
list.addAfter(4, 3);
list.addAfter(1, null);
list.printList(); // 1 -> 2 -> 3 -> 4 -> null

//끝쪽에 노드 추가
const list = new LinkedList();
list.addLast(1);
list.addLast(2);
list.addLast(3);
list.addLast(4);
list.printList(); // 1 -> 2 -> 3 -> 4 -> null
```

▼ Array vs ArrayList vs LinkedList

▼ Array

- 가장 기본적인 자료구조 중 하나로, 동일한 타입의 데이터를 연속된 메모리 공간에 저장하는 자료구조
- Index를 사용해 배열 내 특정 요소에 빠르게 접근 가능
- 배열의 크기를 변경할 수 없으며 미리 크기를 지정해야 함
- 배열에 새로운 요소를 추가/제거하려면 다른 요소들을 이동해야 하므로 비효율적

▼ ArrayList

- Array를 기반으로 구현된 동적 배열
- Array와 마찬가지로 index를 사용해서 요소에 빠르게 접근 가능
- Array와 달리 크기를 동적으로 조정 가능
- ArrayList에 새로운 요소를 추가/제거할 때는 Array처럼 다른 요소들을 이동해야 하므로 비효율적

▼ LinkedList

- 데이터를 저장하는 노드와 그 노드의 다음 노드를 참조하는 포인터로 구성된 자료구조
- index를 사용해 요소에 접근 불가능
- 새로운 요소 추가/제거 시 Array나 ArrayList처럼 다른 요소들을 이동할 필요 없음
- LinkedList는 각 노드가 다음 노드를 가리키는 포인터를 가지고 있으므로 삽입/삭제가 매우 빠름
- 메모리를 더 많이 사용할 수 있음
- 단일/다중 등 여러 종류의 연결 리스트 존재
  단일 : 뒤의 노드만 가리킴
  다중 : 앞,뒤 노드를 모두 가리킴

▼ Stack & Queue (스택 & 큐)

▼ Stack

- 스택이란?
  - 한 쪽 끝에서만 자료를 넣거나 뺄 수 있는 자료구조
  - 후입선출(Last in First Out, LIFO)
  - ex) 쌓아올린 책, 그릇 등의 모습
- 스택의 요소
  - push : 스택에 새로운 요소를 추가하는 연산
  - pop : 스택 가장 위에 있는 요소를 제거하고 반환하는 연산
  - Top : 스택에서 가장 위에 있는 요소를 가리키는 포인터. Top의 값은 스택의 크기와 함께 관리됨 (스택 포인터 SP : 다음 값이 들어갈 위치를 가리키고 있음)
- 특징
  - 주로 함수 호출(call stack)을 구현하거나, 괄호 감사, 백트래킹 같은 알고리즘에서 활용됨
  - 배열이나 연결리스트로 구현 가능

▼ 구현

```
class Stack {
  constructor(size) {
    this.top = -1; // 스택의 top을 -1로 초기화합니다.
    this.maxSize = size; // 스택의 최대 크기를 size로 지정합니다.
    this.stack = new Array(size); // 스택을 배열로 초기화합니다.
  }

  push(data) {
    if (this.isFull()) { // 스택이 꽉 차면 오류 메시지를 출력합니다.
      console.log("Stack is full!");
      return;
    }
    this.top++; // top을 1 증가시키고
    this.stack[this.top] = data; // 데이터를 스택에 추가합니다.
  }

  pop() {
    if (this.isEmpty()) { // 스택이 비어있으면 오류 메시지를 출력합니다.
      console.log("Stack is empty!");
      return;
    }
    let data = this.stack[this.top]; // top 위치의 데이터를 저장하고
    this.top--; // top을 1 감소시킵니다.
    return data; // 저장한 데이터를 반환합니다.
  }

  isEmpty() {
    return this.top === -1; // top이 -1이면 스택이 비어있는 것입니다.
  }

  isFull() {
    return this.top === this.maxSize - 1; // top이 스택의 최대 크기 - 1과 같으면 스택이 꽉 찬 것입니다.
  }
}
```

▼ 동적 배열 스택

push 함수에서 스택이 가득찼을 때 기존 스택의 2배 크기만큼 임시 배열을 만듬 →
array copy를 통해 stack의 index 0부터 MaxSize 만큼을 arr 배열의 0번째터 복사 →
복사한 arr의 참조값을 stack에 덮어씌움 →
마지막으로 MaxSize의 값을 2배 증가시켜줌
= 스택이 가득찼을 때 자동으로 확장되는 스택 구현

```
push(data) {
  if (this.isFull()) { // 스택이 꽉 차면
    this.maxSize *= 2; // maxSize를 2배로 늘리고
    let newStack = new Array(this.maxSize); // 더 큰 스택을 만듭니다.
    for (let i = 0; i <= this.top; i++) { // 기존 스택에 있는 데이터를 새 스택으로 복사합니다.
      newStack[i] = this.stack[i];
    }
    this.stack = newStack; // 새 스택을 기존 스택으로 바꿉니다.
  }
  this.top++;
  this.stack[this.top] = data;
}
```

▼ 스택을 연결리스트로 구현

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class Stack {
  constructor() {
    this.top = null;
    this.size = 0;
  }

  push(data) {
    let newNode = new Node(data);
    if (this.top === null) {
      this.top = newNode;
    } else {
      newNode.next = this.top;
      this.top = newNode;
    }
    this.size++;
  }

  pop() {
    if (this.isEmpty()) {
      console.log("Stack is empty!");
      return;
    }
    let data = this.top.data;
    this.top = this.top.next;
    this.size--;
    return data;
  }

  isEmpty() {
    return this.top === null;
  }

  getSize() {
    return this.size;
```

```
  }
}
```

▼ Queue

- 큐란?
  - 먼저 들어온 데이터가 먼저 나가는 선입선출(First in First Out, FIFO)의 자료구조
  - ex) 줄서기, 대기열 등과 유사한 개념
- 구성 요소
  - front 포인터 : 가장 먼저 들어온 데이터를 가리킴
  - rear 포인터 : 가장 마지막에 들어온 데이터를 가리킴
  - Enqueue : 큐에서 데이터를 추가하는 연산
  - Dequeue : 큐에서 데이터를 삭제하는 연산
  - isEmpty() : 큐가 비어있는지 확인
  - isFull() : 큐가 가득 차있는지 확인
- 특징
  - 주로 데이터의 입출력이 잦은 시스템에서 사용됨
    ex) 프린터 출력 대기열, 네트워크 패킷 대기열, 프로세스 스케줄링 등에서 사용됨
- 종류

  ▼ 일반 큐

  - 큐에 빈 메모리가 남아 있어도 꽉 차있는 것으로 잘못 판단할 수 있음 (rear가 끝에 도달했을 때)

```
class Queue {
  constructor(capacity) {
    this.items = new Array(capacity);
    this.capacity = capacity;
    this.front = 0;
    this.rear = -1;
    this.count = 0;
  }

  enqueue(item) {
    if (this.isFull()) {
      throw new Error("Queue is full");
    }
    this.rear++;
    this.items[this.rear] = item;
    this.count++;
  }

  dequeue() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    const item = this.items[this.front];
    this.items[this.front] = undefined;
    this.front++;
    this.count--;
```

```
    return item;
  }

  isEmpty() {
    return this.count === 0;
  }

  isFull() {
    return this.count === this.capacity;
  }
}
```

▼ 원형 큐

- 배열의 끝과 시작을 연결해 배열의 처음과 끝이 연결된 원형 형태로 데이터를 저장함
  ⇒ front와 rear 값이 일치하는 경우에도 큐가 비어있는 상태와 가득찬 상태 구분 가능함
- 초기 공백 상태 : front = 0, rear = 0
  공백, 포화 상태를 구분하기 위해 자리 하나를 항상 비워둠
- 메모리 공간은 잘 활용하지만 배열로 구현되어 있기 때문에 큐의 크기가 제한됨

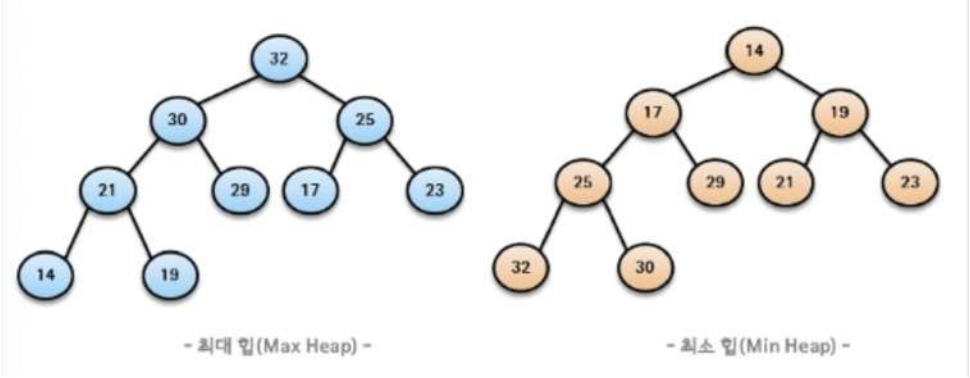
```
class Queue {
  constructor(capacity) {
    this.items = new Array(capacity);
    this.capacity = capacity;
    this.front = 0;
    this.rear = -1;
    this.count = 0;
  }

  enqueue(item) {
    if (this.isFull()) {
      throw new Error("Queue is full");
    }
    this.rear = (this.rear + 1) % this.capacity;
    this.items[this.rear] = item;
    this.count++;
  }

  dequeue() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    const item = this.items[this.front];
    this.items[this.front] = undefined;
    this.front = (this.front + 1) % this.capacity;
    this.count--;
    return item;
  }

  isEmpty() {
    return this.count === 0;
  }

  isFull() {
    return this.count === this.capacity;
  }
}
```

▼ 연결리스트 큐

- 크기 제한이 없고 삽입삭제가 편리함

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class Queue {
  constructor() {
    this.front = null;
    this.rear = null;
    this.count = 0;
  }

  enqueue(item) {
    const node = new Node(item);
    if (this.isEmpty()) {
      this.front = node;
      this.rear = node;
    } else {
      this.rear.next = node;
      this.rear = node;
    }
    this.count++;
  }

  dequeue() {
    if (this.isEmpty()) {
      throw new Error("Queue is empty");
    }
    const item = this.front.data;
    this.front = this.front.next;
    this.count--;
    return item;
  }

  isEmpty() {
    return this.count === 0;
  }
}
```

▼ Heap (힙)



- 힙이란
  - 이진 트리를 기반으로 한 자료구조 중 하나 (이진 트리 : 모든 노드의 자식 노드 개수가 최대 2개인 트리)
  - 여러 값 중 최대값과 최소값을 빠르게 찾아내도록 만들어진 자료구조
- 특징
  - 완전 이진 트리 형태를 띄어야 함
  - 루트 노드는 최대값 또는 최소값을 가짐
  - 모든 노드의 자식 노드들은 부모 노드의 값보다 작거나 큰 값을 가짐
  - 같은 레벨의 노드들은 왼쪽부터 차례대로 채워져야 함
  - 주로 우선순위 큐와 같은 알고리즘에서 사용되며 시뮬레이션 시스템, 작업 스케줄링, 수치해석 계산 등에서도 사용됨
    특히 큰 데이터 집합에서의 최대/최소값 검색에 유용함
    - 우선순위 큐
      - 우선순위 개념을 큐에 도입한 자료구조. 데이터들이 우선순위를 가지고 있으며, 우선순위가 높은 데이터가 먼저 나감
      - 배열, 연결 리스트, 힙[삽입 : O(logn), 삭제 : O(logn)]으로 구현 가능함
- 종류
  - 최대 힙 (Max Heap) : 루트 노드가 최대 값을 가지며, 자식 노드들의 값은 부모 노드의 값보다 작거나 같음
  - 최소 힙 (Min Heap) : 루트 노드가 최소 값을 가지며, 자식 노드들의 값은 부모 노드의 값보다 크거나 같음
- 힙의 연산
  - 삽입
    - 힙의 마지막 노드의 값을 추가하고, 부모 노드와 비교해 필요한 경우 위치를 교환하며 힙 속성을 유지함
  - 삭제
    - 최대값/최소값을 삭제하고, 힙의 마지막 노드를 다른 루트 노드로 이동시킴
    - 그리고 자식 노드들 중 더 큰 값을 가진 자식 노드(최대 힙일 경우) 또는
      더 작은 값을 가진 자식 노드(최소 힙일 경우)의 값을 비교해 필요한 경우 위치를 교환하며 힙 속성을

유지함

- 탐색

  최대값 또는 최소값을 반환

- 구현
  - 배열
    - 힙을 저장하는 표준적인 자료구조
      (이진 트리는 순서대로 index 할당이 가능하기 때문에, 배열을 사용해 노드 저장 가능)
    - 힙에서 index 1부터 시작해 왼쪽 자식 노드는 2i, 오른쪽 자식노드는 2i+1, 부모 노드는 i/2의 인덱스를 가지게 됨
    - 특정 위치의 노드 번호는 새로운 노드가 추가되어도 변하지 않음
      (ex. 루트 노드(1)의 오른쪽 노드 번호는 항상 3)

      ```
      왼쪽 자식 index = (부모 index) * 2
      오른쪽 자식 index = (부모 index) * 2 + 1
      부모 index = (자식 index) / 2
      ```

  - 최대 힙 구현
    - 최대힙의 삽입

      힙에 새로운 요소 들어옴 →
      일단 새로운 노드를 힙의 마지막 노드에 삽입 →
      부모 노드는 인덱스의 /2이므로, 비교 후 새로운 노드가 더 크면 노드와 교환

      ```
      function insertMaxHeap(x) {
        heapSize++;
        maxHeap[heapSize] = x;

        for (let i = heapSize; i > 1; i = Math.floor(i / 2)) {
          if (maxHeap[Math.floor(i / 2)] < maxHeap[i]) {
            swap(Math.floor(i / 2), i);
          } else {
            break;
          }
        }
      }
      ```

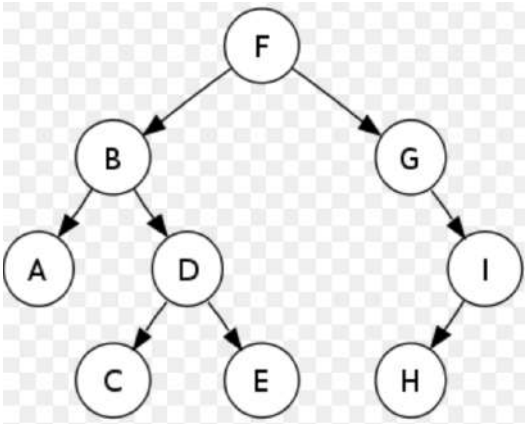
    - 최대힙의 삭제

      최대 힙에서 삭제 연산은 최대값 원소를 삭제하는 것 = 최대값은 루트 노드이므로 루트 노드가 삭제됨 →
      삭제된 루트 노드에는 힙의 마지막 노드를 가져옴 → 힙을 재구성

      ```
      function delete_max_heap() {

          if(heapSize == 0) // 배열이 비어있으면 리턴
              return 0;

          let item = maxHeap[1]; // 루트 노드의 값을 저장
          maxHeap[1] = maxHeap[heapSize]; // 마지막 노드 값을 루트로 이동
          maxHeap[heapSize--] = 0; // 힙 크기를 하나 줄이고 마지막 노드 0 초기화
      ```

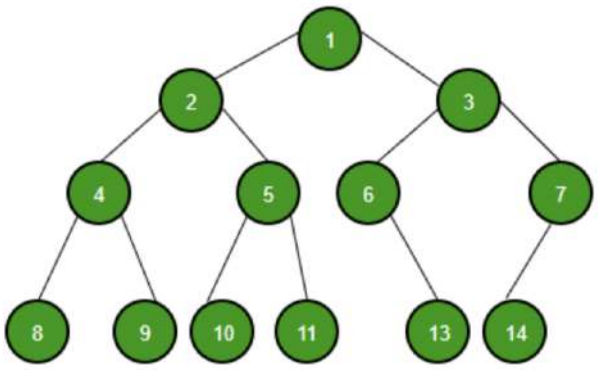
```
    for(let i = 1; i*2 <= heapSize;) {

        // 마지막 노드가 왼쪽 노드와 오른쪽 노드보다 크면 끝
        if(maxHeap[i] > maxHeap[i*2] && maxHeap[i] > maxHeap[i*2+1]) {
            break;
        }

        // 왼쪽 노드가 더 큰 경우, swap
        else if (maxHeap[i*2] > maxHeap[i*2+1]) {
            swap(i, i*2);
            i = i*2;
        }

        // 오른쪽 노드가 더 큰 경우
        else {
            swap(i, i*2+1);
            i = i*2+1;
        }
    }

    return item;

}
```

▼ Tree (트리)



- 트리구조란
  - 계층적인 구조를 표현할 때 사용되는 비선형 자료구조
  - 하나의 루트 노드와 0개 이상의 자식 노드로 구성되며, 자식 노드들은 다시 자신의 자식 노드들을 가질 수 있음
- 특징
  - 하나의 루트 노드를 가지며, 각 노드는 최대 하나의 부모 노드를 가짐
  - 각 노드는 0개 이상의 자식 노드를 가질 수 있음
  - 루트 노드를 제외한 모든 노드는 반드시 하나의 부모 노드를 가짐.
  - 루트 노드와 임의의 노드 사이에는 유일한 경로가 존재하며,
    이런 경로를 통해 계층적 구조 표현 가능

    - 노드와 노드 사이는 간선으로 연결되며, 이 간선은 방향성을 가짐 (노드의 갯수 : N, 간선의 갯수 : N-1)
    - 모든 노드는 자료형으로 표현 가능함
    - 트리는 그래프의 일종으로 분류되지만 일반적인 그래프와 달리,
      트리는 사이클(cycle)이 존재하지 않음
    - 대표적으로는 DB, 운영체제, 그래픽스, 알고리즘 등에서 사용됨
- 트리 순회 방식
    - 깊이 우선 탐색 (Depth-First Search, DFS)
        - 특징
            - 루트 노드에서 출발해, 한 노드의 자식 노드를 모두 방문 후 다음 자식 노드를 방문하는 방식
            - Stack이나 재귀 함수를 이용해 구현 가능함
            - 깊이 우선 탐색은 더 이상 방문할 노드가 없을 때까지 계속해서 자식 노드를 방문하므로,
              최대 깊이까지 파고들게 됨
        - 종류

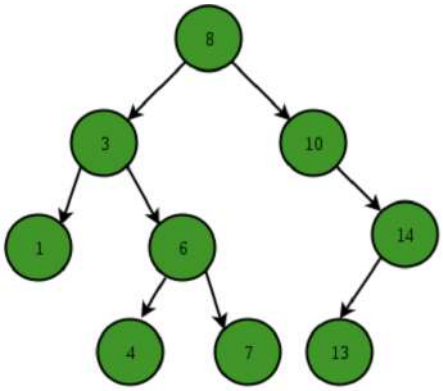


            - 전위 순회 : 현재 노드 → 왼쪽 자식 노드 → 오른쪽 자식 노드 순으로 방문 (1 → 2 → 4 → 8 → 9 → 5 → 10 → 11 → 3 → 6 → 13 → 7 → 14)
            - 중위 순회 : 왼쪽 자식 노드 → 현재 노드 → 오른쪽 자식 노드 순으로 방문 (8 → 4 → 9 → 2 → 10 → 5 → 11 → 1 → 6 → 13 → 3 →14 → 7)
            - 후위 순회 : 왼쪽 자식 노드 → 오른쪽 자식 노드 → 현재 노드 순으로 방문 (8 → 9 → 4 → 10 → 11 → 5 → 2 → 13 → 6 → 14 → 7 → 3 → 1)
    - 너비 우선 탐색 (Breadth-First Search, BFS)
        - 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 13 → 14
        - 루트 노드에서 출발해 같은 레벨에 있는 모든 노드를 먼저 방문 후 다음 레벨로 이동해 반복적으로 방문하는 방식
        - Queue를 이용해 구현 가능
        - 시작 노드로부터 같은 레벨의 노드를 모두 방문한 후, 다음 레벨의 노드를 방문함
        - 루트 노드에서 가장 먼 노드를 찾는 문제 등에서 많이 사용됨

- 트리 자료구조 구현

```
class Tree {
  constructor(rootData) {
    this.root = new Node();
    this.root.data = rootData;
    this.root.children = [];
  }
}

class Node {
  constructor() {
    this.data = null;
    this.parent = null;
    this.children = [];
  }
}
```

▼ Binary Search Tree (이진 탐색 트리)



▼ 이진 탐색 트리란?

- 이진 트리의 일종으로, 데이터를 저장하고 탐색하는 알고리즘 중 하나
- 이진탐색 + 연결리스트
  - 이진탐색 : 탐색에 소요되는 시간복잡도 O(logN : 배열의 크기가 늘어날수록 탐색 속도 빨라짐). 삽입/삭제 불가능

    정렬된 배열에서 특정 한 값을 찾는 알고리즘 중 하나.
    반씩 나누어가며 탐색 범위를 좁혀가는 방법 사용. 탐색 범위를 빠르게 줄여나가면서 원하는 값을 검색

    배열의 중간값 선택 → 찾고자 하는 값과 중간값 비교 → 찾고자 하는 값이 중간값보다 작으면 중간값보다 왼쪽 부분 배열을 다시 탐색 /
    중간값보다 크면 중간값보다 오른쪽 부분 배열을 다시 탐색 / 중간값과 같다면 해당 index 반환

    정렬된 배열에서만 사용 가능하며, 배열을 정렬하는 과정이 필요하지 않으므로 탐색 속도 빠름
  - 연결 리스트 : 삽입,삭제 시간 복잡도는 O(1), 탐색하는 시간 복잡도는 O(N)
  - 이 두가지를 합해 장점을 모두 얻은 것이 '이진탐색트리'
    = 효율적인 탐색 능력을 갖고 자료의 삽입 삭제도 가능하게 만들자

▼ 특징

- 각 노드의 자식이 2개 이하
- 각 노드의 왼쪽 서브트리에는 해당 노드보다 작은 값의 데이터들이 저장되어 있고,
  오른쪽 서브트리에는 해당 노드보다 큰 값의 데이터들이 저장되어 있음
  이런 구조를 가졌기 때문에 데이터 탐색 시 일반적인 배열, 리스트와 달리 탐색 시간을 O(log n)으로 보장할 수 있음
- 중복된 값을 가지지 않음
  검색 목적 자료구조이므로, 중복이 많은 경우 트리를 사용해 검색 속도를 느리게 할 필요 없음
  (트리에 삽입하는 것보다 노드에 count 값을 가지게 해 처리하는게 훨씬 효율적)
- 탐색, 삽입, 삭제 연산의 시간 복잡도는 O(log N)
- 탐색에 유리한 구조이므로 많은 알고리즘에서 사용됨
- 이진 탐색 트리를 확장한 AVL 트리, 레드-블랙 트리 등의 트리도 존재
- 이진탐색트리의 순회는 “중위순회” 방식 (왼 → 루트 → 오)
  중위 순회로 정렬된 순서 읽을 수 있음

▼ 핵심 연산

```
// 이진 탐색 트리 노드 클래스 정의
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}

// 이진 탐색 트리 클래스 정의
class BinarySearchTree {
  constructor() {
    this.root = null;
  }

  // 삽입 메소드
  insert(value) {
    const node = new TreeNode(value);

    if (!this.root) {
      // 루트 노드가 없을 경우 새로운 노드를 루트로 지정
      this.root = node;
      return this;
    }

    let current = this.root;

    while (true) {
      if (value === current.value) {
        // 중복된 값이 들어온 경우
        return undefined;
      }

      if (value < current.value) {
        // 현재 노드 값보다 작은 경우 왼쪽 자식 노드로 이동
        if (!current.left) {
          current.left = node;
          return this;
        }
        current = current.left;
```

```
      } else {
        // 현재 노드 값보다 큰 경우 오른쪽 자식 노드로 이동
        if (!current.right) {
          current.right = node;
          return this;
        }
        current = current.right;
      }
    }
  }

  // 검색 메소드
  search(value) {
    let current = this.root;

    while (current) {
      if (value === current.value) {
        // 값을 찾은 경우 노드를 반환
        return current;
      }

      if (value < current.value) {
        // 현재 노드 값보다 작은 경우 왼쪽 자식 노드로 이동
        current = current.left;
      } else {
        // 현재 노드 값보다 큰 경우 오른쪽 자식 노드로 이동
        current = current.right;
      }
    }

    // 값을 찾지 못한 경우 null 반환
    return null;
  }
}
```

▼ 시간 복잡도

- 균형 트리 : 트리의 높이를 최소화해 탐색 시간을 효율적으로 유지하는 트리
  트리의 높이가 최소화되면 데이터가 균등하게 분포되어있어 모든 노드를 탐색하는데 걸리는 시간이 비슷함

  ⇒ 노드 개수가 N개일 때 O(logN)

- 편향 트리 : 루트 노드를 기준으로 한 쪽이 치우친 형태를 가지는 트리
  트리의 높이가 매우 커져 탐색 시간이 느려질 수 있음.
  따라 데이터의 삽입/삭제/검색 등의 연산이 빈번하게 일어나는 경우 균형 트리 사용하는 것이 더 효율적.
  데이터의 삽입/삭제가 적고 탐색이 많은 경우엔 상대적으로 빠른 속도로 탐색 가능할 수 있음

  ⇒ 노드 개수가 N개일 때 O(N)

- 삽입, 검색, 삭제 시간 복잡도는 트리의 Depth에 비례

▼ 삭제 연산의 3가지 case

1. 자식이 없는 leaf 노드 : 그냥 제거
2. 자식이 1개인 노드 : 지워진 노드에 자식을 올리기
3. 자식이 2개인 노드 : 오른쪽 자식 노드에서 가장 작은 값 or 왼쪽 자식 노드에서 가장 큰 값 올리기

편향된 트리(정렬된 상태 값 트리로 만든 경우 한쪽으로만 뻗음) = 시간복잡도 O(N). 트리 사용할 이유 없음
이를 바로 잡도록 도와주는 개선된 트리 = AVL Tree, RedBlack Tree

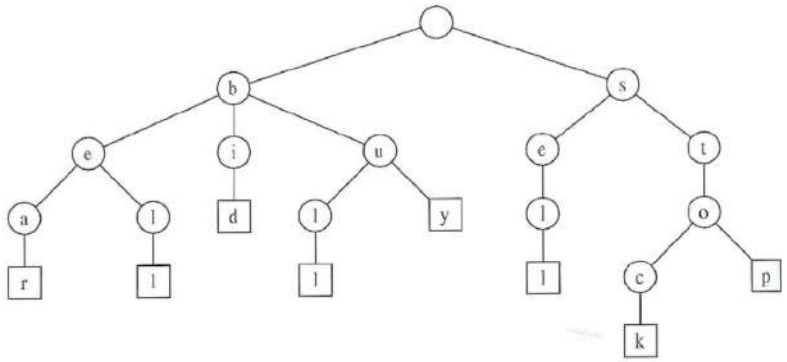
▼ Hash (해시)

- 해시란?
  - key-value 데이터를 저장하는 자료구조
  - 데이터를 저장하는 방법으로 배열을 사용
    배열의 index는 정수 값이어야 하므로, 해시는 key 값을 정수 값으로 변환해 index로 사용함
    이렇게 변환된 정수 값을 배열의 index로 사용해 데이터를 저장하는 것이 해시의 기본 아이디어
  - 검색과 삽입이 빠름
    - 검색 : key 값을 정수 값으로 변환해 배열의 Index로 사용하므로 배열 index만 계산해서 데이터를 검색하면 됨
    - 삽입 : 해시 함수를 사용해 key 값을 정수 값으로 변한한 뒤, 해당 정수 값을 배열의 index로 사용해 데이터를 삽입하기 때문
  - key 값과 해시 함수에 따라 충돌 발생 가능성 있음
  - 삽입/삭제/탐색 시간 복잡도 : O(1)
  - 보안 분야에서도 널리 사용됨.
    키와 해시값 사이에 직접적 연관 없으므로 해시값만 가지고 키를 온전히 복원하기 어려움
    또한 해시 함수는 길이가 서로 다른 입력 데이터에 대해 일정한 길이의 출력을 만들 수 있으므로 '데이터 축약' 기능도 수행 가능
- 해시 함수란?
  - 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수
    매핑 전 원래 데이터의 값 = key, 매핑 후 데이터의 값 = 해시값, 매핑하는 과정 = 해싱
    즉, key 값을 정수 값으로 변환해주는 함수
  - key 값을 배열의 index로 변환할 때 충돌이 발생할 가능성이 있으므로, (=해시 충돌)
    충돌을 최소화하는 좋은 해시 함수를 선택해야 함
  - 충돌이 나는데도 해시 테이블을 사용하는 이유

    적은 리소스로 많은 데이터를 효율적으로 관리하기 위해서.
    예컨대 해시함수로 하드디스크나 클라우드에 존재하는 무한에 가까운 데이터(키)들을
    유한한 개수의 해시값으로 매핑함으로써 작은 크기의 캐쉬 메모리로도 프로세스를 관리할 수 있음
  - 좋은 해시 함수는?
    1. 유일성 : 서로 다른 두 키 값이 같은 정수 값으로 해시되지 않아야 함 ⇒ 해시함수는 언제나 동일한 해시값을 리턴
    2. 균일성 : 가능한 모든 키 값이 균등하게 해시되어야 함
  - 충돌 발생 시 서로 다른 두 key 값이 같은 정수 값으로 해시될 수 있음
  - 충돌 해결 기법
    - 체이닝 : 연결리스트로 노드를 계속 추가하는 방식 (제한 없이 계속 연결 가능하지만 메모리 문제 발생)
    - Open Addressing : 해시 함수로 얻은 주소가 아닌 다른 주소에 데이터 저장할 수 있도록 허용 (해당 키 값에 저장되어 있으면 다음 주소에 저장)

        - 선형 탐사 : 정해진 고정 폭으로 옮겨 해시값의 중복 피함
        - 제곱 탐사 : 정해진 고정 폭을 제곱수로 옮겨 해시값의 중복 피함
    - 해시 테이블
        - 해시 함수를 사용해 키를 해시값으로 매핑하고, 해시값을 index 혹은 주소 삼에 데이터의 값을 키와 함께 저장하는 자료구조
        - 데이터가 저장되는 곳 = bucket(or slot)
        - 해시 테이블의 기본 연산 : 삽입, 삭제, 탐색

▼ Trie (트라이)



- 트라이란
    - 검색 트리의 일종으로 문자열 검색에 특화된 자료구조
    - 트리 구조를 사용해 문자열을 저장하고 검색할 수 있음
    - 루트 노드부터 시작하며, 각 노드는 문자와 자식 노드를 가지고 있음.
      루트노드에서부터 각 문자를 따라 내려가면서 문자열을 저장하는 것이 기본 아이디어
    - 문자열 검색/삽입/삭제 빠름
      검색할 문자열을 Trie의 루트 노드에서부터 각 문자를 따라 내려가며 검색하면 되기 때문
    - 정수형에서 이진탐색트리 이용할 경우 시간복잡도 O(logN)
      but, 문자열에 적용했을 때 문자열의 최대 길이가 M일 경우 O(M*logN)이 됨
      ⇒ 트라이 활용 시 O(M)으로 문자열 검색 가능
    - but, 문자열 검색을 위해 메모리 많이 사용함
      문자열의 길이에 따라 노드의 개수가 증가하기 때문
      따라 문자열 길이가 매우 긴 경우 Trie 사용 어려울 수도 있음
    - 자동완성 기능, 검색어 추천 등의 기능에서 사용됨

▼ 구현

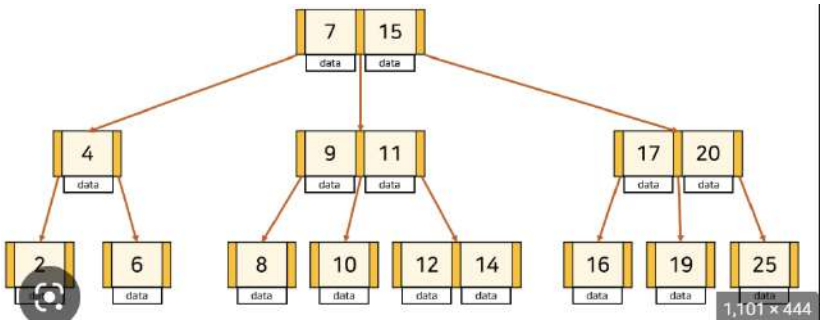
```
class TrieNode {
  constructor() {
    this.children = {}; // 자식 노드를 저장하는 객체
    this.endOfWord = false; // 단어의 끝인지 여부를 저장하는 변수
  }
}
```

```
class Trie {
  constructor() {
    this.root = new TrieNode(); // 루트 노드
  }

  // Trie에 단어를 삽입하는 메소드
  insert(word) {
    let node = this.root;
    for (let i = 0; i < word.length; i++) {
      const char = word[i];
      if (!(char in node.children)) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }
    node.endOfWord = true; // 마지막 노드에 endOfWord를 true로 설정하여 단어의 끝을 표시한다.
  }

  // Trie에서 단어를 검색하는 메소드
  search(word) {
    let node = this.root;
    for (let i = 0; i < word.length; i++) {
      const char = word[i];
      if (char in node.children) {
        node = node.children[char];
      } else {
        return false;
      }
    }
    return node.endOfWord; // 마지막 노드의 endOfWord 값이 true면 true를 반환한다.
  }

  // Trie에서 접두사를 검색하는 메소드
  startsWith(prefix) {
    let node = this.root;
    for (let i = 0; i < prefix.length; i++) {
      const char = prefix[i];
      if (char in node.children) {
        node = node.children[char];
      } else {
        return false;
      }
    }
    return true; // prefix를 모두 검색했을 때, true를 반환한다.
  }
}

// Trie 사용 예시
const trie = new Trie();
trie.insert("apple");
console.log(trie.search("apple")); // true
console.log(trie.search("app")); // false
console.log(trie.startsWith("app")); // true
trie.insert("banana");
console.log(trie.search("banana")); // true
console.log(trie.search("ban")); // false
console.log(trie.startsWith("ban")); // true
```

▼ B Tree & B+ Tree

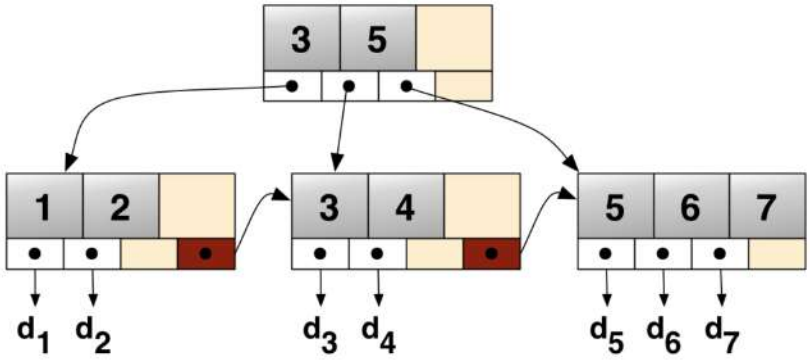
- 이진 트리는 하나의 부모가 두 개의 자식밖에 가지질 못하고,
  균형이 맞지 않으면 검색 효율이 선형 검색 급으로 떨어짐
  하지만 이진 트리 구조의 간결함과 균형만 맞다면 검색/삽입/삭제 모두 O(logN)의 성능을 보이므로 계속 개선시키기 위한 노력이 있음

▼ B Tree



- B Tree란?
  - 디스크 기반의 외부 저장소를 이용한 데이터 구조
  - 이진 트리의 일반화한 형태로 여러 개의 자식을 가지는 노드를 사용해 키를 저장하고 탐색하는 트리
    즉, 이진 트리를 확장해 더 많은 수의 자식을 가질 수 있게 일반화시킴
  - 자식 수에 대한 일반화를 진행하면서, 트리의 균형을 자동으로 맞춰주는 로직까지 갖춤.
    단순하고 효율적이며 레벨로만 따지만 완전한 균형을 맞춘 트리
  - 디스크 입출력이 많이 일어나는 상황에서도 효율적인 탐색과 삽입/삭제 수행 가능
- 규칙
  - 노드의 자료수가 N이면, 자식 수는 N+1이어야 함
  - 각 노드의 자료는 정렬된 상태여야 함
  - 모든 leaf 노드는 같은 depth를 가짐
  - 각 노드는 여러 개의 key-value 쌍을 가질 수 있으며 key는 오름차순으로 연결되어 있음
  - 모든 내부 노드는 적어도 t-1개의 key를 가짐 (루트 노드 제외)
  - 모든 내부 노드는 t개 이상의 자식을 가짐
  - 루트 노드는 적어도 2개 이상의 자식을 가짐
  - 루트 노드를 제외한 모든 노드는 적어도 M/2개의 자료를 가지고 있어야 함
  - 모든 left 노드는 비어있을 수 있으며, 최소 t-1개의 key-value 쌍을 가짐
  - 입력 자료는 중복될 수 없음
- 특징
  - 각 노드에 많은 key-value 쌍을 저장할 수 있으므로 대용량 데이터 관리할 때 유용
    (대량의 데이터를 처리할 때, 검색 구조의 경우 하나의 노드에 많은 데이터를 가질 수 있다는 점은 큰 장점)
  - B트리의 높은 차수는 트리의 높이를 낮춰 탬색 속도를 향상시킴
  - 디스크 입출력을 최소화하기 위해 노드의 크기는 디스크 블록 크기에 맞게 조정되어야 함
  - 데이터베이스 인덱스, 파일 시스템 등에서 널리 사용되는 자료구조
    대부분의 데이터베이스 시스템에서는 B 트리를 기본 인덱스 구조로 사용함

▼ B+ Tree



- B+ Tree란?
  - B Tree의 변형 구조.
  - index 부분과 leaf 노드로 구성된 순차 데이터 부분으로 이루어짐.
    인덱스 부분의 key 값은 leaf에 있는 key 값을 직접 찾아가는데 사용
  - 데이터를 효율적으로 저장하고 검색하는데 사용되는 자료구조 중 하나
  - 많은 양의 데이터를 처리하는데 적합하며, 파일 시스템이나 데이터베이스에서 인덱스 구현에 널리 사용됨
  - 균형 이진 트리와 달리 각 노드에 많은 수의 키를 저장함
  - 일반적으로 B+ Tree의 노드는 하위 노드를 가리키는 포인터와 함께 키 값을 저장함
    각 노드는 키로 정렬되며, 키 값에 따라 자식 노드로 나누어짐
  - 검색/삽입/삭제 작업의 시간 복잡도 = O(log N)
  - 데이터베이스에서 인덱스로 사용될 때 범위 검색 쿼리를 매우 효율적으로 수행 가능
  - 균형 이진 트리보다 복잡하며 구현 어려움
  - 삽입/삭제 작업 시 노드 재배치 및 병합이 필요하므로 이런 작업이 자주 발새앟는 경우 성능에 영향 미칠 수 있음
- 장단점
  - 장점
    - 블럭 사이즈를 더 많이 이용 가능 (key 값에 대한 하드디스크 액세스 주소가 없으므로)
    - leaf 노드끼리 연결리스트로 연결되어있어 범위 탐색에 유리
  - 단점
    - B Tree의 경우 최상 케이스에서는 루트에서 끝날 수 있지만,
      B+Tree에서는 무조건 leaf 노드까지 내려가봐야 함
- B- Tree & B+ Tree

- B-tree : 각 노드에 데이터가 저장
  B+tree : index 노드와 leaf 노드로 분리되어 저장
  (또한, leaf 노드는 서로 연결되어 있어서 임의접근이나 순차접근 모두 성능이 우수)
- B-tree : 각 노드에서 key와 data 모두 들어갈 수 있고, data는 disk block으로 포인터가 될 수 있음
  B+tree는 각 노드에서 key만 들어감. 따라서 data는 모두 leaf 노드에만 존재
- B+tree는 add와 delete가 모두 leaf 노드에서만 이루어짐

▼ Database

▼ Key

- key란?
  - 데이터베이스에서의 key는 데이터를 고유하게 식별하기 위해 사용되는 하나 이상의 열
  - 각 레코드는 key를 포함하며, 이 key를 통해 레코드 식별 가능
  - key는 데이터베이스에서 중복되지 않아야 하며, 데이터베이스의 Primary Key는 이런 요구 사항을 갖춰야 함
    대개, Primary Key는 자동으로 생성되며, 데이터베이스 시스템에서 관리됨
  - 데이터베이스에서는 key를 index로 사용해 검색 및 정렬 작업 수행 가능
    index는 키 값을 저장하며 빠른 검색 및 정렬 작업을 위해 사용됨
  - 검색, 정렬 시 Tuple을 구분할 수 있는 기준이 되는 Attribute
  - DB 설계 시, 적절한 Key를 선택하고 이를 적절히 활용하여 데이터를 효율적으로 관리할 수 있도록 해야 함
- 종류
  1. Primary Key(기본 키)

     특정 레코드를 고유하게 식별하는 데 사용되는 Key
     각 레코드는 반드시 Primary Key를 가져야 하며, 이 Key는 중복되지 않아야함
  2. Foreign Key(외래 키)

     다른 테이블의 Primary Key를 참조하는 Key
     외래 키를 사용하면 여러 테이블을 관계성을 맺을 수 있으며, 데이터의 일관성을 유지할 수 있음
  3. Unique Key(고유 키)

     중복되지 않는 값을 가지는 Key
     Primary Key와 마찬가지로 고유성을 보장하지만, Primary Key와 달리 NULL 값을 허용할 수 있음
  4. Composite Key(복합 키)

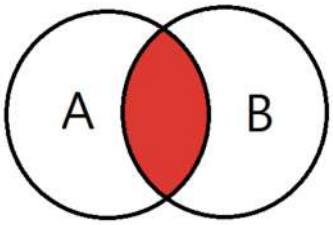
     두 개 이상의 열(column)을 조합하여 만든 Key
     각 열을 개별적으로 사용하는 것보다 더 많은 데이터 식별 정보를 제공할 수 있음
  5. Candidate Key(후보 키)

     Primary Key가 될 수 있는 Key.
     후보 키는 중복되지 않는 값을 가지며, NULL 값을 가질 수 없음.
     Primary Key가 여러 개일 경우 후보 키는 Primary Key로 선택되지 않은 Key를 의미함

6. Alternate Key(대체 키)
   후보 키 중 Primary Key로 선택되지 않은 Key
   대체 키는 데이터를 식별하기 위해 사용될 수 있지만, Primary Key로 사용되지는 않음

▼ Join

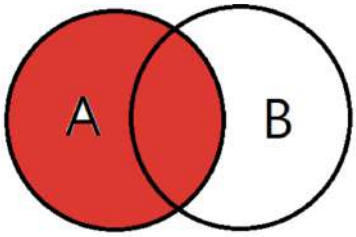
- 두 개 이상의 테이블(table)에서 데이터를 결합하는 작업
- 각 테이블에서 가져온 데이터를 조합하여 하나의 결과 테이블을 생성할 수 있음
- 테이블을 연결하려면 적어도 하나의 컬럼을 서로 공유하고 있어야 하므로 이를 이용해 데이터 검색에 활용함
- 종류
  - INNER JOIN



교집합 = 기준 테이블과 join 테이블의 중복된 값을 보여줌
두 개의 테이블에서 공통된 열(column)을 기준으로 데이터를 결합하는 작업.
INNER JOIN은 기준이 되는 열에서 값이 일치하는 데이터만을 선택함

```
SELECT
A.NAME, B.AGE
FROM EX_TABLE A
INNER JOIN JOIN_TABLE B ON A.NO_EMP = B.NO_EMP
```

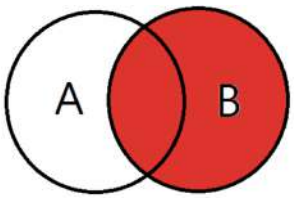
  - LEFT OUTER JOIN



첫 번째 테이블을 기준으로 두 번째 테이블을 결합하는 작업
LEFT JOIN을 사용하면 첫 번째 테이블의 모든 데이터와 두 번째 테이블에서 일치하는 데이터를 선택할 수 있음

```
SELECT
A.NAME, B.AGE
FROM EX_TABLE A
LEFT OUTER JOIN JOIN_TABLE B ON A.NO_EMP = B.NO_EMP
```

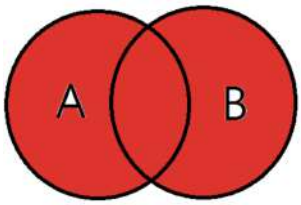
- RIGHT OUTER JOIN



두 번째 테이블을 기준으로 첫 번째 테이블을 결합하는 작업
RIGHT JOIN을 사용하면 두 번째 테이블의 모든 데이터와 첫 번째 테이블에서 일치하는 데이터를 선택할 수 있음

```
SELECT
A.NAME, B.AGE
FROM EX_TABLE A
RIGHT OUTER JOIN JOIN_TABLE B ON A.NO_EMP = B.NO_EMP
```

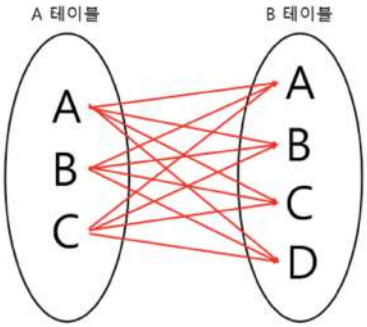
- FULL OUTER JOIN



합집합 = 두 개의 테이블에서 모든 데이터를 선택하는 작업
FULL OUTER JOIN을 사용하면 두 테이블에서 일치하는 데이터와 일치하지 않는 데이터를 모두 선택할 수 있음

```
SELECT
A.NAME, B.AGE
FROM EX_TABLE A
FULL OUTER JOIN JOIN_TABLE B ON A.NO_EMP = B.NO_EMP
```

- CROSS JOIN

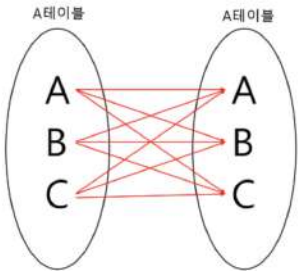


두 개 이상의 테이블에서 조합 가능한 모든 행(row)을 생성하는 작업
CROSS JOIN은 각 행에서 다른 모든 행과 결합하여 결과 테이블을 생성함
ex. A가 3개 B가 4개면 총 3*4 = 12개의 데이터가 검색됨

일반적으로 두 개의 테이블에서만 사용되며, 모든 경우의 수를 생성할 때 사용됨

```
SELECT
A.NAME, B.AGE
FROM EX_TABLE A
CROSS JOIN JOIN_TABLE B
```

- SELF JOIN



하나의 테이블에서 자기 자신을 결합하는 작업을 의미
SELF JOIN은 하나의 테이블에서 여러 개의 별명(alias)을 사용하여 테이블을 조합함

ex. 고객(Customer) 테이블에서는 동일한 지역에 거주하는 고객을 찾기 위해 SELF JOIN을 사용할 수 있음
이 경우, 고객 테이블을 지역 번호와 고객 번호로 JOIN하여 동일한 지역에 거주하는 고객을 찾을 수 있음

```
SELECT
A.NAME, B.AGE
FROM EX_TABLE A, EX_TABLE B
```

▼ SQL Injection

- SQL Injection이란?
  - 웹 애플리케이션에서 보안 취약점을 이용하여 공격자가 악의적인 SQL 쿼리를 데이터베이스에 전송하는 것
  - 취약점을 이용하면 데이터베이스에서 사용되는 SQL 쿼리문을 조작하여 데이터베이스를 비정상적인 동작을 유도할 수 있음
    이를 통해 공격자는 데이터베이스에 있는 기밀 정보를 훔치거나 데이터베이스의 무단 수정 및 삭제를 시도
  - 공격자는 일반적으로 웹 애플리케이션의 입력 폼을 통해 악성 SQL 쿼리를 삽입함
- 공격 방법

1. Union-based SQL Injection
    - 취약점이 존재하는 웹 애플리케이션에서 데이터베이스에 대한 쿼리를 수행할 때 사용되는 UNION 연산자를 이용한 공격
    - UNION 연산자는 두 개의 SELECT 쿼리문을 하나로 합치는 연산자이며, 이를 이용하여 데이터베이스의 데이터를 조작할 수 있음
    - 공격자는 웹 애플리케이션에서 입력 폼을 이용하여 SQL 쿼리문을 조작함
    - 이때, 입력 폼에서 입력한 값을 조작하여 UNION 연산자를 이용하여 다른 SELECT 쿼리문과 함께 실행되도록 유도함
    - 이렇게 하면 공격자는 데이터베이스에서 조회한 데이터를 다른 SELECT 쿼리문과 함께 출력할 수 있으며,
      이를 통해 데이터베이스에서 기밀 정보를 훔칠 수 있음
2. Error-based SQL Injection
    - 취약점이 존재하는 웹 애플리케이션에서 SQL 쿼리문에 대한 에러 메시지를 이용한 공격
    - 에러 메시지는 SQL 쿼리문에서 오류가 발생하면 발생하게 됨
      공격자는 이를 이용하여 데이터베이스에서 기밀 정보를 획득할 수 있음
    - 공격자는 취약점이 존재하는 웹 애플리케이션의 입력 폼을 이용하여 SQL 쿼리문을 조작함
      이때, 입력 폼에서 입력한 값을 조작하여 SQL 쿼리문에서 에러를 발생시키도록 유도함
    - 이렇게 하면 공격자는 에러 메시지를 통해 데이터베이스에서 조회한 데이터를 확인 가능
      이를 통해 데이터베이스에서 기밀 정보를 훔칠 수 있음

- 방어 방법
    1. Prepared Statements 사용
        - 데이터베이스에서 쿼리문을 실행하기 전에 먼저 쿼리문의 구조를 먼저 정의하는 방법
        - 이를 이용해 쿼리문을 생성할 때 사용자의 입력값을 직접 삽입하는 것이 아니라,
          입력값을 매개변수로 전달하여 쿼리문을 실행하는 방법
        - 이를 통해 사용자의 입력값을 검증하고, 쿼리문의 구조를 미리 정의하여 SQL Injection 공격을 방어할 수 있음
        - preparestatement 사용 시 특수문자를 자동으로 escaping해줌
          이를 활용해 서버측에서 필터링 과정을 통해 공격을 방어함
    2. 입력값 검증
        - 사용자가 입력한 데이터를 검증하는 것
        - 입력값의 길이, 입력값의 형식, 특수문자 등을 검사하여 SQL Injection 공격에 사용될 수 있는 문자열을 필터링
    3. 권한 제한
        - 사용자에게 부여되는 권한을 제한하는 것
        - 사용자에게는 필요한 최소한의 권한만 부여하고, 데이터베이스에서 중요한 데이터는 보안이 강화된 서버에서 저장하도록 설정

▼ SQL vs NOSQL

▼ SQL **(관계형 DB)**

- Structured Query Language
- 데이터베이스와 상호작용하는 표준화된 프로그래밍 언어
- SQL을 사용하여 RDBMS에서 데이터를 검색, 삽입, 수정 및 삭제 가능

▼ RDBMS (관계형 데이터베이스)

- 데이터를 테이블 형태로 저장하는 데이터베이스 시스템
- 테이블은 관련된 데이터의 집합이며, 각각의 테이블은 특정 키(primary key)를 사용하여 고유하게 식별됨
- 관계형 데이터베이스에서 데이터를 조작하려면 SQL(Structured Query Language)이라는 표준 언어를 사용
- 테이블은 열(column)과 행(row)으로 구성.
  열은 특정 데이터 유형을 가지며, 각 열은 고유한 열 이름으로 식별됨
  행은 테이블 내의 각각의 데이터 레코드를 나타내며, 각 행은 특정 키(primary key) 값을 가지고 있음
- 데이터를 여러 개의 테이블로 나누고, 이러한 테이블 간의 관계를 정의함으로써 데이터를 구성
  이러한 관계는 기본 키(primary key)와 외래 키(foreign key)를 사용하여 정의
- 외래 키는 다른 테이블의 기본 키를 참조하며, 이를 통해 여러 테이블 간의 관계를 형성함
- 즉 데이터의 중복을 피하기 위해 '관계'를 이용
- 장점
  - 데이터의 중복을 최소화하여 데이터 일관성을 유지 가능
  - 여러 테이블 간의 관계를 정의하여 복잡한 데이터 구조를 처리 가능
  - SQL을 사용하여 데이터를 검색, 삽입, 수정 및 삭제 가능
  - 대규모 데이터베이스에 대한 확장성이 높음
  - 명확하게 정의된 스키마, 데이터 무결성 보장
- 단점
  - 대량의 데이터가 존재할 경우, 데이터베이스의 성능이 저하될 수 있음
  - 데이터의 일관성을 유지하기 위해 여러 개의 테이블을 조작해야 하므로 복잡한 쿼리를 작성하는 데 어려움

▼ **NoSQL (비관계형 DB)**

- Not Only SQL
- 관계형 데이터베이스와 달리 스키마 없는 데이터 모델을 사용하는 비관계형 데이터베이스 시스템
  (관계형 DB의 반대 (스키마도 없고 관계도 없음))
- NoSQL 데이터베이스는 대량의 비정형 데이터를 저장하고 처리하기 위해 설계되었으며, 대부분 분산 환경에서 작동함
- 데이터 모델에 따라 다양한 유형 존재

    - 문서형 데이터베이스(Document Store)

      JSON, BSON 등의 문서 형태로 데이터를 저장하는 데이터베이스. MongoDB, CouchDB 등이 이 유형에 속함

    - 키-값 데이터베이스(Key-Value Store)

      간단한 구조의 키-값 쌍으로 데이터를 저장하는 데이터베이스. Redis, Riak 등이 이 유형에 속함

    - 그래프 데이터베이스(Graph Database)

      노드와 엣지로 구성된 그래프 형태의 데이터를 저장하는 데이터베이스. Neo4j, ArangoDB 등이 이 유형에 속함

    - 칼럼 패밀리 데이터베이스(Column Family Database)
    - 칼럼 그룹으로 구성된 데이터를 저장하는 데이터베이스. HBase, Cassandra 등이 이 유형에 속함
  - 장점
    - 스키마가 없거나 동적인 스키마를 가짐
    - 데이터 분산을 위한 클러스터링을 지원
    - 수평적 확장 용이
    - 대용량의 비정형 데이터를 저장 및 처리하기에 적합
    - ACID(원자성, 일관성, 고립성, 지속성)를 보장하지 않는 경우가 많음
  - 단점
    - 데이터 일관성 유지에 어려움이 있을 수 있음
    - 트랜잭션 처리의 한계가 있을 수 있음
    - SQL을 사용하지 않기 때문에 관계형 데이터베이스에서 지원하는 다양한 쿼리를 사용할 수 없음

▼ Anomaly (이상)

- 잘못된 테이블 설계는 DB에 저장된 데이터의 일관성을 유지하는 것을 어렵게 할 수 있음
- 이는 Anomaly를 발생시키는 주요 원인 중 하나
- 몇 가지 잘못된 테이블 설계로 인해 발생하는 Anomaly 예시
  - 삽입 이상 (Insertion Anomaly)
    - 불필요한 데이터를 추가해야지, 삽입할 수 있는 상황
    - 테이블에 데이터 삽입 시, 데이터 일부가 누락되어 삽입할 수 없는 경우 있음
      이는 일반적으로 테이블에 반복되는 데이터가 있는 경우 발생함
    - ex. 새 고객 정보를 삽입하려면 해당 고객이 이미 주문을 한 경우에도 모든 정보를 함께 삽입해야함
      그렇지 않으면 새로운 고객 데이터를 삽입할 수 없음
  - 삭제 이상 (Deletion Anomaly)
    - 튜플 삭제로 인해 꼭 필요한 데이터까지 함께 삭제되는 문제
    - 테이블에서 데이터 삭제 시, 다른 데이터와 연결된 데이터도 함께 삭제되어 불필요한 데이터 손실이 발생하는 경우가 있음

        - ex. 고객의 주문 정보를 저장하는 테이블에서 특정 고객의 모든 주문을 삭제할 때,
          해당 고객의 모든 주문 정보와 함께 해당 주문에 대한 제품 정보도 함께 삭제됨
    - 갱신 이상 (Update Anomaly)
        - 일부만 변경하여, 데이터가 불일치 하는 모순의 문제
        - 테이블에서 데이터를 갱신할 때, 일부 데이터만 갱신되어 데이터의 불일치가 발생하는 경우가 있음
        - ex. 주문 정보를 저장하는 테이블에서 주문 번호로 행을 식별하는 경우,
          특정 주문의 배송 주소를 변경하면 해당 주문을 식별하는 모든 행의 배송 주소도 함께 변경되어야gka
          그렇지 않으면 데이터의 불일치가 발생
- 이런 Anomaly를 방지하기 위해 효율적이고 일관된 테이블 설계가 필요
- 이를 위해서는 테이블의 정규화(Normalization)와 같은 데이터베이스 설계 원칙을 준수해야함

▼ DB index

- 테이블 내의 행(row)들에 대한 검색 속도를 향상시키기 위한 자료구조
- 특정 컬럼(column) 또는 컬럼 조합에 대한 키(key)와 해당 키가 존재하는 행의 위치를 포함하는 데이터 구조
- 일반적으로 B-Tree나 해시 테이블과 같은 자료구조를 사용하여 구현됨
- 인덱스를 사용 시 DB는 데이터를 검색할 때 전체 테이블을 스캔하지 않고도 인덱스를 사용하여 해당 데이터를 빠르게 찾을 수 있음
  인덱스는 데이터베이스의 성능을 향상시키는 데 매우 유용하며, 대부분의 데이터베이스 시스템에서 지원됨
- index 사용해서 DB에서의 작업
    1. 데이터를 검색할 때 - 인덱스는 특정 키를 사용하여 데이터를 빠르게 찾을 수 있으므로 데이터 검색 작업이 빠름
    2. 데이터를 정렬할 때 - 인덱스는 데이터를 정렬하는 데도 사용됨
    3. 데이터를 그룹화할 때 - 인덱스는 데이터를 그룹화하는 데도 사용됨
- 그러나 인덱스를 생성하는 것은 시스템 자원을 사용하기 때문에 필요하지 않은 인덱스를 생성하면 오히려 성능을 저하시킬 수 있음
  또한 인덱스는 데이터베이스의 크기를 증가시키기 때문에 데이터베이스에서 적절한 인덱스를 생성하고 유지하는 것이 중요

▼ 정규화 (Normalization)

- 관계형 데이터베이스에서 데이터 중복을 최소화하고 데이터 무결성을 보장하기 위한 프로세스
- 데이터 중복이 최소화되면 데이터베이스에서 일관성을 유지하고 데이터 변경이 쉬워지기 때문에
  데이터베이스의 성능을 향상시키는 데 도움됨
- 일반적으로 데이터베이스 설계의 초기 단계에서 수행됨
- 데이터베이스 설계자는 각각의 엔티티(Entity)를 분석하여 중복 데이터와 종속성을 확인한 후, 이를 정규화 과정을 통해 해결함
- 정규화는 1NF, 2NF, 3NF, BCNF, 4NF, 5NF 등의 단계로 나누어져 있으며,
  더 높은 단계일수록 데이터 중복이 최소화되며 더 많은 데이터 무결성을 보장함
  (대체적으로 1~3단계 정규화까지의 과정 거침)

- 정규화 단계
  - 제 1정규화(1NF)

    모든 열이 원자 값(Atomic Value)을 가지도록 테이블을 분해하는 과정
    즉, 각 열이 하나의 값만 가지도록 분해하여 중복을 제거하는 것
    예를 들어, "도시/국가/인구"를 가진 테이블이 있다면, 이를 1차 정규형으로 분해하면 "도시/국가", "국가/인구" 테이블로 분해됨
  - 제 2정규화(2NF)

    부분 함수적 종속성(Partial Functional Dependency)을 제거하는 것
    즉, 기본 키가 아닌 열이 기본 키에 대해 함수적 종속성을 가지지 않도록 분해하는 것
    예를 들어, "주문번호/제품번호/제품명/단가"를 가진 테이블이 있다면,
    이를 2차 정규형으로 분해하면 "주문번호/제품번호/단가", "제품번호/제품명" 테이블로 분해됨
  - 제 3정규화(3NF)

    이행적 종속성(Transitive Dependency)을 제거하는 것
    즉, A->B, B->C와 같은 종속 관계에서 A->C와 같은 종속 관계를 가지지 않도록 분해하는 것
    예를 들어, "제품번호/제품명/제조사/제조사위치"를 가진 테이블이 있다면,
    이를 3차 정규형으로 분해하면 "제품번호/제품명/제조사", "제조사/제조사위치" 테이블로 분해됨
- 정규화를 수행하는 주요 이유
  - 데이터 중복을 최소화하여 저장 공간을 절약함
  - 데이터 무결성을 유지하고 데이터의 일관성을 보장함
  - 데이터베이스의 유연성을 높여 데이터 변경이 쉽게 가능함
- 하지만 정규화가 과도하게 수행되면 성능 저하와 복잡성 증가 등의 부작용이 발생할 수 있음
  ⇒ 적절한 수준의 정규화 수행이 필요
- 또한 정규화는 데이터 중복을 최소화하기 위한 프로세스이지만,
  필요에 따라 중복 데이터를 허용하는 비정규화(denormalization)도 사용됨

▼ 트랜잭션

- 데이터베이스에서 데이터의 논리적인 작업 단위
- 하나의 트랜잭션은 하나 이상의 데이터베이스 연산들을 포함하며,
  이들 연산은 모두 하나의 원자적인 작업 단위로 수행되어야 함
- 즉, 트랜잭션은 데이터베이스에 일어나는 모든 변경 작업들이 완전하게 수행되거나,
  아니면 전혀 수행되지 않도록 보장하는 역할을 함
- 상태 변화 : SQL 질의어를 통해 DB에 접근하는 것
  작업 단위 : 많은 SQL 명령문들을 사람이 정하는 기준에 따라 정하는 것
- 하나의 트랜잭션 설계를 잘 만드는 것이 데이터를 다룰 때 많은 이점을 가져다줌
- 트랜잭션 특징(ACID)
  - 원자성(Atomicity) : 트랜잭션은 데이터베이스에 수행된 모든 연산들이 일관성 있는 상태로 실행되거나,
    실행되지 않도록 보장함

    - 일관성(Consistency) : 트랜잭션의 수행 결과는 일관성 있는 데이터베이스 상태를 유지함
    - 격리성(Isolation) : 트랜잭션의 수행 중에 다른 트랜잭션의 작업이 끼어들지 못하도록 보장함
    - 지속성(Durability) : 트랜잭션의 수행 결과는 영구적으로 유지됨
- 트랜잭션은 BEGIN, COMMIT, ROLLBACK 등의 명령어를 사용하여 제어됨
    - BEGIN : 트랜잭션의 시작을 알림
    - COMMIT : 트랜잭션의 완료와 함께 결과를 데이터베이스에 반영
    - ROLLBACK : 트랜잭션의 취소를 의미
    - 이러한 명령어를 사용하여 데이터베이스에 대한 안전한 작업 수행을 보장할 수 있음
- Transaction 관리를 위한 DBMS의 전략
    - Transaction 관리를 위한 DBMS의 전략에는 로깅(Logging)과 병행 제어(Concurrency Control)가 있음
      이때 DBMS의 구조와 함께 Page Buffer Manager 또는 Buffer Manager, UNDO, REDO가 사용됨
    - 로깅(Logging)

      트랜잭션에서 수행된 데이터베이스 연산들을 기록하는 메커니즘
      트랜잭션에서 수행된 모든 변경 작업들은 로그 파일에 기록되며, 이를 통해 데이터의 원자성, 일관성, 지속성을 보장할 수 있음
      로그 파일에는 변경된 데이터 값과 해당 값이 변경된 위치 등의 정보가 기록되어 있음
    - 병행 제어(Concurrency Control)

      여러 개의 트랜잭션이 동시에 데이터베이스에 접근할 때, 서로 충돌하지 않도록 보장하는 메커니즘
      병행 제어를 통해 데이터의 일관성을 보장하면서도 데이터베이스에 대한 동시 접근성을 향상시킬 수 있음
      병행 제어 기술에는 2PL(Two Phase Locking), MVCC(Multi-Version Concurrency Control), Timestamp 등이 있음

    - DBMS의 구조

      DBMS는 크게 세 가지 구성요소로 이루어져 있음
        - DB : 데이터를 저장하는 공간
        - DBMS 소프트웨어 : 데이터베이스를 관리하는 소프트웨어
        - 데이터베이스 스키마 : 데이터베이스의 구조와 제약 조건을 정의
    - Page Buffer Manager or Buffer Manager
        - DBMS의 메모리 관리 시스템
        - 데이터베이스에서 데이터를 읽거나 쓸 때, 해당 데이터를 담고 있는 페이지를
          디스크에서 읽어와 메모리 버퍼에 저장하고, 변경된 내용을 디스크에 기록
        - 이를 통해 I/O 비용을 줄이고 데이터베이스의 성능을 향상시킴
    - UNDO
        - 로그 파일을 이용하여 트랜잭션 이전 상태로 되돌리는 메커니즘
        - 즉, 트랜잭션이 실행되다가 오류가 발생하여 롤백을 수행하거나, 커밋을 하기 전에 롤백을 수행해야 하는 경우
          UNDO 로그를 이용하여 이전 상태로 되돌리게 됨

    - REDO
        - REDO는 로그 파일을 이용하여 트랜잭션 실행 시 변경된 데이터를 재실행하는 메커니즘
        - 즉, 트랜잭션 실행 중 데이터베이스에 변경이 발생하였지만,
          커밋되기 전에 DBMS가 강제 종료되는 등의 비정상적인 상황이 발생한 경우,
          로그 파일을 이용하여 트랜잭션을 재실행하여 데이터의 일관성을 유지함

▼ 트랜잭션 격리 수존

- 여러 개의 트랜잭션이 동시에 실행될 때, 트랜잭션들 간에 서로 어떤 방식으로 데이터를 공유하는지를 제어하는 기능
- 격리 수준을 높일수록 데이터 무결성을 보장할 수 있지만, 동시성이 감소하게 됨
- 각 격리 수준은 데이터 무결성과 동시성 사이에서 트레이드오프 관계가 있으며, 상황에 맞게 적절한 격리 수준을 선택해야함.
  일반적으로 READ COMMITTED 또는 REPEATABLE READ 격리 수준이 가장 많이 사용됨
- 종류
    1. READ UNCOMMITTED (미커밋된 읽기)
        - 가장 낮은 격리 수준
        - 다른 트랜잭션에서 아직 커밋되지 않은 데이터를 읽을 수 있음
        - 이 때문에 Dirty Read(더티 리드) 문제가 발생할 수 있음
    2. READ COMMITTED (커밋된 읽기)
        - 하나의 트랜잭션이 커밋되기 전까지는 다른 트랜잭션에서 해당 데이터를 읽을 수 없음
        - 이를 통해 Dirty Read 문제는 해결되지만, Non-repeatable Read(비반복 가능한 읽기) 문제가 발생할 수 있음
    3. REPEATABLE READ (반복 가능한 읽기)
        - 트랜잭션이 실행 중일 때 조회한 데이터는 동일한 트랜잭션 내에서 조회해도 항상 같은 결과를 보장
        - 하지만, Phantom Read(유령 읽기) 문제가 발생할 수 있음
    4. SERIALIZABLE (직렬화 가능한)
        - 가장 높은 격리 수준
        - 모든 트랜잭션은 순차적으로 실행되어야 함
        - 이를 통해 Dirty Read, Non-repeatable Read, Phantom Read 문제가 모두 해결됨
        - 하지만, 동시성이 가장 낮아지기 때문에 성능 저하 문제가 발생할 수 있음

▼ 레디스

- 오픈 소스, 인 메모리 데이터 구조 저장소
- key-value 데이터 모델을 사용하며, 메모리에 데이터를 저장하고 디스크에 데이터를 유지함
- 보통 데이터베이스는 하드 디스크나 SSD에 저장하지만 Redis는 메모리(RAM)에 저장해서 디스크 스캐닝이 필요없어 매우 빠름

- 높은 성능, 고 가용성 및 확장성을 제공하여, 인터넷, 응용 프로그램, 게임, 미디어 및 데이터 분석 분야에서 다양한 용도로 사용됨
- 캐시, 메시징, 세션 관리, 대기열 관리 등 다양한 용도로 사용됨
- 대규모 데이터를 처리하는 데 이상적이며, 성능, 가용성 및 확장성이 요구되는 응용 프로그램에서 매우 유용함
- 특징
  - 인 메모리 데이터 저장소 : 모든 데이터를 메모리에 저장함. 이를 통해 빠른 속도를 제공
  - 다양한 데이터 구조 지원 : 문자열, 해시, 목록, 집합, 정렬된 집합 등 다양한 데이터 구조를 지원함
  - 높은 성능 : 비동기식 입출력을 사용하여 빠른 응답 시간을 보장함
  - 클러스터링 : 수평적으로 확장 가능한 클러스터링을 지원함
  - Pub/Sub 지원 : Pub/Sub 메커니즘을 지원하여 메시지 브로커로 사용될 수 있음
  - 데이터 보존 : 메모리에서 데이터를 유지하면서 동시에 디스크에 데이터를 지속적으로 보존함
- RAM 백업 과정
  - snapshot : 특정 지점을 설정하고 디스크에 백업
  - AOF(Append Only File) : 명령(쿼리)들을 저장해두고, 서버가 셧다운되면 재실행해서 다시 만들어 놓는 것
- value
  1. String (text, binary data) - 512MB까지 저장이 가능함
  2. set (String 집합)
  3. sorted set (set을 정렬해둔 상태)
  4. Hash
  5. List (양방향 연결리스트도 가능)

▼ 저장 프로시저

- DB에 미리 저장된 프로그램으로, SQL문과 프로그래밍 언어의 구문을 혼합하여 사용할 수 있는 기능
- DB 내부에 저장되어, 어떤 클라이언트나 애플리케이션에서도 호출 가능
- 저장 프로시저를 사용하면 DB 내에서 비즈니스 로직을 구현할 수 있음
- 일련의 SQL문을 실행하여, 복잡한 데이터 검색 및 조작 작업 수행 가능
- 프로시저 내부에서 조건문, 루프, 변수, 예외처리 등 다양한 프로그래밍 언어의 기능을 사용하여 비즈니스 로직을 작성할 수 있음
- 일반적으로 데이터베이스 관리 시스템(DBMS)에 내장되어 있으며, 대부분의 주요 데이터베이스(DB)에서 지원됨
- SQL Server, Oracle, MySQL, PostgreSQL 등 다양한 데이터베이스에서 저장 프로시저를 사용할 수 있음
- 장점
  - 보안성: 저장 프로시저는 데이터베이스 내부에 저장되어 있으므로, 악의적인 공격으로부터 데이터 보호 가능

    - 성능: 저장 프로시저를 사용하면 데이터베이스 내부에서 실행되므로, 네트워크 오버헤드를 줄이고 성능 향상 가능
    - 재사용성: 저장 프로시저를 사용하면 여러 클라이언트 및 애플리케이션에서 재사용할 수 있으며, 개발 및 유지 보수 비용 줄일 수 있음
- 단점
    - 호환성 : 구문 규칙이 SQL / PSM 표준과의 호환성이 낮기 때문에 코드 자산으로의 재사용성이 나쁨
    - 성능 : 문자 또는 숫자 연산에서 프로그래밍 언어인 C나 Java보다 성능이 느리다.
    - 디버깅 : 에러가 발생했을 때, 어디서 잘못됐는지 디버깅하는 것이 힘들 수 있음