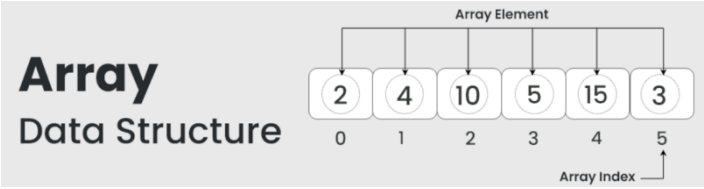
* 배열(Array)이란?

컴퓨터 과학에서 가장 기본적이면서도 중요한 데이터 구조 중 하나이다. 배열은 고정된 크기의 연속적인 메모리 공간을 할당하여 데이터를 저장하는 구조다. 각 데이터는 인덱스 라는 번호를 통해 접근할 수 있기 때문에, 배열을 사용하면 메모리 절감 효과 및 효율적인 데이터 검색이 가능하다.

배열의 개념과 특징 배열은 동일한 데이터 타입의 값을 일정한 크기로 모아 놓은 자료구조(Data Structure)다. 배열 내의 모든 값은 같은 타입이어야 하며, 배열의 크기가 고정되어 있기에 한 번 생성된 배열은 크기를 변경할 수 없다. 즉, 같은 자료형의 데이터를 일정한 크기의 메모리 공간에 연속적으로 저장하는 자료구조다. 이러한 배열을 기반으로 더 복잡한 자료구조가 만들어진다.



Array는 데이터를 한 줄로 줄 세워 놓고 인덱스 번호를 붙여서, 그 번호로 바로 접근할 수 있다.

* 배열의 활용 예
- 배열은 matrix 연산에서 자주 사용된다. (다차원 데이터를 쉽게 처리 가능)
- 정렬 알고리즘에서 중요한 역할을 한다.
- 게임개발 시 좌표정보를 배열로 관리하면 위치를 빠르게 업데이트하고 접근할 수 있다.

* **ndarray는 NumPy의 핵심 자료구조**로, 고성능 과학/수치 연산을 위한 다차원 배열 객체다.
기초적인 연산(합계, 평균, 정렬)을 넘어서 다차원 배열 처리, 브로드캐스팅, 벡터화, 슬라이싱, 마스킹, 선형대수 연산 등 다양한 고급 작업이 가능하다.

numpy의 ndarray로 할 수 있는 고급 작업

주제	핵심 기능/예시
1. 다차원 배열 생성	`np.zeros`, `np.ones`, `np.random`, `np.arange` 등
2. 브로드캐스팅	다른 크기의 배열 간 연산
3. 벡터화 연산	반복문 없이 빠르게 처리
4. 마스킹/조건 필터링	조건에 맞는 요소 추출/변경
5. reshape, axis 연산	`reshape`, `axis=0/1` 방향 지정
6. 선형대수 연산	행렬곱, 전치, 역행렬, 고유값 등
7. 고차원 슬라이싱	2D, 3D 배열 슬라이싱과 인덱싱
8. 고속 집계 함수	`sum`, `mean`, `std`, `argmax`, `unique`, `cumsum` 등

요약해 보면
- 배열은 데이터 저장의 가장 기본 구조, 인덱스로 빠르게 접근 가능,
- 삽입 / 삭제에는 비효율적, 많은 자료구조가 배열을 바탕으로 만들어짐.

*** Python으로 코드 구현 하기 ---**

예1) 학생 점수 배열에서 평균을 구하기

```
import numpy as np

scores = np.array([70, 80, 90, 100, 85])
average = np.mean(scores)
print("평균 점수:", average)
```

예2) 배열에서 가장 큰 값을 찾기

```
scores = np.array([70, 80, 90, 100, 85])
max_score = np.max(scores)
print("최댓값:", max_score)
```

예3) 점수를 오름차순으로 정렬

```
scores = np.array([70, 80, 90, 100, 85])
sorted_scores = np.sort(scores)
print("정렬된 점수:", sorted_scores)
```

예4) NumPy 배열에서 검색

```
scores = np.array([70, 80, 90, 100, 85])

# 특정 값이 존재하는지 확인
print(90 in scores)
print(95 in scores)

# 특정 값의 위치(인덱스) 찾기
result = np.where(scores == 90)
print("90점의 위치(인덱스):", result[0])   # [2]
```

* 참고로 배열은 그 자체로 유용하지만 경우에 따라 다른 데이터 구조가 더 적합할 수 있다. 예를 들어 동적 크기를 지원하는 '연결 리스트'는 데입터의 삽입과 삭제가 빈번한 경우에 유리하다.

자료구조(Data Structure)란

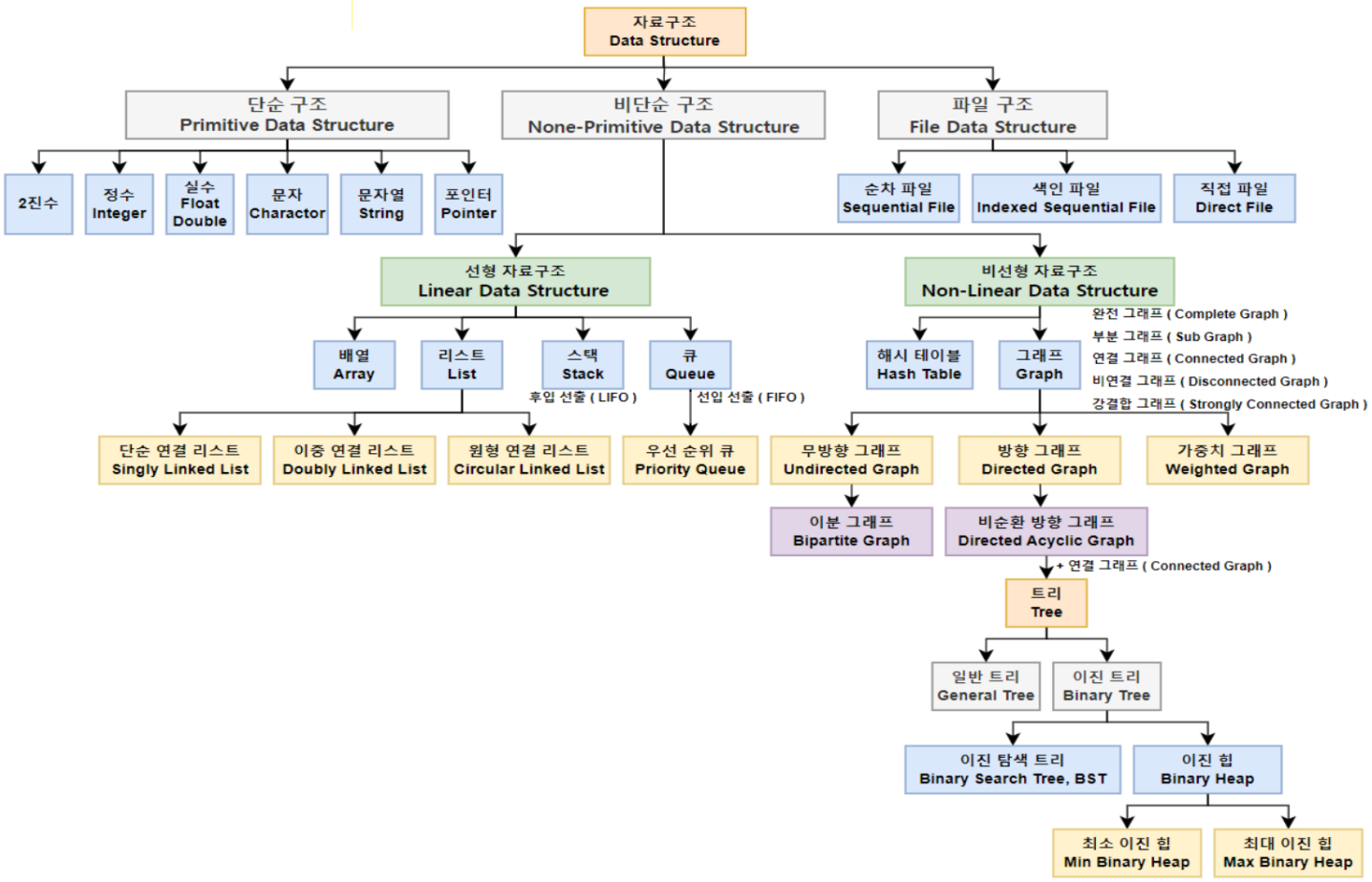
데이터(Data)를 저장(Storage)하고, 조직(Organization)하고, 사용(Access / Operation)하는 체계적인 방식을 말한다.
데이터의 "형태"를 정하고, 그 위에 "동작"을 설계해서, 컴퓨터가 문제를 더 빠르고 효율적으로 해결하도록 만드는 도구라 할 수 있다. 컴퓨터는 수많은 데이터를 처리한다. 그런데 이 데이터를 어떻게 저장하느냐에 따라 속도 및 메모리 사용량도 달라진다. 그래서 자료구조는 "문제를 얼마나 똑똑하게 푸는가"를 결정짓는 중요한 기반이다.

*** 왜 자료구조가 중요한가?**
예를 들어 연락처 앱에서 이름으로 검색할 때, 게시판에서 댓글을 시간순으로 정렬할 때, 게임에서 몬스터를 거리순으로 정렬할 때 이 모든 게 자료구조와 관련된 문제다. 작업에 따라 효율적인 자료구조를 쓰면 검색, 정렬, 삭제 등을 빠르게 할 수 있다.

*** 자료구조의 분류** - 자료구조는 크게 두 가지로 나눌 수 있다.

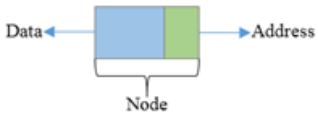
(1) 선형 구조 : 데이터가 일렬로 나열된 구조
- 배열(Array) : 고정 크기, 인덱스로 접근이 빠름 (ex. 파이썬 list)
- 리스트(List) : 유동적 크기, 삽입/삭제가 유리
  - 배열 기반 리스트 (ArrayList)
  - 연결 리스트 (Linked List)
- 스택(Stack) : 나중에 넣은 데이터가 먼저 나감 (LIFO)
- 큐(Queue) : 먼저 넣은 데이터가 먼저 나감 (FIFO)

(2) 비선형 구조 : 데이터가 계층적, 혹은 복잡하게 연결된 구조
- 트리(Tree) : 계층 구조 (ex. 폴더 구조, 족보)
- 그래프(Graph) : 정점(노드) + 간선(연결), 복잡한 관계 (ex. SNS 관계망)



* 기본적인 자료구조의 종류 *

1. 연결된 리스트 (Linked List)



* 노드(Node) - 각 노드는 두 부분으로 구성되어 있다.
1) Data: 실제 저장할 값 (예: 10, 20, 30, 40)
2) Address (Pointer): 다음 노드를 가리키는 주소(또는 참조)

* Linked List의 구조
10 → 20 → 30 → 40 순서로 연결된 노드들
각각의 노드는 다음 노드의 위치(주소)를 알고 있다.
마지막 노드(40)는 더 이상 가리킬 노드가 없기 때문에 X (null)로 표시됨.

* 삽입 예시 (20과 30 사이에 25 추가)
10 → 20 → 25 → 30 → 40
20의 다음 주소를 25로 바꾸고, 25의 다음 주소를 30으로 연결하면 끝!

Linked List는 노드들이 연결된 형태로 데이터를 순차적으로 저장하는 추상 데이터 타입(ADT, 데이터를 다루는 연산을 추상화(abstract)한 개념)이다. 각 노드는 데이터 부분과 다음 노드를 가리키는 주소 부분(링크)으로 구성된다. 이 구조는 동적으로 데이터를 관리하기 위해 사용된다.

장점
- 필요에 따라 메모리를 할당해 크기를 동적으로 조정할 수 있음.
- 어느 위치에나 새로운 노드를 삽입하거나 기존 노드를 삭제하는 과정이 배열에 비해 효율적
-> 주로 링크만 변경하면 됨.

단점
- 배열처럼 인덱스를 통한 접근이 불가능, 특정 요소에 접근하기 위해서는 순차적으로 탐색해야 함.
- 각 노드가 데이터 외에 다음 노드의 주소(링크)를 저장해야 하므로 추가적인 메모리 공간이 필요.

시간복잡도
- 탐색 : O(n) - 특정 요소를 찾기 위해 처음부터 끝까지 순차적으로 탐색해야 함.
- 삽입 및 삭제 : O(1) - 위치를 알고 있을 경우에는 링크만 바꾸면 되므로 빠름.

Array와의 차이점
- 배열은 초기에 크기가 고정되고 메모리 상에서 연속적인 공간에 할당됨. 반면 Linked list는 필요에 따라 각 요소(노드)가 메모리의 임의의 위치에 동적으로 할당됨.
- 배열은 크기 조정이 어렵고 크기 변경을 위해서는 새로운 배열을 생성하고 데이터를 복사해야 함. 하지만 Linked list는 노드의 추가 및 제거를 통해 손쉽게 크기 조절 가능.
- 배열은 인덱스를 통해 O(1)에 임의 접근이 가능하지만 Linked list는 O(n)에 순차 접근해야 함.
- Linked list는 추가적인 메모리(링크 정보)를 사용하지만, 배열은 데이터만큼의 메모리만 사용.

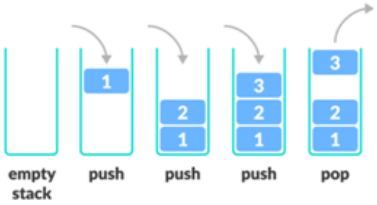
사용 사례 : 데이터를 자주 추가하거나 삭제하는 상황에서 효과적.
예: 사진 앨범, 음악 플레이어의 재생 목록, 브라우저 방문기록, Undo 기능 등.

*** Python으로 코드 구현 예 ---**

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        node = Node(data)
        if not self.head:
            self.head = node
        else:
            cur = self.head
            while cur.next:
                cur = cur.next
            cur.next = node

    def print_list(self):
        cur = self.head
        while cur:
            print(cur.data, end=" → ")
            cur = cur.next
        print("None")

ll = LinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
ll.append(40)
ll.print_list()  # 출력: 10 → 20 → 30 → None
```

2. 스택 (Stack)

스택(Stack)은 후입선출(Last In, First Out, LIFO) 원칙을 따르는 선형 자료구조다. 즉, 가장 마지막에 스택에 추가된 요소가 가장 먼저 제거된다. 이 개념은 실생활에서 대형 마트의 카트를 적재 후 하나씩 꺼내는 방식과 유사하다.



스택(Stack)이란? 후입선출(Last-In, First-Out, LIFO) 구조를 따른다. 나중에 들어간 데이터가 먼저 나옴.
주요 연산:
- push: 데이터를 스택에 넣는 연산
- pop: 데이터를 스택에서 꺼내는 연산

현재 그림은
1) empty stack : 초기에는 아무것도 없는 빈 스택이다.
2) push 1 : 숫자 1을 스택에 넣는다. 스택의 맨 위(top)에 1이 쌓임.
3) push 2 : 숫자 2를 스택에 넣는다. 2가 1 위에 쌓임. (2가 top)
4) push 3 : 숫자 3을 스택에 넣는다. 3이 가장 위에 올라감. (3이 top)
5) pop : 스택에서 맨 위(top)의 요소 3을 꺼낸다.
결과적으로 스택에는 2, 1만 남는다.

장점
- 구현이 간단하다. 배열이나 연결 리스트를 기반으로 쉽게 구현할 수 있음.
- 빠른 연산: 삽입(push)과 삭제(pop) 모두 스택의 상단에서 일어나므로 매우 빠름 → 시간복잡도 O(1)

단점
- 유연성 부족: 오직 한쪽(상단)에서만 데이터를 넣고 뺄 수 있어서, 중간이나 아래쪽 요소에 직접 접근 불가.
- 크기 제한: 배열 기반 스택은 정해진 크기를 초과하면 더 이상 데이터를 넣을 수 없음 (동적 할당 방식은 예외)

시간복잡도
- 탐색(비어 있는지, 가득 찼는지, 상단의 요소를 조회) : O(1)
- 삽입 및 삭제 : O(1) -> 스택 연산이 상단의 요소에만 작용하기 때문

※ 스택은 탐색(search)에는 적합하지 않다, 전체 요소 중간 값을 찾는 경우에 부적합함.

사용 사례
- 웹 브라우저의 뒤로 가기 : 방문한 페이지를 스택에 저장해, pop으로 되돌아감.
- 함수 호출 스택 : 함수가 호출될 때마다 스택에 저장되고, 호출이 끝나면 스택에서 제거됨.
- Undo 기능 : 실행한 작업을 순서대로 쌓아두고, 되돌릴 때 pop으로 복원
- 괄호 검사, 수식 계산 등에서도 많이 사용됨.

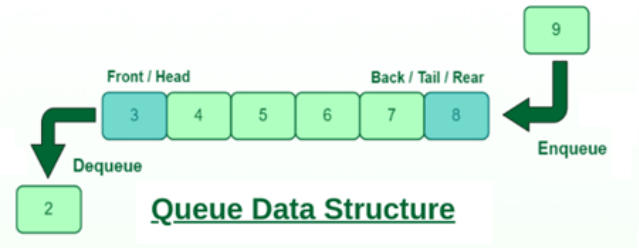
*** Python으로 코드 구현 예 ---**

```
class Stack:
    def __init__(self):
        self.items = []
```

```
    def is_empty(self):
        return len(self.items) == 0      # 스택이 비었는지 확인

    def push(self, item):
        self.items.append(item)          # 스택에 아이템 추가

    def pop(self):
        # 스택에서 아이템 제거 및 반환
        if self.is_empty():
            raise IndexError("스택이 비어 있습니다.")
        return self.items.pop()

    def peek(self):
        # 스택의 최상단 아이템 조회
        if self.is_empty():
            raise IndexError("스택이 비어 있습니다.")
        return self.items[-1]

    def size(self):
        return len(self.items)      # 스택에 있는 요소의 개수 반환

stack = Stack()

stack.push(1)
stack.push(2)
stack.push(3)

print("Top of stack:", stack.peek())          # 출력: 3
print("Pop:", stack.pop())                    # 출력: 3
print("Pop:", stack.pop())                    # 출력: 2
print("Is empty?", stack.is_empty())          # 출력: False
print("Pop:", stack.pop())                    # 출력: 1
print("Is empty?", stack.is_empty())          # 출력: True
```

3. 큐 (Queue)

큐는 한쪽 끝에서는 데이터의 추가(addition)만, 반대쪽 끝에서는 삭제(deletion)만 이루어지는 선형 자료구조다.
선입선출(First In, First Out, FIFO) 원칙을 따른다. 즉, 가장 먼저 추가된 요소가 가장 먼저 제거되는 구조다.
이 개념은 실생활에서 줄을 서는 방식과 매우 유사하다. 먼저 온 사람이 먼저 나가는 것처럼, 큐에서도 먼저 들어온 데이터가 먼저 처리된다.



기본 구조 : 큐는 양 끝이 다르게 동작한다.
- **Front (Head): 데이터를 꺼내는 위치 → Dequeue 연산**
- **Rear (Tail, Back): 데이터를 넣는 위치 → Enqueue 연산**

각 요소의 의미

요소	설명
`2`	큐의 맨 앞에 있던 값. **Dequeue** 되어 제거됨
`3, 4, 5, 6, 7, 8`	현재 큐에 남아 있는 데이터들
`9`	새로 들어올 데이터. **Enqueue** 되어 뒤쪽에 추가됨
화살표 방향	데이터의 흐름을 나타냄 (왼쪽 → Dequeue / 오른쪽 → Enqueue)

큐 자료구조를 배열로 구현한다면, 기본적으로 사용되는 세 가지 주요 구성 요소는 다음과 같다.
- Queue : 큐의 요소들을 저장하는 배열의 이름. FIFO 원칙을 준수한다.
- Front : 큐에서 첫 번째 요소가 저장된 배열의 인덱스. 즉 큐에서 다음에 삭제될 요소의 위치를 나타냄.
  큐에서 요소가 제거될 때마다 Front 인덱스는 다음 요소를 가리키도록 갱신됨.
- Rear : 큐에 마지막으로 추가된 요소가 저장된 배열의 인덱스. 즉 큐의 끝 부분이며, 큐에 요소가 추가될 때마다
  Rear 인덱스는 새로운 요소가 추가된 위치로 업데이트됨.

동작 원리
- Enqueue(요소 추가) : 새로운 요소가 큐에 추가되면, 그 요소는 Rear 인덱스가 가리키는 위치에 저장된다.
  요소가 추기된 후 Rear 인덱스는 배열의 다음 위치로 이동한다.
- Dequeue(요소 제거) : 큐에서 요소를 제거하려면, Front 인덱스가 가리키는 위치의 요소가 제거된다.
  요소 제거 후 front 인덱스는 배열의 다음 위치로 이동하여 다음으로 제거될 요소를 가리킨다.

장점
- 자료의 순서를 유지하면서 요소를 관리. 이는 시간에 따른 데이터 처리에 적합
- 큐의 연산(삽입, 삭제)이 간단
- 양 끝에서 접근이 가능

단점
- 중간 요소의 수정 및 접근이 비효율적임. 즉 큐의 중간에 있는 요소 접근시 앞에 있는 요소를 모두 제거해야 함
- 배열 기반 구현인 경우, 큐가 가득 찬 경우에 재배치가 필요할 수 있음.

시간복잡도
- 삽입 및 삭제 : O(1) - 하지만 동적 배열을 사용하는 경우에는 추가적인 시간이 소요될 수 있음.
- 탐색 (큐의 맨 앞 요소 확인) : O(1)

사용 사례 : 프린트 작업 대기열, 프로세스 스케줄링 등에 사용됨.

*** Python으로 코드 구현 예 ---**

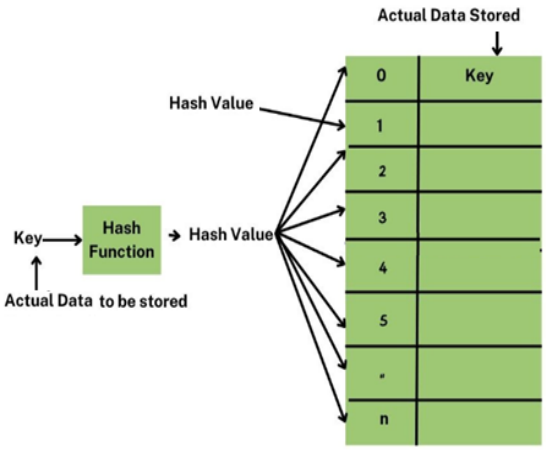
```
# 내장 자료형인 list를 사용하여 enqueue, dequeue, peek, is_empty 등의 연산을 구현

class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0  # 큐가 비어있는지 확인

    def enqueue(self, item):
        self.items.append(item)       # 큐의 뒤쪽에 데이터 추가

    def dequeue(self):
        # 큐의 앞쪽에서 데이터 제거
        if self.is_empty():
            raise IndexError("큐가 비어 있습니다.")
        return self.items.pop(0)

    def peek(self):
        # 큐의 가장 앞 데이터를 확인 (제거하지 않음)
        if self.is_empty():
            raise IndexError("큐가 비어 있습니다.")
        return self.items[0]

    def size(self):
        return len(self.items)  # 큐에 저장된 데이터 개수 반환

q = Queue()
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)

print("Front item:", q.peek())       # 출력: 2
print("Dequeue:", q.dequeue())    # 출력: 2
print("Dequeue:", q.dequeue())    # 출력: 3
print("Is empty?", q.is_empty())    # 출력: False
print("Dequeue:", q.dequeue())    # 출력: 4
print("Is empty?", q.is_empty())    # 출력: True
```

4. 해시 테이블(Hash Table)

해시 테이블은 키(Key)를 값(Value)에 매핑하는 자료구조다. 이 구조는 해시 함수(Hash Function)를 사용하여 키가 저장될 위치(인덱스)를 계산한다. 이 덕분에 특정 키에 대한 검색, 삽입, 삭제 작업을 매우 빠르게(O(1)에 가깝게) 수행할 수 있다.

해시 함수는 키를 고정된 범위의 인덱스로 변환하는 함수이다. 서로 다른 키가 같은 인덱스로 매핑되는 경우를 충돌(Collision)이라 하며, 이를 해결하기 위해 Chaining이나 Open Addressing 등의 방법을 사용한다.

Python에서는 dict 자료형이 해시 테이블 구조를 따른다



Hash Table 구조

1) Key 입력 : 사용자가 저장하고 싶은 실제 데이터가 하나의 Key로 표현된다.

예: "student_id" → "홍길동"

2) Hash Function 적용 : 입력된 Key는 Hash Function을 거친다.

Hash Function은 이 Key를 일정한 범위의 숫자(해시 값, Hash Value)로 변환한다.

이 값은 보통 배열 인덱스로 사용된다.

예: "홍길동" → Hash Function → 4 (→ 4번 인덱스에 저장)

3) 해시 값(Hash Value) → 테이블 인덱스

해시 함수의 결과인 숫자는 해시 테이블의 인덱스로 사용되며 해당 위치에 실제 데이터(Value)를 저장.

[배열 형태로 구현된 테이블]

인덱스: 0 1 2 3 4 5 ...

값: - - - - 홍길동 - ...

4) 실제 데이터 저장

결과적으로 Key → Hash Function → Index 과정을 거쳐, 데이터가 적절한 위치에 빠르게 저장됨.

이후 검색 시에도 같은 키로 해시 값을 계산하면, 즉시 해당 인덱스로 이동해 값을 빠르게 찾을 수 있다.

해시 함수 (Hash Function) : 데이터를 효율적으로 관리하기 위해, 입력된 데이터를 수학적 연산을 통해 고정된 길이의 숫자(또는 값)로 변환하는 함수다. 이 함수는 주로 Key를 해시 테이블 내의 특정 위치(버킷)로 Mapping하는 데 사용된다.

해시 값 (Hash Value) : 해시 함수가 키를 입력 받아 계산한 출력 값이다. 이 값은 해시 테이블에서 키-값 쌍을 저장할 위치(=버킷)를 결정하는 데 사용됨. 예를 들어, 키 'tom'에 해시 함수를 적용했을 때 해시 값이 4가 나왔다면, 'tom'이라는 키와 그에 해당하는 값은 해시 테이블의 4번 위치에 저장된다.

장점
- 빠르게 데이터에 접근할 수 있다.
- 해시 테이블은 필요한 만큼의 메모리만 사용하여 데이터를 저장하므로 공간 효율성이 높다.
- 키를 통해 직접 값에 접근할 수 있어 데이터 관리가 용이하다.

단점
- 두 개 이상의 키가 동일한 해시 값을 가질 때 해시 충돌이 발생할 수 있다.
- 해시 테이블은 키의 순서를 유지하지 않아 순서가 중요한 경우에는 적합하지 않다.
- 해시 테이블이 가득 차면 크기를 조정(리해싱)해야 하는데, 이 과정이 복잡하다.

시간복잡도
- 탐색 : 평균 O(1)이지만 모든 키가 동일한 해시 값을 가지는 경우 최악에서는 O(n)
- 삽입 및 삭제 : 평균 O(1), 최악의 경우 O(n)

사용 사례 : 해시 테이블은 빠른 데이터 검색, 삽입, 삭제가 필요한 상황에서 매우 유용하게 사용된다.
예를 들어,
사용자 이름으로 사용자 정보를 빠르게 검색해야 할 때,
어떤 데이터(예 : 이메일, ID)가 이미 존재하는지 확인해야 할 때,
중복 여부를 빠르게 검사하거나, 대량의 데이터를 효율적으로 관리할 때
이러한 경우에 해시 테이블은 높은 성능을 발휘한다.

*** Python으로 코드 구현 예 ---**

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # 체이닝: 각 칸이 리스트

    def _hash(self, key):
        # 간단한 해시 함수: 문자열을 아스키 값으로 변환 후 모듈로 연산
        return sum(ord(char) for char in key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        # 이미 같은 키가 있으면 업데이트
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        # 없으면 새로 추가
        self.table[index].append((key, value))

    def get(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None  # 키가 없으면 None 반환

    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return True
        return False
```

```
ht = HashTable()

ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("grape", 150)

print("apple:", ht.get("apple"))          # 출력: 100
print("banana:", ht.get("banana"))     # 출력: 200

ht.insert("apple", 300)   # 값 업데이트
print("apple (updated):", ht.get("apple"))        # 출력: 300

ht.delete("banana")
print("banana (deleted):", ht.get("banana"))   # 출력: None

# Python 내장 dict(내부적으로 매우 정교한 해시 테이블)와 비교
my_dict = {}
my_dict["apple"] = 100
print(my_dict["apple"])   # 100
```

5. 그래프(Graph)

그래프는 정점(Node 또는 Vertex)과 이를 연결하는 간선(Edge)으로 구성된 자료구조다. 복잡한 관계나 연결 구조를 표현할 때 자주 사용된다. 그래프는 간선의 방향 유무에 따라 다음 두 가지로 나뉜다.

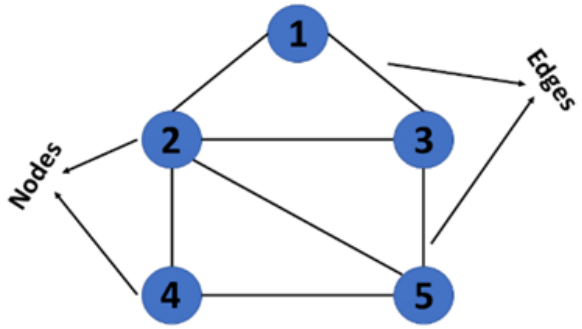
- 방향 그래프(Directed Graph) : 간선에 방향이 존재하며, A → B는 B → A와 다르다.
- 무방향 그래프(Undirected Graph) : 간선에 방향이 없으며, A — B는 B — A와 동일한 연결을 의미한다.

실생활 예

- 무방향 그래프 : 친구 관계, 도로 지도 (양방향 통행)
- 방향 그래프 : 트위터 팔로우, 웹 페이지 링크, 교통 흐름 (일방통행 포함)

용어 설명

- 노드(Node, 정점) : 데이터를 나타내는 점
- 엣지(Edge, 간선) : 노드 간의 연결 또는 관계
- 가중치(Weight) : 엣지에 숫자 값이 추가된 경우 (예: 거리, 비용 등)



무방향 그래프(Undirected Graph)이다.

🟤 Nodes (또는 Vertices, 정점) : 파란 원 안의 숫자들: 1, 2, 3, 4, 5

각 노드는 하나의 데이터 또는 개체를 나타낸다.

⇔ Edges (간선) : 노드 간을 연결하는 선들이 edge(간선)이다.

간선은 두 노드 사이의 관계 또는 연결을 의미한다.

예: (1,2), (1,3), (2,3), (2,4), (3,5), (4,5) 등

* 방향 없음 (Undirected)

모든 간선은 방향 화살표 없이 양방향으로 연결되어 있다.

즉, 노드 2에서 노드 3으로 갈 수 있다면, 반대로도 갈 수 있다는 의미이다.

<-- 이 그래프의 특징

요소	개수	설명
노드(Node)	5개	1, 2, 3, 4, 5
간선(Edge)	6개	1-2, 1-3, 2-3, 2-4, 3-5, 4-5
방향성	없음	무방향 그래프

장점

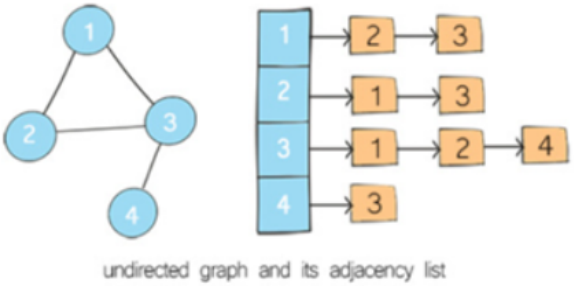
- 복잡한 관계나 구조를 표현하는 데 매우 유용하다.
  예: 소셜 네트워크, 지도, 추천 시스템 등에서 객체 간의 연결 관계를 자연스럽게 모델링 가능
- 노드(정점)와 간선(엣지)의 추가 및 삭제가 비교적 유연하게 이루어질 수 있다.
  → 구조 변경이 자유롭기 때문에 동적인 데이터 모델링에 적합하다.

단점

- 구조가 복잡하여 직접 구현하거나 다루는 것이 어렵다.
  특히 탐색(DFS, BFS), 최단경로(Dijkstra) 등 그래프 알고리즘의 이해가 쉽지 않을 수 있다.
- 대규모 그래프에서는 노드와 간선이 많아질수록 메모리 사용량이 급격히 증가할 수 있다.
  → 예: SNS의 수억 개의 사용자와 연결 정보

시간 복잡도

- 그래프의 시간 복잡도는 그래프를 표현하는 방법에 따라 달라진다.
- 그래프를 표현하는 두 가지 일반적인 방식은 인접 리스트(Adjacency List)와 인접 행렬(Adjacency Matrix) 이다.



undirected graph and its adjacency list

인접 리스트 (Adjacency List)

: 각 노드를 key로 삼고, 그 노드와 연결된 이웃 노드들을 리스트로 저장하는 방식.

- 왼쪽: 정점(Node)들이 간선(Edge)으로 연결된 무방향 그래프
- 오른쪽: 이 그래프를 인접 리스트로 나타낸 것

📝 예시 해석

노드 1 → [2, 3]

노드 2 → [1, 4]

노드 3 → [1, 4]

노드 4 → [2, 3]

* 특징 : 공간 효율이 좋음 (특히 간선이 적은 희소 그래프에 적합)

노드 간 연결 관계를 빠르게 조회 가능

Graph Representation of Undirected graph to Adjacency Matrix

인접 행렬 (Adjacency Matrix)이란?

: 행과 열이 노드 번호를 나타냄. 노드 i와 노드 j가 연결되어 있으면 1, 없으면 0

- 왼쪽: 정점 0, 1, 2로 이루어진 무방향 그래프

- 오른쪽: 이를 2차원 배열(행렬)로 표현한 인접 행렬

예시 해석

	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

→ 예: `0-1`, `0-2`, `1-2` 모두 연결되어 있음

*** 특징 : 빠른 연결 여부 확인 (i, j 인덱스 조회 → O(1))**

노드 수가 많을 경우 공간 비효율 (간선이 적어도 전체 NxN 공간 필요)

두 방식 비교 요약

항목	인접 리스트	인접 행렬
공간 효율성	좋음 (간선 수만큼)	나쁨 (노드 수²)
연결 확인 속도	느림 (리스트 순회)	빠름 (즉시 조회)
삽입/삭제	유연	느림
적합한 경우	**희소 그래프** (Sparse)	**밀집 그래프** (Dense)

각각의 시간 복잡도는 다음과 같다.

1. 인접 리스트

   탐색 : O(N+E) - 모든 노드(N)와 엣지(E)를 방문할 수 있음.

   추가 : O(1)

   삭제 : 노드의 경우는 O(N+E), 엣지의 경우는 O(E)

2. 인접 행렬

   탐색 : O(N^2) - 모든 가능한 노드 쌍을 확인해야 함.

   추가 : 노드 추가의 경우는 O(N^2), 엣지 추가의 경우는 O(1)

   삭제 : 노드 삭제의 경우는 O(N^2), 엣지 삭제의 경우는 O(1)

사용 사례 : 소셜 네트워크 (유저 간의 친구 관계 및 추천 시스템),

지도의 경로 찾기 및 거리 계산, 네트워크 라우팅 등에 활용됨.

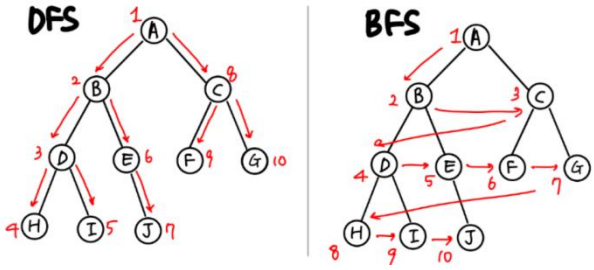
*** Python으로 코드 구현 예 ---**

```
# 1) 인접 리스트로 그래프 구현 (Python 딕셔너리)
# 이 방식은 간선이 적은 그래프(희소 그래프)에서 특히 효율적.
graph = {
    1: [2, 3],
    2: [1, 4],
    3: [1, 4],
    4: [2, 3]
}
```

```
# 연결 관계 출력
for node in graph:
    print(f"노드 {node} → {graph[node]}")

# 연결 여부 확인 함수
def is_connected(graph, node1, node2):
    return node2 in graph.get(node1, [])

# 예
print("노드 1과 3 연결됨?", is_connected(graph, 1, 3))  # True
print("노드 2와 3 연결됨?", is_connected(graph, 2, 3))  # False

# DFS(깊이 우선 탐색)로 모든 노드 방문
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# 실행
print("DFS 순회 결과:")
dfs(graph, 1)  # 1 → 2 → 4 → 3 (또는 다른 순서도 가능)
print('₩n--------------------')

# 2) 인접 행렬로 그래프 표현
# 노드 번호: 1~4 (→ 인덱스는 0~3)
# 인접 행렬 생성: 4x4 2차원 리스트 (0으로 초기화)
size = 4
adj_matrix = [[0] * size for _ in range(size)]

# 간선 추가 (무방향이므로 양쪽 다 설정)
edges = [
    (1, 2),
    (1, 3),
    (2, 4),
    (3, 4)
]
```

```
for a, b in edges:
    adj_matrix[a-1][b-1] = 1  # 인덱스 보정
    adj_matrix[b-1][a-1] = 1  # 무방향이므로 대칭

# 출력
print("인접 행렬:")
for row in adj_matrix:
    print(row)

# 출력 결과
# [0, 1, 1, 0]
# [1, 0, 0, 1]
# [1, 0, 0, 1]
# [0, 1, 1, 0]
# → 이 행렬은 다음을 의미한다.
#   adj_matrix[0][1] = 1 → 노드 1과 2 연결
#   adj_matrix[1][3] = 1 → 노드 2와 4 연결 (대각선 기준 대칭이므로 무방향 그래프임을 반영)

def is_connected(matrix, a, b): # 연결 여부 확인
    return matrix[a-1][b-1] == 1

print("노드 1과 2 연결됨?", is_connected(adj_matrix, 1, 2))  # True
print("노드 2와 3 연결됨?", is_connected(adj_matrix, 2, 3))  # False
```

**** BFS(너비 우선 탐색)과 DFS(깊이 우선 탐색)에 대한 개념, 차이점, 작동 방식, 사용 사례 등을 설명 ****



그림과 같이 그래프 형태로 데이터를 정렬했을 때 깊이 탐색을 우선하는 게 DFS, 너비 탐색을 우선하는 게 BFS이다. 탐색 방향과 그로 인해 사용하는 자료 구조가 조금 다를 뿐 모든 노드를 다 탐색한다는 것은 같기 때문에 DFS로 풀리는 문제는 보통 BFS로도 풀린다.

DFS와 BFS의 비교

항목	DFS (깊이 우선 탐색)	BFS (너비 우선 탐색)
탐색 방향	한 방향으로 깊게	가까운 노드부터 넓게
자료구조	스택(Stack) or 재귀	큐(Queue)
구현 난이도	간단 (재귀로 구현 쉬움)	큐 사용 (조금 더 복잡)
주요 사용 예	백트래킹, 경로 전체 탐색	최단 거리, 레벨 기반 탐색
경로 보장	최단 경로 아님	최단 경로 보장 (가중치 없음)

```
그래프 구조:
1 - 2
|   |
3 - 4

DFS(1부터 시작): 1 → 2 → 4 → 3
BFS(1부터 시작): 1 → 2 → 3 → 4
```

1. DFS (Depth-First Search, 깊이 우선 탐색)

정의 : DFS는 한 방향으로 계속 깊이 들어가며 탐색하다가 더 이상 갈 곳이 없으면 이전 단계로 되돌아가 다른 방향을 탐색하는 방식 → 재귀 함수 또는 스택으로 구현할 수 있다.

* 작동 방식 예 : 정점 1부터 시작 → 1 → 2 → 4 → (더 이상 없으므로 백트래킹) → 3
* 특징
  - 우선 깊이 탐색, 나중에 옆 노드 탐색
  - 재귀 구조로 구현하기 쉬움
  - 경로를 추적하거나 백트래킹(되돌아가기) 문제에 적합
* 사용 예 : 미로 찾기, 퍼즐 해결 (예: 스도쿠), 백트래킹 문제 (n-퀸, 조합, 부분 집합 등)

2. BFS (Breadth-First Search, 너비 우선 탐색)

정의 : BFS는 시작 노드와 인접한 모든 노드를 먼저 탐색한 후, 그 다음 레벨의 노드들을 탐색하는 방식.
→ 큐(Queue)를 사용해서 구현한다.

* 작동 방식 예 : 정점 1부터 시작 → 1 → 2, 3 → 4
* 특징
  - 레벨 순서로 탐색
  - 최단 거리 탐색 문제에 적합
  - 큐(Queue) 구조 활용
* 사용 예 : 최단 경로 찾기 (지도로 경로 탐색), 소셜 네트워크 추천, 트리/그래프에서 레벨 순회

*** Python으로 코드 구현 예 ---**

```
# 인접 행렬 준비 (노드 1~4, 무방향 그래프)
n = 4  # 노드 수
adj_matrix = [    # 인접 행렬 초기화 (4x4)
    [0, 1, 1, 0],   # Node 1
    [1, 0, 0, 1],   # Node 2
    [1, 0, 0, 1],   # Node 3
    [0, 1, 1, 0]    # Node 4
]

# 1. DFS (Depth-First Search)
def dfs(matrix, visited, node):
    visited[node] = True
    print(node + 1, end=' ')  # 노드 번호 출력 시 +1 (1부터 시작처럼 보이기 위해)
    for i in range(len(matrix[node])):
        if matrix[node][i] == 1 and not visited[i]:
            dfs(matrix, visited, i)

# 실행
visited = [False] * n
print("DFS 순회:")
dfs(adj_matrix, visited, 0)  # 노드 1부터 시작
```

```
print('----------------------')
# 2. BFS (Breadth-First Search)
from collections import deque

def bfs(matrix, start):
    visited = [False] * len(matrix)
    queue = deque()
    queue.append(start)
    visited[start] = True

    while queue:
        node = queue.popleft()
        print(node + 1, end=' ')

        for i in range(len(matrix[node])):
            if matrix[node][i] == 1 and not visited[i]:
                queue.append(i)
                visited[i] = True

# 실행
print("₩nBFS 순회:")
bfs(adj_matrix, 0)  # 노드 1부터 시작

# 실행 결과
# DFS 순회: 1 2 4 3
# BFS 순회: 1 2 3 4
# 참고: DFS는 깊이 우선, BFS는 너비 우선이므로 순회 순서가 다르다.
# (DFS는 한 갈래로 깊게 탐색, BFS는 가까운 노드부터 탐색)
```

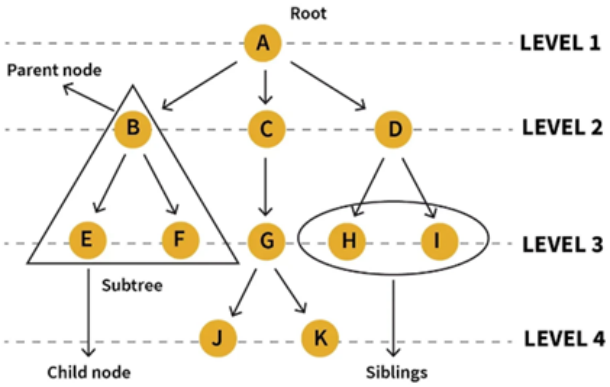
6. 트리(Tree)

Node들이 Edge로 연결되어 있는 계층적인 자료구조다.

- 반드시 하나의 Root Node를 가지고 있으며, Root Node에서 시작하여 Child Node로 분기되며 확장된다.
- 각 노드는 0개 이상의 자식 노드를 가질 수 있다. 자식이 없는 노드는 리프 노드(Leaf Node)라고 한다.
- 트리는 사이클(순환)이 없는 구조로, 하나의 부모 노드에서 출발한 경로가 다시 자기 자신으로 되돌아올 수 없다.

* 핵심 용어 정리

- 루트 노드 : 트리의 시작점이 되는 노드
- 부모 노드 : 자식 노드를 가지는 노드
- 자식 노드 : 부모 노드로부터 연결된 하위 노드
- 리프 노드 : 자식이 없는 노드 (끝 노드)
- 서브 트리 : 특정 노드를 루트로 하는 부분 트리
- 높이(Height) : 루트에서 가장 깊은 노드까지의 거리



*** 트리 구조의 특성 요약 ***

개념	설명
루트 노드	트리의 시작 노드 (A)
부모/자식 관계	위→아래 방향 연결
서브트리	어떤 노드를 루트로 하는 하위 트리
리프 노드	자식이 없는 노드
레벨(Level)	루트에서의 거리, 깊이

트리(Tree)의 구성 요소

🌳 전체 구조 : 트리는 계층적인(위계적인) 자료구조다.
노드들이 부모-자식 관계로 연결되며, 위쪽에서 아래쪽으로 뻗어나간다.

● A: Root Node (루트 노드)
- 트리의 가장 꼭대기 노드, 트리는 하나의 루트 노드를 기준으로 시작함
- 그림에서는 A가 루트 노드

● Parent & Child Nodes (부모/자식 노드)
- 부모 노드: 자식 노드를 가지는 노드 → 예: B는 E와 F의 부모
- 자식 노드: 어떤 부모로부터 내려온 노드 → 예: E는 B의 자식

● Level (레벨) : 트리의 깊이(계층)를 의미
예: A: Level 1
B, C, D: Level 2
E, F, G, H, I: Level 3
J, K: Level 4

● Subtree (서브트리) : 트리의 일부 구조를 따로 떼어낸 것
예: B를 루트로 하는 하위 구조인 {B, E, F}는 하나의 서브트리

● Siblings (형제 노드) : 같은 부모를 가지는 노드들
예: H와 I는 모두 G의 자식 → 형제 관계 (Siblings)

● Leaf Node (리프 노드) : 자식이 없는 노드
예: E, F, J, K, H, I

장점

- 트리 구조는 계층적인 관계를 표현하는데 적합하다.
- 이진 검색 트리(BST)와 같은 트리 구조는 데이터 검색, 삽입, 삭제에 효율적이다.
- 균형 잡힌 트리 구조는 노드 접근 시간을 최소화한다.

단점

- 구현하기 복잡할 수 있다.
- 각 노드는 자식 노드에 대한 참조를 저장해야 하므로 추가적인 메모리 사용이 발생한다.

시간복잡도

- 탐색 : 평균 O(log n), 최악 O(n) (이진 검색 트리에서 편향된 트리의 경우)
- 삽입 및 삭제 : 평균 O(log n), 최악 O(n)

사용 사례 : 파일과 디렉토리의 계층적 구조를 표현할 때, 데이터베이스에서 데이터의 효율적인 검색, 삽입, 삭제를 구현할 때 (이진 검색 트리 등을 사용), UI구조, 라우팅 알고리즘 등에서 유용하게 사용된다.

*** Python으로 코드 구현 예 ---**

```
# (DFS는 한 갈래로 깊게 탐색, BFS는 가까운 노드부터 탐색) class TreeNode:
class TreeNode:
    def __init__(self, data):
        self.data = data                # 노드 값 (예: 'A')
        self.children = []              # 자식 노드 목록

    def add_child(self, child_node):
        self.children.append(child_node)

# 노드 생성
A = TreeNode('A')  # Root
B = TreeNode('B');  C = TreeNode('C');  D = TreeNode('D');
E = TreeNode('E');  F = TreeNode('F');  G = TreeNode('G');
H = TreeNode('H');  I = TreeNode('I');  J = TreeNode('J');  K = TreeNode('K')

# 트리 구조 연결
A.add_child(B);  A.add_child(C);  A.add_child(D);
B.add_child(E);  B.add_child(F);  C.add_child(G);
G.add_child(J);  G.add_child(K);
D.add_child(H);  D.add_child(I);

# 트리 출력 함수 (DFS 방식)
def print_tree(node, level=0):
    print('  ' * level + f'- {node.data}')
    for child in node.children:
        print_tree(child, level + 1)

# 실행
print("트리 구조:")
print_tree(A)
```

출력 결과 :

```
- A
  - B
    - E
    - F
  - C
    - G
      - J
      - K
  - D
    - H
    - I
```

7. 힙(Heap)

모든 노드가 특정한 순서를 유지하며 구성된 완전 이진 트리 형태의 자료구조다.

힙은 두 가지 주요한 특성을 가진다.

1) 구조적 특성 (Structural Property) : 항상 완전 이진 트리(Complete Binary Tree)의 형태를 유지한다.
즉, 왼쪽에서 오른쪽으로 차례대로 채워지며, 마지막 레벨을 제외하고는 노드가 모두 채워진 형태다.

2) 순서적 특성 (Heap Order Property) : 각 부모 노드와 자식 노드 사이의 값의 크기 관계에 따라 두 가지 유형이 존재한다.

~ 최대 힙 (Max Heap)
- 부모 노드 ≥ 자식 노드
- 루트에 가장 큰 값이 위치
- 예: 우선순위가 높은 작업을 먼저 처리해야 할 때 사용

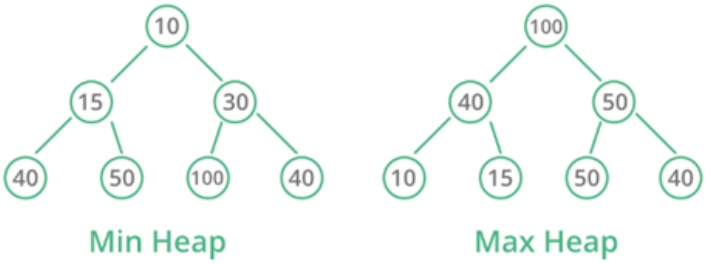
~ 최소 힙 (Min Heap)
- 부모 노드 ≤ 자식 노드
- 루트에 가장 작은 값이 위치
- 예: 가장 작은 값부터 처리할 필요가 있을 때 사용 (예: 다익스트라 알고리즘)

Max Heap 예:

```
        50
      /    \
    30      40
   /  \    /
  10  20  35
```

Min Heap 예:

```
        10
      /    \
    20      30
   /  \    /
  50  40  35
```



Heap Data Structure

📌 **좌측: Min Heap (최소 힙)**
- **부모 노드 ≤ 자식 노드 관계를 만족한다.**
- **가장 작은 값이 루트(최상단)에 위치한다.**
- **모든 부모 노드(10, 15, 30)가 자식 노드보다 작거나 같음을 확인할 수 있다.**

📌 **우측: Max Heap (최대 힙)**
- **부모 노드 ≥ 자식 노드 관계를 만족한다.**
- **가장 큰 값이 루트에 위치한다.**

힙의 두 가지 핵심 특성

특성 구분	설명
구조적 특성	완전 이진 트리 (왼쪽부터 빈칸 없이 채움)
순서적 특성	부모-자식 간 값의 대소관계 유지 (Min 또는 Max)

힙 사용 예

힙 종류	특징	사용 예시
Min Heap	가장 작은 값이 루트	다익스트라 알고리즘, 작업 우선순위
Max Heap	가장 큰 값이 루트	최대값 추출, 우선순위 큐

장점
- 최대 힙에서는 최대값을, 최소 힙에서는 최소값을 O(1)에 접근할 수 있어 최대값과 최소값에 효율적으로 접근.
- 중간 값을 효율적으로 찾고, 값을 효율적으로 정렬할 수 있다.
- 동적 데이터 관리에 효율적이다. 데이터가 추가/제거 되어도 힙은 빠르게 재구성할 수 있다.

단점
- 순차적 접근이 불리하다.
- 임의의 노드를 삭제하거나 검색하는데 비효율적이다.

시간복잡도
- 힙 생성 : O(n)
- 삽입 : O(log n)
- 삭제 : 최대값 또는 최소값 삭제는 O(log n), 임의의 값 삭제는 O(n)
- 최대값 또는 최소값 접근 : O(1)

사용 사례 : 우선순위 큐(Priority Queue), 최댓값/최솟값 빠르게 찾기, 힙 정렬(Heap Sort),
다익스트라 알고리즘 (최단 경로 탐색)

*** Python으로 코드 구현 예 ---**

```
# Python의 내장 모듈 heapq는 Min Heap만 제공하므로, Max Heap은 약간의 트릭으로 구현한다.

# 1. Min Heap 구현 (heapq 사용)
import heapq

# 빈 Min Heap 생성
min_heap = []

# 원소 추가 (heapq는 자동으로 최소 힙 유지)
heapq.heappush(min_heap, 10)
heapq.heappush(min_heap, 15)
heapq.heappush(min_heap, 30)
heapq.heappush(min_heap, 40)
heapq.heappush(min_heap, 50)
heapq.heappush(min_heap, 100)

print("Min Heap 내부 구조:", min_heap)

# 가장 작은 값 꺼내기
print("heappop():", heapq.heappop(min_heap))   # 10
print("그 다음 최소값:", heapq.heappop(min_heap))   # 15

# 출력 결과
# Min Heap 내부 구조: [10, 15, 30, 40, 50, 100]
# heappop(): 10
# 그 다음 최소값: 15
```

```
# 2. Max Heap 구현 (heapq + 음수 활용)
# Python에는 Max Heap이 내장되어 있지 않기 때문에, 값을 음수로 바꿔서 Min Heap처럼 사용
# Max Heap처럼 작동시키기 위해 음수로 저장
max_heap = []

heapq.heappush(max_heap, -100)
heapq.heappush(max_heap, -40)
heapq.heappush(max_heap, -70)
heapq.heappush(max_heap, -10)
heapq.heappush(max_heap, -15)
heapq.heappush(max_heap, -50)

# 출력 시 다시 -1 곱해서 원래 값으로 되돌림
print("Max Heap 내부 구조 (원래 값 기준):", [-x for x in max_heap])

# 가장 큰 값 꺼내기
print("heappop():", -heapq.heappop(max_heap))          # 100
print("그 다음 최대값:", -heapq.heappop(max_heap))   # 70

# 출력 결과
# Max Heap 내부 구조 (원래 값 기준): [100, 40, 70, 10, 15, 50]
# heappop(): 100
# 그 다음 최대값: 70

print('₩n**MaxHeap 클래스로 직접 구현한 코드**')
class MaxHeap:
    def __init__(self):
        self.heap = []

    def push(self, val):
        self.heap.append(val)
        self._heapify_up()

    def pop(self):
        if not self.heap:
            return None
        self._swap(0, -1)
        max_val = self.heap.pop()
        self._heapify_down()
        return max_val

    def _heapify_up(self):
        idx = len(self.heap) - 1
```

```
        while idx > 0:
            parent = (idx - 1) // 2
            if self.heap[idx] > self.heap[parent]:
                self._swap(idx, parent)
                idx = parent
            else:
                break

    def _heapify_down(self):
        idx = 0
        n = len(self.heap)
        while 2 * idx + 1 < n:
            left = 2 * idx + 1
            right = 2 * idx + 2
            largest = idx

            if self.heap[left] > self.heap[largest]:
                largest = left
            if right < n and self.heap[right] > self.heap[largest]:
                largest = right
            if largest == idx:
                break
            self._swap(idx, largest)
            idx = largest

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

# 사용 예
h = MaxHeap()
for val in [100, 40, 70, 10, 15, 50]:
    h.push(val)

print("Max Heap pop:", h.pop())   # 100
print("다음 pop:", h.pop())          # 70

# 출력 결과
# Max Heap pop: 100
# 다음 pop: 70
```