HW2_알고리즘_보고서

20기 인턴 송수민

- 목차

1. 알고리즘
1.1. 정의
특정 문제를 해결하기 위해 순서대로 정의된 유한한 절차나 규칙의 모음
원하는 출력을 만들어 내는 과정을 기술한 것
컴퓨터 과학에서는 입력값을 받아 원하는 출력을 생성하는 명확한 단계를 기술하는 것

1.2. 특징
- 입력 : 알고리즘은 0또는 그 이상의 외부에서 제공된 자료가 존재해야 함
- 출력 : 알고리즘은 최소 1개 이상의 결과를 가져야 함

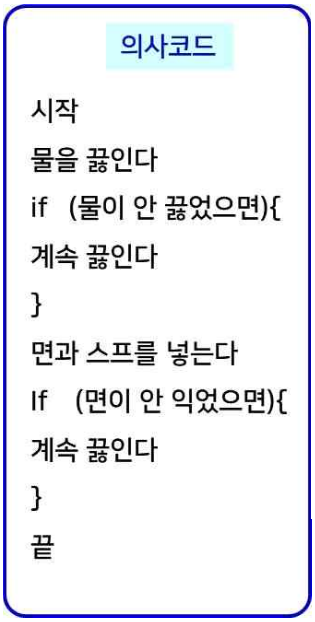
- 명확성 : 알고리즘의 각 단계는 명확하여 애매함이 없어야 함 ex) 1. 좋은 예시 문제: 1부터 10까지의 합을 구하라. 알고리즘(명확한 표현): 1) 변수 sum을 0으로 초기화한다. 2) 변수 i를 1로 초기화한다. 3) i가 10 이하일 동안 다음을 반복한다: 4) sum ← sum + i 5) i ← i + 1 6) sum을 출력한다. -> 여기서는 각 단계가 구체적이고 해석의 여지가 없다. 누구나 이대로 실행하면 같은 결과(55)를 얻음 2. 안 좋은 예시 문제: 1부터 10까지의 합을 구하라. 알고리즘(애매한 표현): 1) 어떤 변수에 적절한 초기값을 넣는다. 2) 1부터 10까지 반복하면서 합을 구한다. 3) 결과를 출력한다. -> 이 경우 "적절한 초기값이 뭔지?", "합을 어떻게 구하라는 건지?"가 모호함, 사람마다

다르게 해석할 수 있고, 컴퓨터는 실행할 수 없음
- 유한성 : 알고리즘은 단계들을 유한한 횟수로 거친 후 문제를 해결하고 종료해야 함, 알고리즘의 한 단계 이후 m의 값은 n보다 작으며, m!=0이면 n의 값은 다음 번 단계에서 줄어듦
- 효과성 : 알고리즘의 모든 연산들은 충분히 단순해야 함

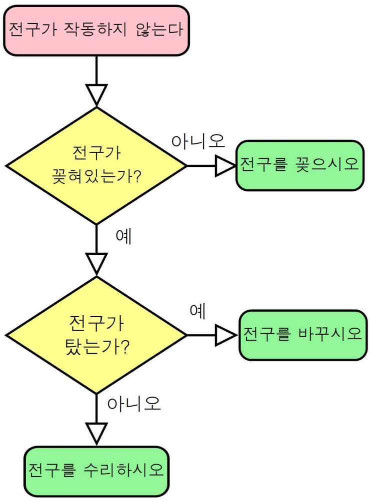
즉, 알고리즘은 어떠한 입력 존재 시, 입력에 따라 명령을 정확하게 실행하고, 효과적으로 결과물을 도출할 수 있다면 알고리즘으로 볼 수 있음, 반대로 명령에 애매함이 있다거나 유한한 시간 안에 끝내는 것이 보장되지 않은 경우를 메서드라고 함

1.3. 표현 방법

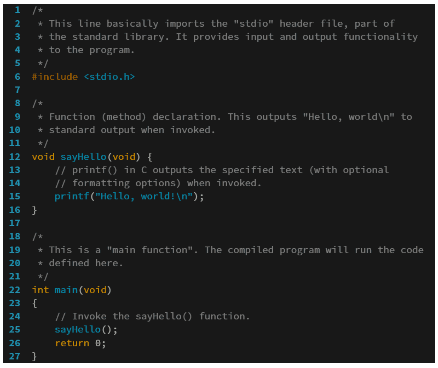
- 의사코드



- 순서도



- 프로그래밍 언어



```c
/*
 * This line basically imports the "stdio" header file, part of
 * the standard library. It provides input and output functionality
 * to the program.
 */
#include <stdio.h>

/*
 * Function (method) declaration. This outputs "Hello, world\n" to
 * standard output when invoked.
 */
void sayHello(void) {
    // printf() in C outputs the specified text (with optional
    // formatting options) when invoked.
    printf("Hello, world!\n");
}

/*
 * This is a "main function". The compiled program will run the code
 * defined here.
 */
int main(void)
{
    // Invoke the sayHello() function.
    sayHello();
    return 0;
}
```

1.4. 평가

1.4.1. 시간 복잡도

- 정의 : 알고리즘의 소요 시간은 정확히 평가할 수 없음, 그러므로 자료의 수 n이 증가할 때 시간이 증가하는 대략적인 패턴을 시간 복잡도라고 함

- 종류

- $O(1)$과 같은 상수(constant) 형태
- $O(\log_2 n)$ 과 같은 로그(logarithmic) 형태
- $O(n)$과 같은 선형(linear)
- $O(n\log n)$과 같은 선형로그(linear-logarithmic) 형태
- $O(n^c), O(n^3)$과 같은 다차(polynomial) 형태
- $O(c^n), O(3^n)$과 같은 지수(exponential) 형태
- $O(n!)$과 같은 팩토리얼(factorial) 형태

일반적으로 위로 갈수록 알고리즘이 매우 빨라지며, 아래로 갈수록 n의 값이 커지고 급격하게 알고리즘의 수행시간이 증가함

시간/n	1	2	3	4	8	16	32	64	1000
1	1	1	1	1	1	1	1	1	1
log n	0	1	1.58	2	3	4	5	6	9.97
n	1	2	3	4	8	16	32	64	1000
nlog n	0	2	4.75	8	24	64	160	384	9966

n^2	1	4	9	16	64	256	1024	4096	1000000
n^3	1	8	27	64	512	4096	32768	262144	1000000000
2^n	2	4	8	16	256	65536	4294967296	약 1.844 x 10^{19}	약 1.07 x 10^{301}
n!	1	2	6	24	40320	2092278988000	약 2.63 x 10^{35}	약 1.27 x 10^{89}	약 4.02 x 10^{2567}

n이 작을 때는 알고리즘 사이에 큰 차이가 없음. 하지만, n의 값이 커질수록 시간 복잡도가 커질수록 수행시간이 급격하게 길어지게 됨.
알고리즘을 개선해서 지수 형태의 알고리즘 코드를 로그와 같은 시간 복잡도가 작은 형태로 변경할 수 있다면, 프로그램의 엄청난 성능 향상을 기대할 수 있음.

1.4.2. 공간 복잡도

- 정의 : 알고리즘을 실행하기 위해 필요한 메모리(저장 공간)의 총량을 입력 크기 n의 함수로 표현한 것

현실에서 시간 복잡도 보다 중요도가 떨어짐.
시간이 적으면서 메모리가 지수적으로 증가하는 경우가 없기 때문.
하지만, 동적 계획법에서는 메모리가 많이 필요하기 때문에 중요함.
또한, 임베디드. 펌웨어 등 하드웨어 환경이 한정된 경우 공간 복잡도도 상당히 중요함.

2. 알고리즘의 종류
2.1 정렬 알고리즘 (Sorting)
데이터를 일정한 순서(오름차순/내림차순) 로 재배치하는 방법.

2.1.1. 선택 정렬 (Selection Sort)

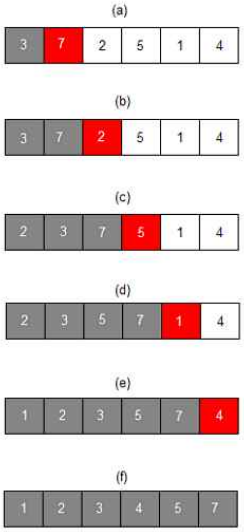
아이디어: 매번 가장 작은(큰) 원소를 선택하여 맨 앞으로 보내는 방식
시간복잡도: $O(n^2)$
특징: 구현 간단, 하지만 비효율적

2.1.2. 버블 정렬 (Bubble Sort)

아이디어: 인접한 두 원소를 비교하여 큰 값을 뒤로 이동

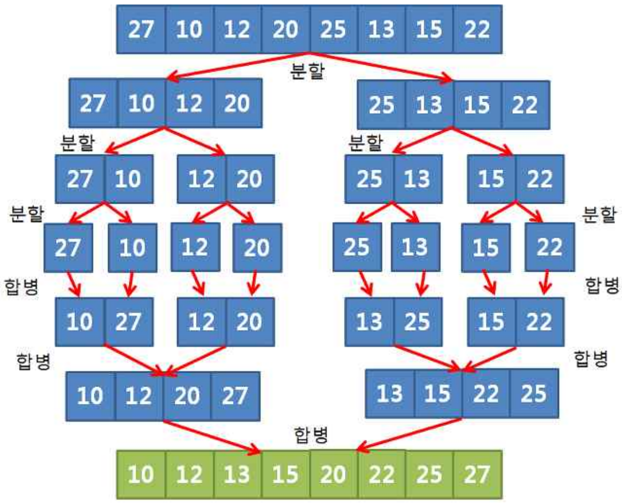
시간복잡도: $O(n^2)$
특징: 교환 횟수가 많아 비효율적, 거의 정렬된 경우 빠름

2.1.3. 삽입 정렬 (Insertion Sort)



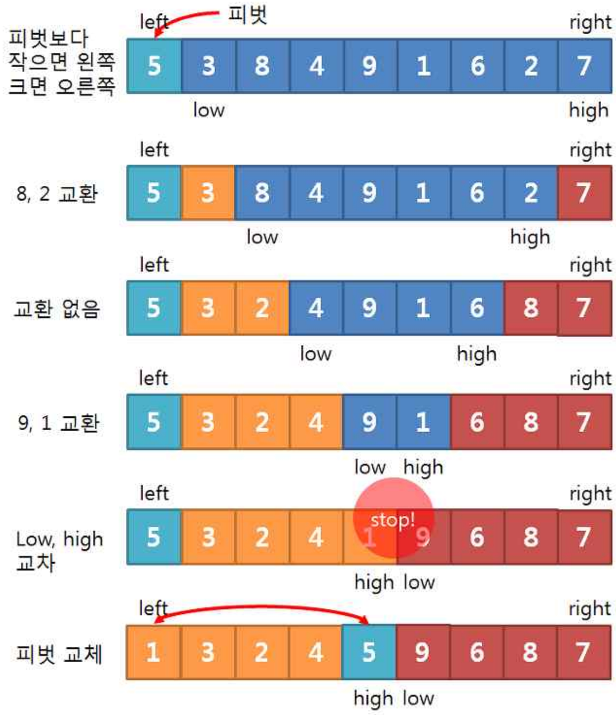
아이디어: 이미 정렬된 부분에 새로운 값을 적절한 위치에 삽입
시간복잡도: $O(n^2)$, 최선 $O(n)$ (거의 정렬된 경우 효율적)
특징: 소규모 데이터 정렬에 적합

2.1.4. 병합 정렬 (Merge Sort)



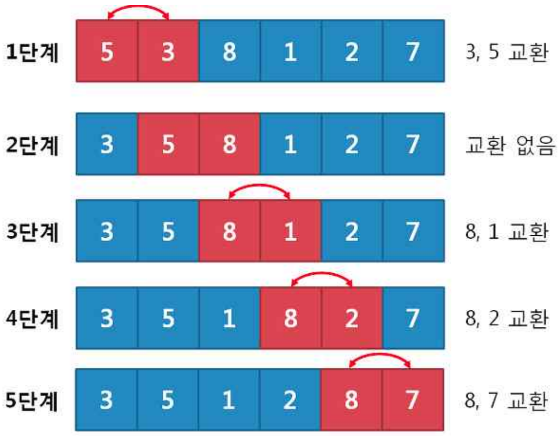
아이디어: 분할 정복(Divide & Conquer) – 배열을 반으로 나누고 정렬 후 병합
시간복잡도: $O(n \log n)$
특징: 안정 정렬, 추가 메모리 필요

2.1.5. 퀵 정렬 (Quick Sort)



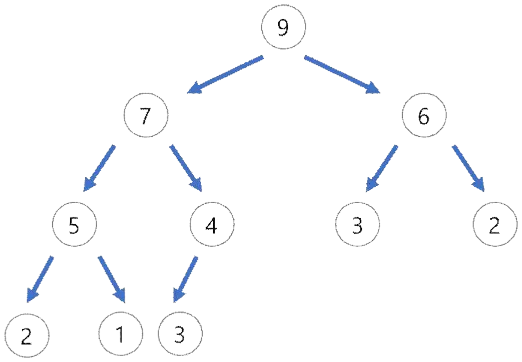
아이디어: 피벗(Pivot)을 기준으로 작은 값은 왼쪽, 큰 값은 오른쪽으로 분할
시간복잡도: 평균 O(n log n), 최악 O(n²)
특징: 보통 가장 빠른 정렬, 재귀 활용

2.1.6. 쉘 정렬 (Shell Sort)

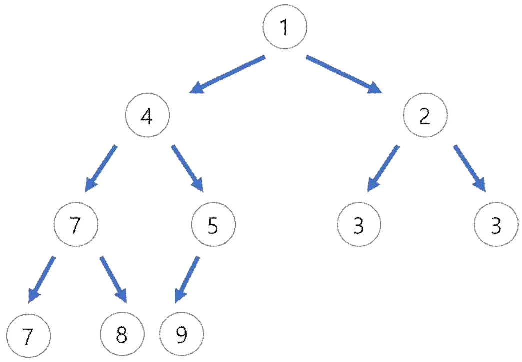


아이디어: 삽입 정렬을 보완 → 일정 간격(gap)으로 떨어진 원소끼리 정렬 후 간격을 줄여감
시간복잡도: 평균 O(n log n) 정도
특징: 삽입 정렬보다 빠름

2.1.7. 힙 정렬 (Heap Sort)



-최대 힙(max heap)-



-최소 힙(min heap)-

아이디어: 최대 힙(또는 최소 힙)을 구성해 하나씩 꺼내며 정렬

시간복잡도: O(n log n)

특징: 추가 메모리 불필요, 항상 일정한 성능

2.2. 탐색 알고리즘 (Searching)

데이터 구조에 따라 탐색 방법이 달라짐.

2.2.1. 배열 탐색

순차 탐색 (Linear Search)

처음부터 끝까지 차례대로 검색 → O(n)

이진 탐색 (Binary Search)

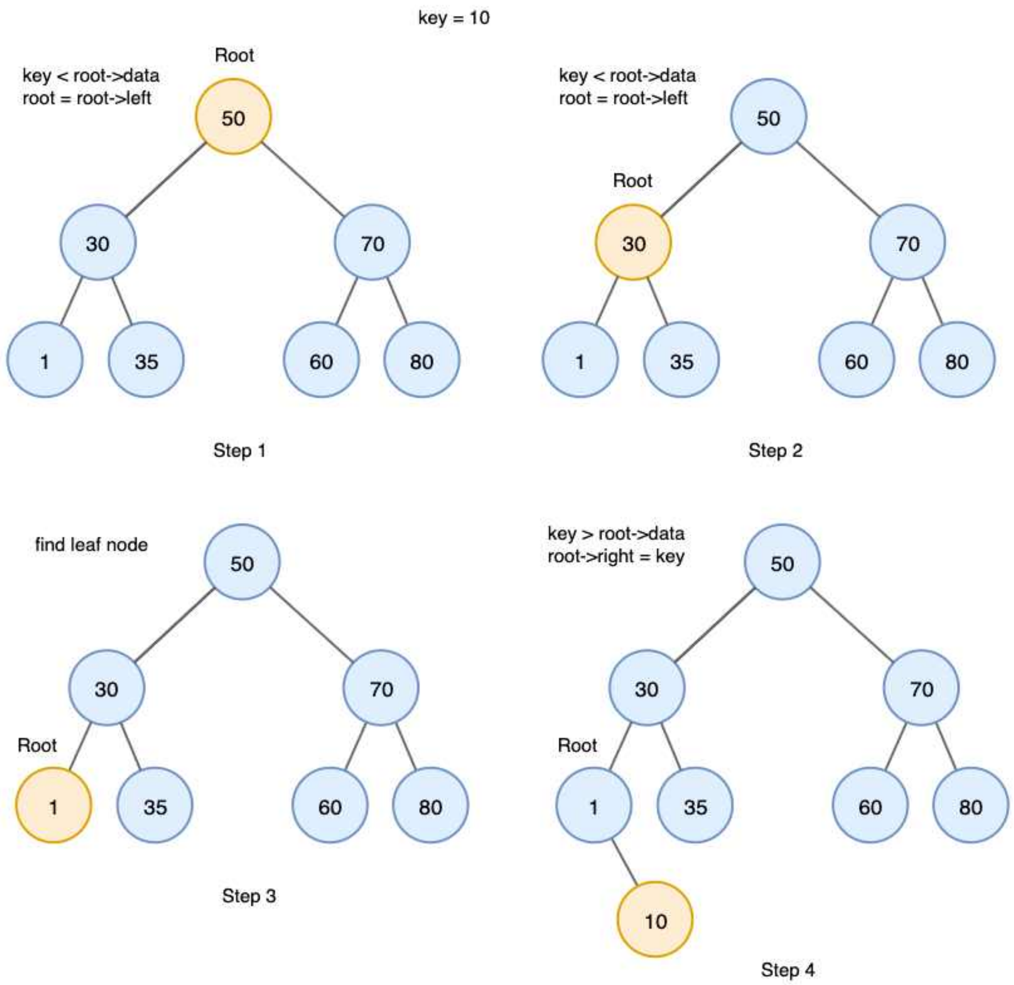
정렬된 배열에서 중앙값을 기준으로 탐색 → O(log n)

2.2.2. 행렬 탐색

단순 탐색: 행렬 전체를 순차적으로 확인 → $O(n^2)$

정렬된 행렬 탐색: 각 행/열이 정렬된 경우, 오른쪽 위/왼쪽 아래에서 시작하면 O(n) 가능

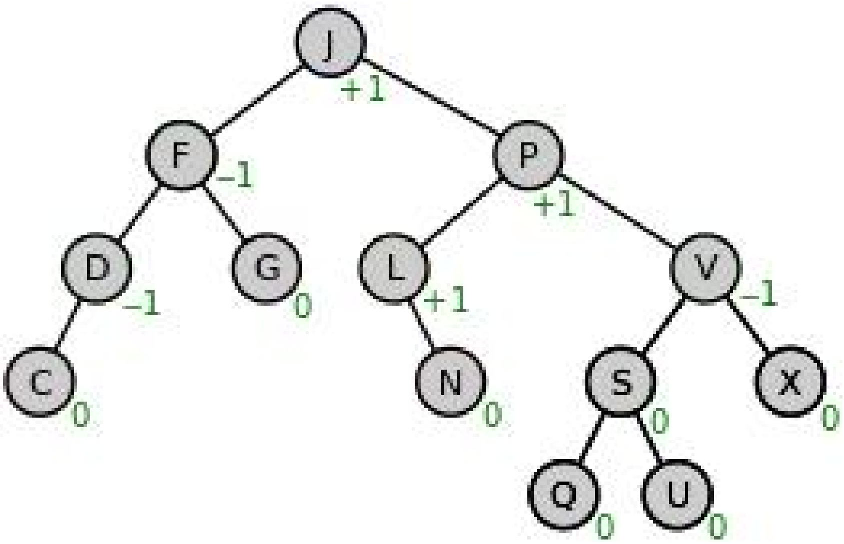
2.3. 트리



2.3.1. 이진 탐색 트리 (BST)

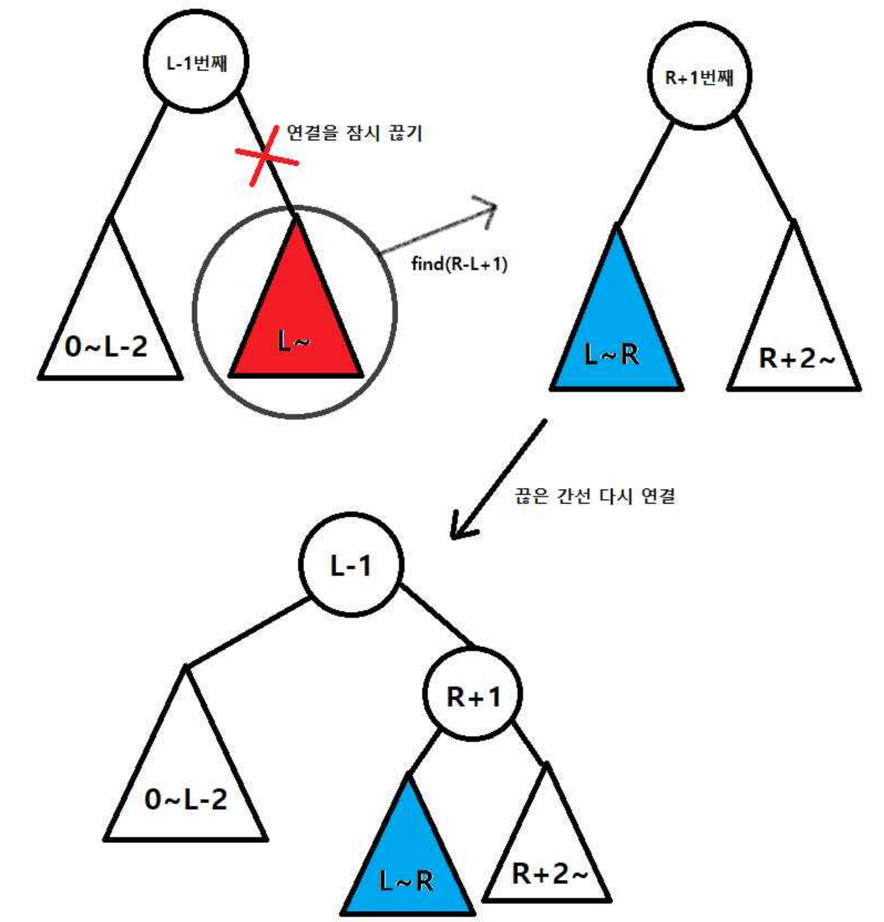
- 정의: 모든 노드에 대해 왼쪽 < 현재 < 오른쪽이 항상 성립.
- 핵심 불변식: 중위순회(Inorder)하면 항상 오름차순.
- 시간복잡도: 평균 탐색/삽입/삭제 O(log n), 최악(편향) O(n).
- 삭제 팁:
    - 자식 0개: 바로 제거
    - 자식 1개: 자식 올리기
    - 자식 2개: 후계자(successor)(오른쪽 서브트리의 최소)와 값 교환 후 제거
- 언제 쓰나: 간단 구현/메모리 절약이 필요하지만 균형 보장은 필요 없을 때.

2.3.2. AVL 트리



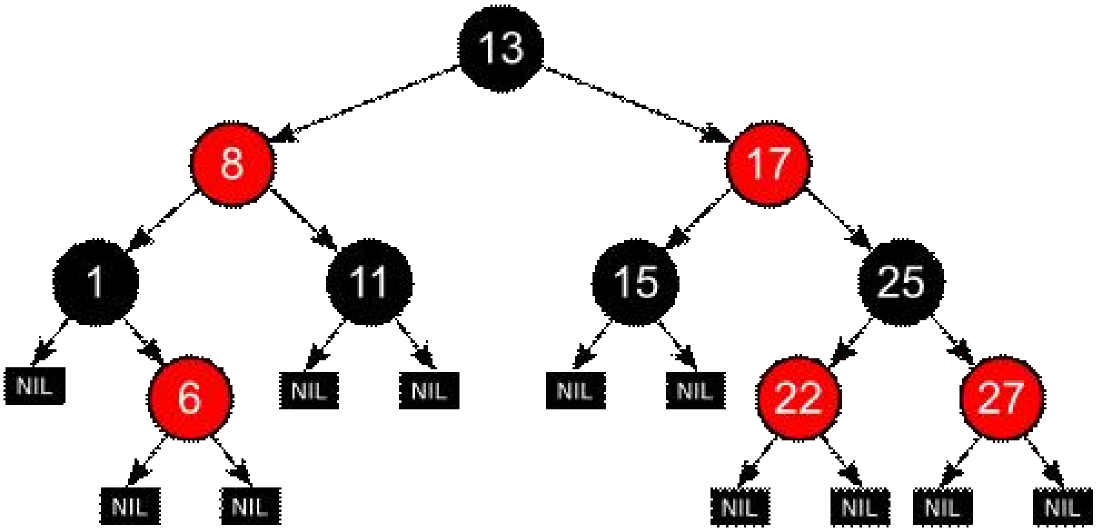
- 정의: 자기균형 BST. 각 노드의 balance = 높이(왼)-높이(오) 가 -1, 0, +1 범위.
- 시간복잡도: 탐색/삽입/삭제 모두 O(log n) (높이 엄격히 제한).
- 회전 유형:
  LL(오른쪽 회전), RR(왼쪽 회전), LR(왼→오), RL(오→왼)
- 장단점: 탐색 성능 최상. 삽입/삭제 때 회전과 높이 갱신 오버헤드가 RBT보다 큼.
- 언제 쓰나: 탐색이 특히 많은 워크로드(읽기 위주)에서 안정적 성능.

2.3.3. 스플레이 트리 (Splay Tree)



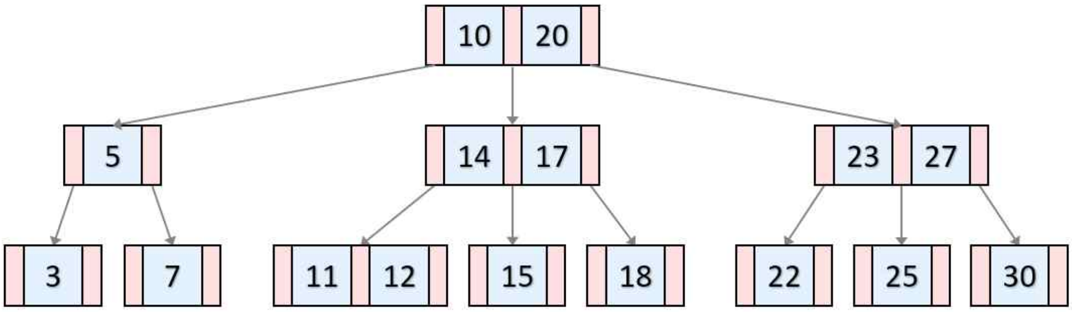
- 아이디어: 접근한 노드를 연속 회전(Zig/Zig-Zig/Zig-Zag)으로 루트로 당김(Splaying).
- 보장: 연산 암묵적 균형으로 아몰타이즈드 O(log n), 최악 O(n).
- 특징: 자주 쓰는 키가 루트 근처에 모여 지역성(Locality) 뛰어남. 추가 메타데이터 불필요.
- 언제 쓰나: 최근/자주 접근 키가 뚜렷한 패턴의 캐시/사전 구조.

2.3.4. 레드-블랙 트리 (Red-Black Tree)



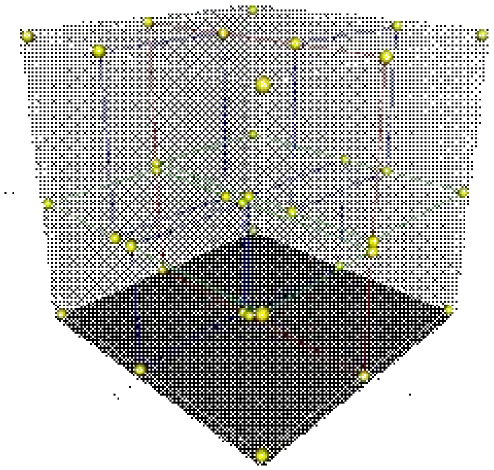
- 정의: 색 속성(빨강/검정)으로 균형 유지하는 BST.
- 불변식(요지):
  - 루트·리프(NIL)는 검정
  - 빨강 노드의 자식은 항상 검정
  - 임의 노드→리프 모든 경로의 검정 노드 수 동일(black-height)
- 복잡도: 탐색/삽입/삭제 O(log n). 회전 수가 적어 실사용 삽입/삭제가 빠름.
- 언제 쓰나: 표준 라이브러리(Map/Set), OS 스케줄러 등 일반 목적 균형 BST의 디폴트.

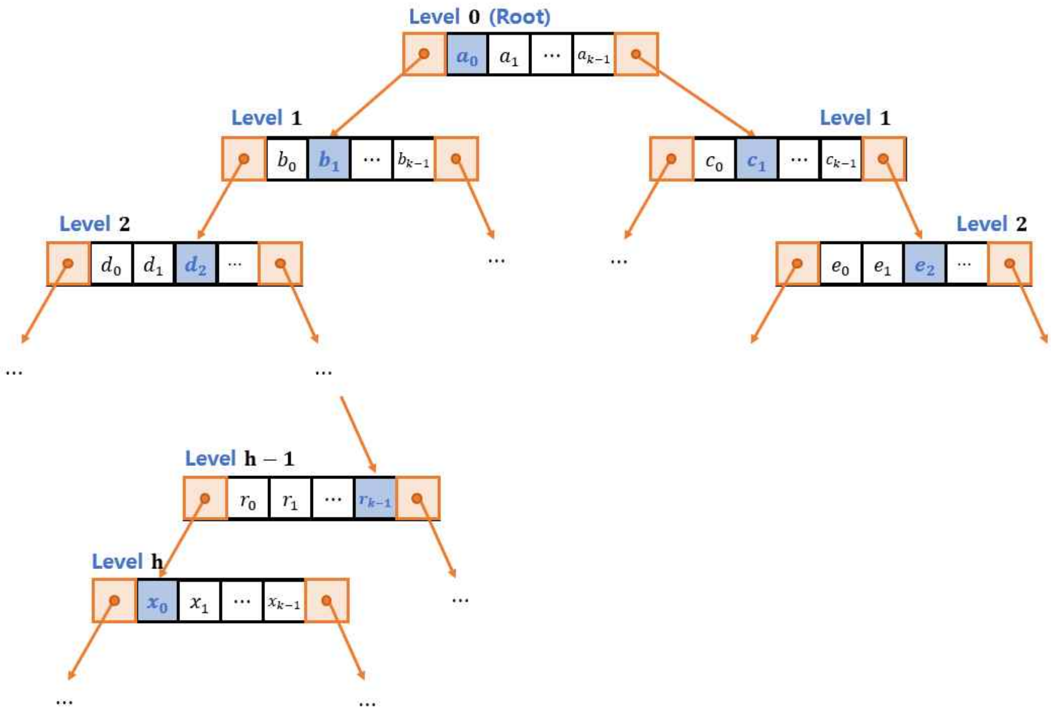
2.3.5. B-트리 (B-Tree)



- 정의: 다분기(m-ary) 균형 검색트리, 각 노드가 여러 키를 보관(디스크/SSD 친화적).
- 속성(차수 m 기준):
  - 루트 제외 내부노드는 ⌈m/2⌉−1 … m−1개의 키, ⌈m/2⌉ … m개의 자식
  - 모든 리프의 깊이 동일(완전 균형)
- 복잡도: 탐색/삽입/삭제 $O(\log_m n)$ (노드 액세스 수가 작음 → I/O 효율).
- 변형: B+트리(리프에만 데이터, 리프 간 연결 리스트로 범위 질의 최적).
- 언제 쓰나: 데이터베이스/파일시스템(페이지 단위 I/O) 표준 인덱스.

2.3.6. KD-트리 (k-dimensional tree)

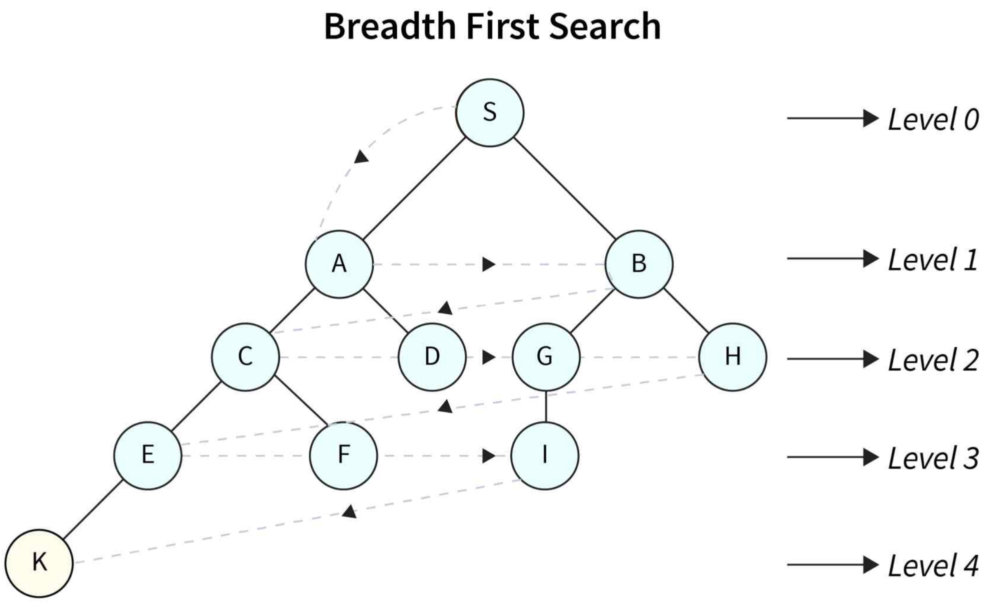




- 정의: d차원 포인트를 축 분할(axis-aligned) 로 재귀적 이진 분할.
- 구성: 깊이 depth에서 축 axis = depth % d 선택, 중앙값으로 분할(가급적 균형).
- 용도: 최근접 이웃(NN), 범위 질의(range), k-NN.
- 복잡도:
  - 빌드: 보통 O(n log n)
  - NN 평균 O(log n)(저차원에서), 최악 O(n)
- 주의: 차원의 저주—차원 수가 커지면 분할이 효과 떨어져 선형에 근접.
- 언제 쓰나: 2D/3D 지리·로보틱스·컴퓨터비전에서 공간 질의.

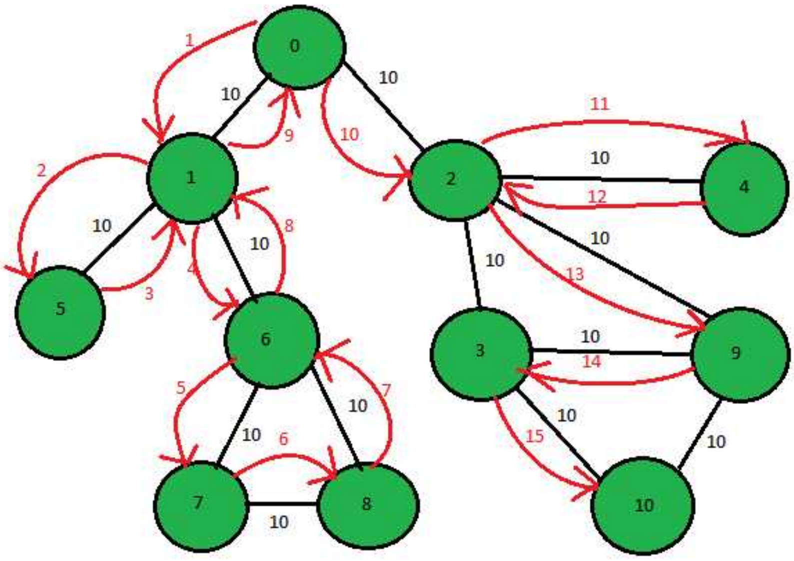
2.4. 그래프
- 공통: 표현 & 기본
- 표현:
인접 리스트: 희소 그래프에 효율적 (O(V+E) 순회).
인접 행렬: 밀집 그래프, 간선 테스트 O(1), 메모리 $O(V^2)$.
방문 배열/거리/부모는 대부분 알고리즘의 기본 상태로 관리.

2.4.1. BFS (너비 우선 탐색)



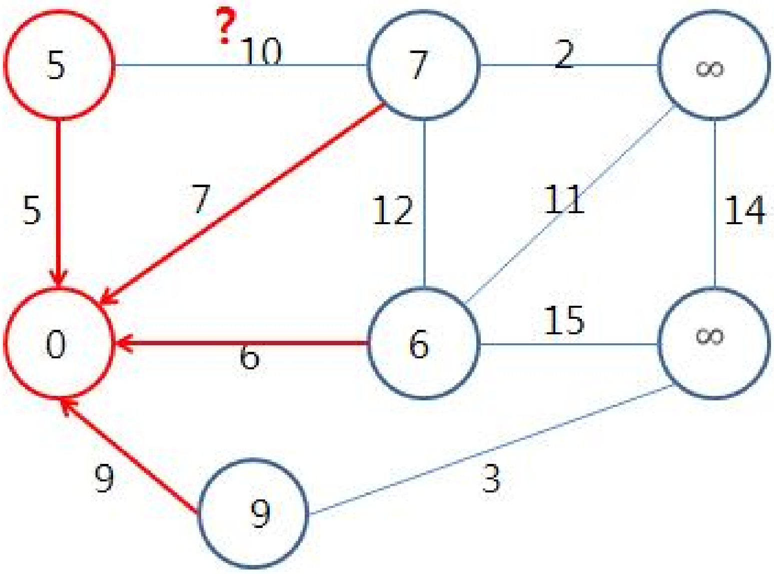
- 개념: 시작점에서 가까운 정점부터 레벨 순서로 탐색(큐).
- 복잡도: O(V+E)(인접 리스트).
- 특징: 무가중치 그래프 최단 경로. 레벨 그래프 생성.
- 활용: 최단 거리, Flood-fill, 연결성 판정, 이분 그래프 판정.

2.4.2. DFS (깊이 우선 탐색)



- 개념: 한 경로를 끝까지 내려간 뒤 백트래킹(스택/재귀).
- 복잡도: O(V+E).
- 활용: 사이클 탐지, 위상정렬(DAG), 강결합요소(SCC)(Tarjan/Kosaraju), 브리지/단절점.

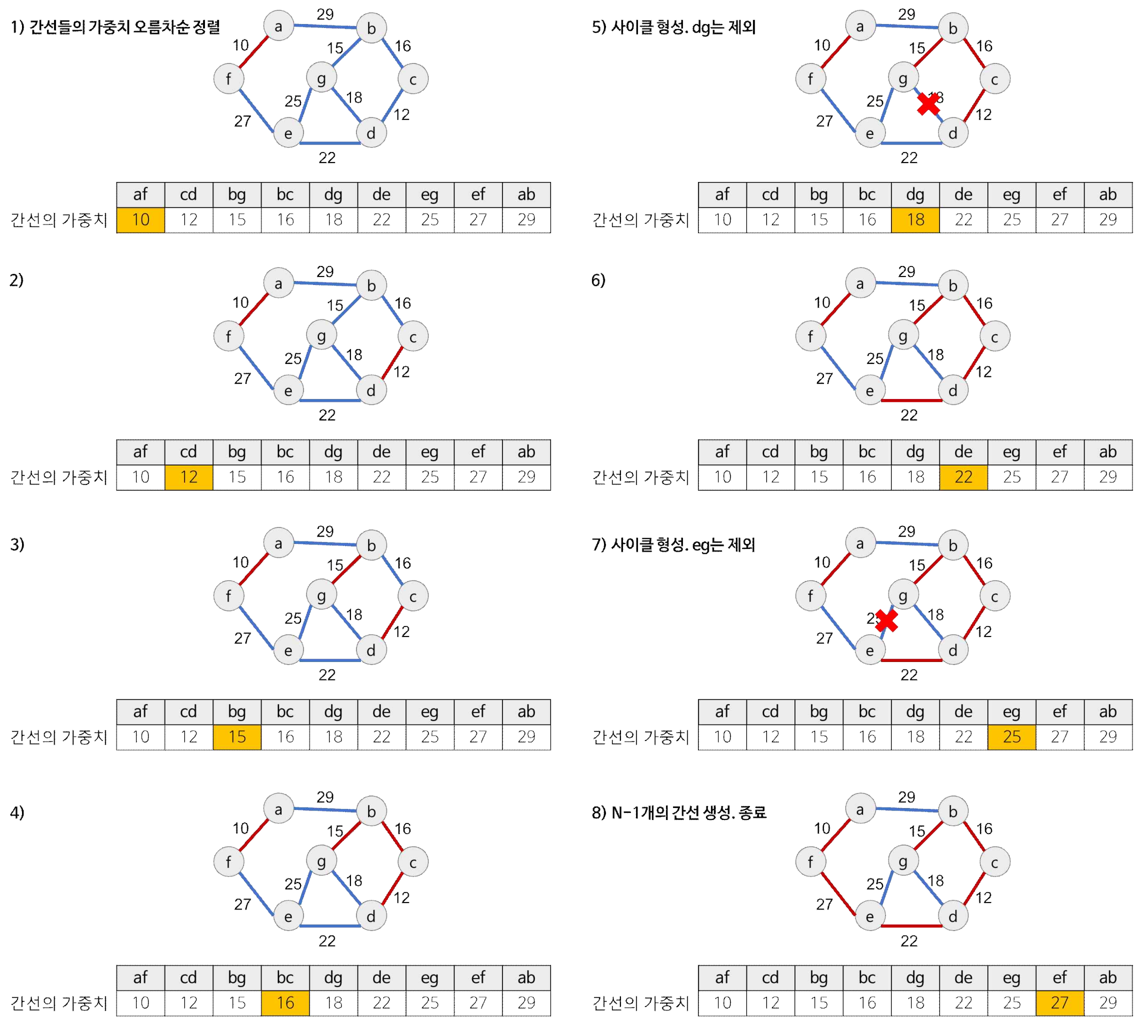
2.4.3. 프림 (Prim) - MST



- 개념: 한 정점에서 시작해 트리에 가장 저렴한 간선을 하나씩 추가(점 확장).
- 전제: 연결, 무방향, 가중치.
- 복잡도:

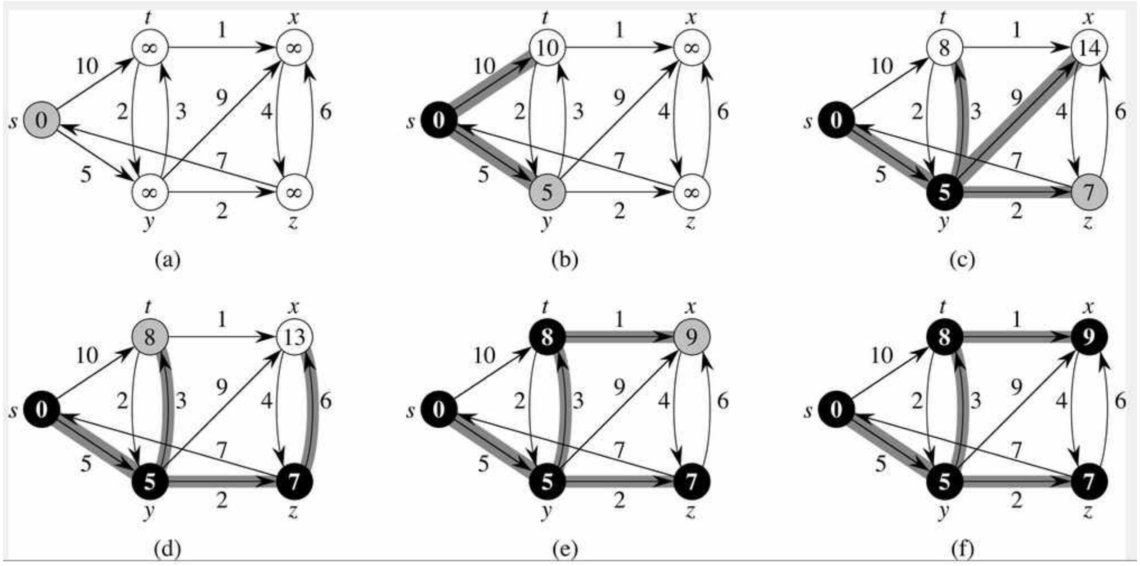
- 이진 힙 + 인접 리스트: O(E log V)
- 피보나치 힙: O(E + V log V) (이론상)
- 언제 유리?: 밀집 그래프에서 좋음(힙 기반). 시작점 무관.

2.4.4. 크루스칼 (Kruskal) – MST



- 개념: 간선을 가중치 오름차순으로 보며, 사이클이 안 생기면 채택(유니온-파인드/DSU).
- 복잡도: O(E log E) ≈ O(E log V).
- 언제 유리?: 희소 그래프에서 간선 정렬 후 DSU로 빠름.
- 팁: DSU는 경로 압축 + 랭크/사이즈로 거의 상수.

2.4.5. 다익스트라 (Dijkstra) – 단일 시작 최단경로 (비음수)



(a) (b) (c) (d) (e) (f)

- 전제: 가중치 음수 없음.
- 원리: dist가 가장 작은 정점을 우선 확정(최소힙, 그리디).
- 복잡도:
  - 이진 힙: O((V+E) log V)
  - 인접 행렬: O(V^2)
- 주의: 음수 간선 있으면 오답 가능 → 벨만-포드/Johnson 사용.
- 소스 코드

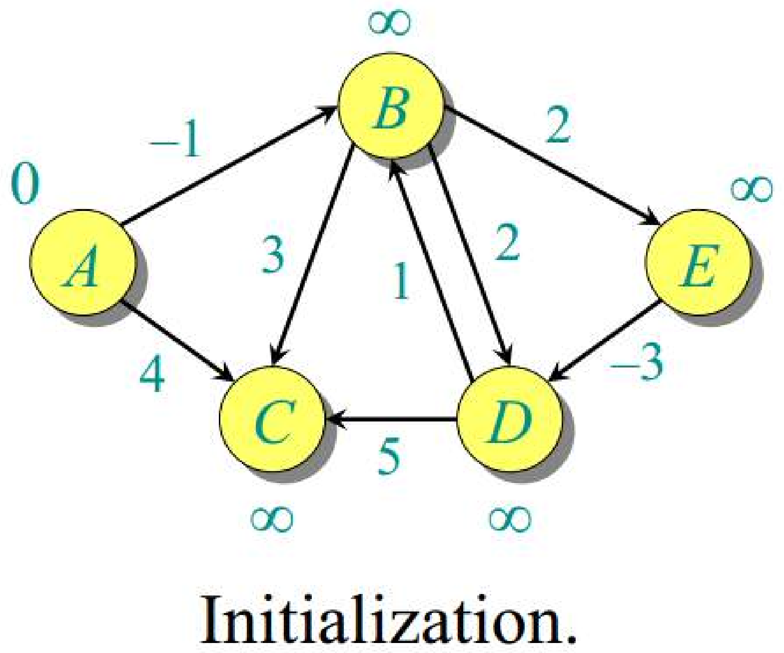
```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
#include <algorithm>

using namespace std;

enum class CellType { Empty, Block, Start, End };

struct GridModel {
    int rows, cols;
    vector<vector<CellType>> cells;
    pair<int,int> start{-1,-1}, goal{-1,-1};

    GridModel(int r,int c): rows(r), cols(c), cells(r, vector<CellType>(c, CellType::Empty)) {}
```

```
    bool inBounds(int r,int c) const { return r>=0 && r<rows && c>=0 && c<cols; }
    CellType at(int r,int c) const { return cells[r][c]; }
    void set(int r,int c, CellType t) { cells[r][c]=t; }
    int idx(int r,int c) const { return r*cols + c; }
    pair<int,int> coord(int idx) const { return {idx/cols, idx%cols}; }
};

struct DJNode{ int idx, dist; bool operator<(const DJNode& o) const { return
dist>o.dist; } };
struct SearchResult { bool found=false; vector<int> visited_order; vector<int> path; };

SearchResult Dijkstra(const GridModel& grid){
    SearchResult out;
    if(grid.start.first<0 || grid.goal.first<0) return out;

    int R=grid.rows, C=grid.cols;
    int startIdx=grid.idx(grid.start.first, grid.start.second);
    int goalIdx=grid.idx(grid.goal.first, grid.goal.second);

    vector<int> dist(R*C,numeric_limits<int>::max());
    vector<int> parent(R*C,-1);
    vector<bool> visited(R*C,false);
    priority_queue<DJNode> pq;

    dist[startIdx]=0;
    pq.push({startIdx,0});

    auto neighbors=[&](int r,int c){
        static const int dr[4]={-1,1,0,0};
        static const int dc[4]={0,0,-1,1};
        vector<pair<int,int>> nb;
        for(int k=0;k<4;k++){
            int nr=r+dr[k], nc=c+dc[k];
            if(grid.inBounds(nr,nc)          &&          grid.at(nr,nc)!=CellType::Block)
nb.push_back({nr,nc});
        }
        return nb;
    };

    while(!pq.empty()){
        auto cur=pq.top(); pq.pop();
```

```
        if(visited[cur.idx]) continue;
        visited[cur.idx]=true;
        out.visited_order.push_back(cur.idx);

        if(cur.idx==goalIdx){ out.found=true; break; }

        int r=cur.idx/C, c=cur.idx%C;
        for(auto [nr,nc]: neighbors(r,c)){
            int ni=grid.idx(nr,nc);
            if(visited[ni]) continue;
            int nd=dist[cur.idx]+1;
            if(nd<dist[ni]){
                dist[ni]=nd;
                parent[ni]=cur.idx;
                pq.push({ni, nd});
            }
        }
    }

    if(out.found){
        int cur=goalIdx;
        while(cur!=-1){ out.path.push_back(cur); cur=parent[cur]; }
        reverse(out.path.begin(), out.path.end());
    }
    return out;
}

// ---------------- main ----------------
int main(){
    GridModel grid(5,5);
    grid.start={0,0}; grid.goal={4,4};
    grid.set(1,2,CellType::Block);
    grid.set(2,2,CellType::Block);
    grid.set(3,2,CellType::Block);

    auto res=Dijkstra(grid);

    cout << "Dijkstra 탐색 순서: ";
    for(int idx: res.visited_order) cout << "("<<idx/5<<","<<idx%5<<") ";
    cout << "₩n";
```

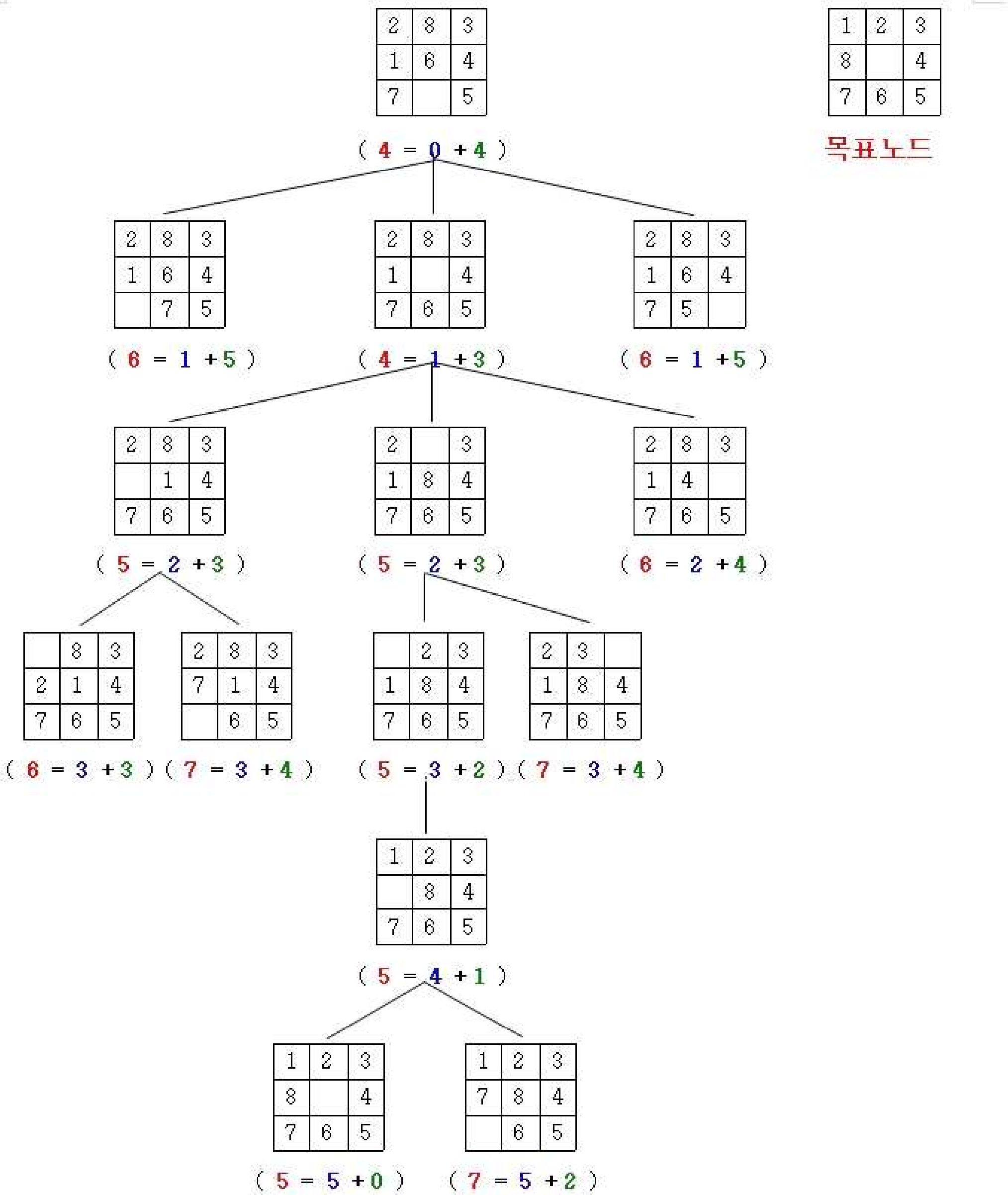
```
    if(res.found){
        cout << "Dijkstra 경로: ";
        for(int idx: res.path) cout << "("<<idx/5<<","<<idx%5<<") ";
        cout << "₩n";
    }else cout << "경로 없음₩n";

    return 0;
}
```

2.4.6. 벨만-포드 (Bellman-Ford) - 음수 허용 & 음수 사이클 검출



- 원리: 모든 간선 완화(relax) 작업을 V−1회 반복, 추가 1회에서 변하면 음수 사이클.
- 복잡도: O(VE).
- 언제 쓰나: 음수 간선이 존재할 수 있을 때, 또는 음수 사이클 유무 확인.

2.4.7. A* (A-star) - 휴리스틱 최단경로



- 개념: f(n)=g(n)+h(n) (시작→n 실제비용 g, n→목표 추정치 h)로 우선순위 큐 탐색.
- 최적성 조건:
    - 허용적(admissible): h(n) ≤ 실제 비용 → 최적해 보장
    - 일관/모노토닉(consistent): h(u) ≤ w(u,v)+h(v) → 재방문 줄고 Dijkstra처럼 동작
- 휴리스틱 예: 격자 맨해튼 거리, 유클리드 거리, 지형 가중치 반영 등.
- 복잡도: O(E)~지수적까지 h 품질에 좌우(좋은 h일수록 탐색 축소).
- 활용: 경로 계획(로보틱스/게임), 지도 내비.
- 소스코드

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>
```

```
#include <algorithm>
#include <cmath>

using namespace std;

enum class CellType { Empty, Block, Start, End };

struct GridModel {
    int rows, cols;
    vector<vector<CellType>> cells;
    pair<int,int> start{-1,-1}, goal{-1,-1};

    GridModel(int r, int c) : rows(r), cols(c), cells(r, vector<CellType>(c,
CellType::Empty)) {}

    bool inBounds(int r,int c) const { return r>=0 && r<rows && c>=0 && c<cols; }
    CellType at(int r,int c) const { return cells[r][c]; }
    void set(int r,int c, CellType t) { cells[r][c]=t; }
    int idx(int r,int c) const { return r*cols + c; }
    pair<int,int> coord(int idx) const { return {idx/cols, idx%cols}; }
};

// Manhattan 거리
int manhattan(int r1,int c1,int r2,int c2){ return abs(r1-r2)+abs(c1-c2); }

struct PQNode { int idx,f,g; bool operator<(const PQNode& o) const { return f>o.f; }
};

struct SearchResult { bool found=false; vector<int> visited_order; vector<int> path; };

SearchResult AStar(const GridModel& grid){
    SearchResult out;
    if(grid.start.first<0 || grid.goal.first<0) return out;

    int R=grid.rows, C=grid.cols;
    int startIdx=grid.idx(grid.start.first, grid.start.second);
    int goalIdx=grid.idx(grid.goal.first, grid.goal.second);

    vector<int> g(R*C, numeric_limits<int>::max());
    vector<int> parent(R*C, -1);
    vector<bool> closed(R*C, false);
```

```
    priority_queue<PQNode> open;

    g[startIdx]=0;
    open.push({startIdx, manhattan(grid.start.first, grid.start.second, grid.goal.first, grid.goal.second), 0});

    auto neighbors=[&](int r,int c){
        static const int dr[4]={-1,1,0,0};
        static const int dc[4]={0,0,-1,1};
        vector<pair<int,int>> nb;
        for(int k=0;k<4;k++){
            int nr=r+dr[k], nc=c+dc[k];
            if(grid.inBounds(nr,nc) && grid.at(nr,nc)!=CellType::Block) nb.push_back({nr,nc});
        }
        return nb;
    };

    while(!open.empty()){
        auto cur=open.top(); open.pop();
        if(closed[cur.idx]) continue;
        closed[cur.idx]=true;
        out.visited_order.push_back(cur.idx);

        if(cur.idx==goalIdx){ out.found=true; break; }

        int r=cur.idx/C, c=cur.idx%C;
        for(auto [nr,nc]: neighbors(r,c)){
            int ni=grid.idx(nr,nc);
            if(closed[ni]) continue;
            int tentative=g[cur.idx]+1;
            if(tentative<g[ni]){
                g[ni]=tentative;
                parent[ni]=cur.idx;
                int h=manhattan(nr,nc,grid.goal.first,grid.goal.second);
                open.push({ni, tentative+h, tentative});
            }
        }
    }

    if(out.found){
```

```
        int cur=goalIdx;
        while(cur!=-1){ out.path.push_back(cur); cur=parent[cur]; }
        reverse(out.path.begin(), out.path.end());
    }
    return out;
}

// ---------------- main ----------------
int main(){
    GridModel grid(5,5);
    grid.start={0,0}; grid.goal={4,4};
    grid.set(1,2,CellType::Block);
    grid.set(2,2,CellType::Block);
    grid.set(3,2,CellType::Block);

    auto res=AStar(grid);

    cout << "A* 탐색 순서: ";
    for(int idx: res.visited_order) cout << "("<<idx/5<<","<<idx%5<<") ";
    cout << "₩n";

    if(res.found){
        cout << "A* 경로: ";
        for(int idx: res.path) cout << "("<<idx/5<<","<<idx%5<<") ";
        cout << "₩n";
    }else cout << "경로 없음₩n";

    return 0;
}
```