자료구조 (3)

25.05.19 (월) 오후 5시 ~ 7시

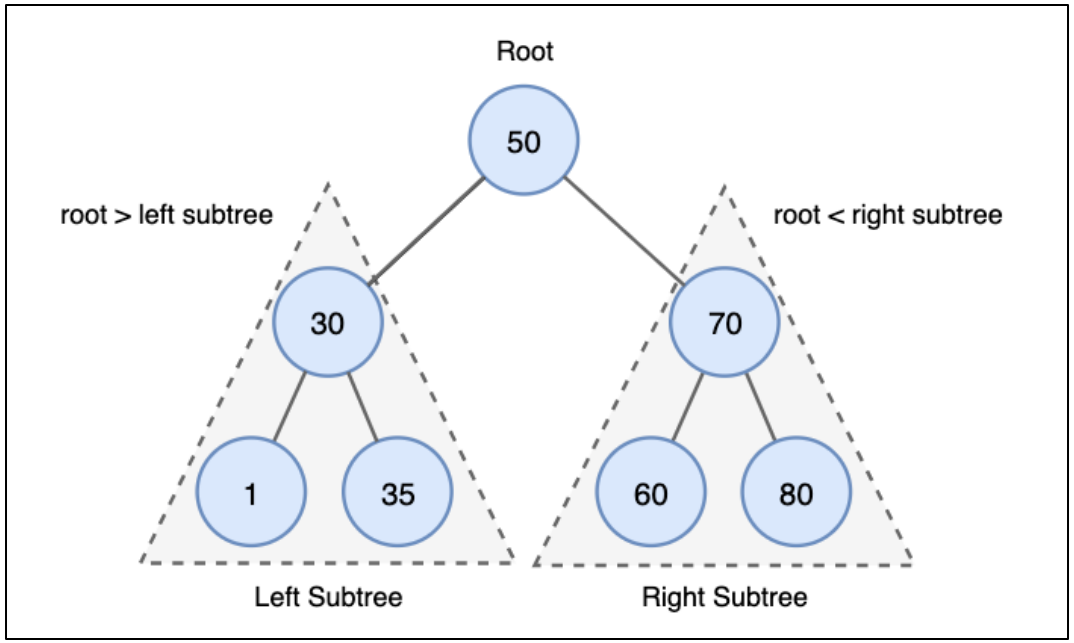
목차

- 이진 탐색 트리
- 우선순위 큐(힙)
- 서로소 집합
- 최소스패닝트리

이진 탐색 트리

이진 탐색 트리 | 01 개요

- 이진 트리의 한 종류인 이진 탐색 트리(이하, BST)는 다음과 같은 조건을 만족하는 트리를 말한다.
  - 어떤 노드의 왼쪽 서브트리에는 해당 노드보다 작은 값들만 존재한다.
  - 어떤 노드의 오른쪽 서브트리에는 해당 노드보다 큰 값들만 존재한다.
  - 위 규칙이 모든 서브트리에 대해 재귀적으로 적용되며, 중복된 키(노드)를 가지지 않는다.



이진 탐색 트리 | 02 연산

- 이진 탐색 트리는 탐색, 삽입, 삭제 연산을 지원한다.
  - 탐색 : 특정한 키가 BST에 존재하는지 확인
  - 삽입 : 트리에 새로운 값을 추가
  - 삭제 : 트리에서 특정 키를 가진 노드를 제거
- C++ STL에서 '자기 균형' 이진 탐색 트리를 제공한다. 모든 연산이 O(log n)에 동작한다.

```cpp
#include <iostream>
#include <set>
using namespace std;

int main() {
    set<int> s;
    s.insert(10);
    s.insert(5);
    s.insert(20);

    for (int x : s) cout << x << " ";
    if (s.find(10) != s.end())
        cout << "\n10 found\n";
    s.erase(10);
}
```

이진 탐색 트리 | 02 연산

- **탐색**
  - 정의 : 특정한 키가 트리 안에 존재하는지 확인하는 연산이다.
  - 동작 방식 : 현재 노드와 키를 비교하면서 왼쪽 혹은 오른쪽 서브트리로 재귀적으로 이동한다.
  - 시간 복잡도 : 평균적으로 O(log n)의 연산이다. 편향된 트리일 경우 O(n)이다.
  - 구현
    - 현재 노드가 null이면 → 찾는 값 없음
    - 현재 노드의 값 == 찾는 키 → 찾았다!
    - 키 < 현재 노드의 값 → 왼쪽 서브트리로 이동
    - 키 > 현재 노드의 값 → 오른쪽 서브트리로 이동

```cpp
int searchBST(const vector<int>& tree, int key, int index = 0) {
    // 범위를 벗어나거나 빈 노드(-1)일 경우 실패
    if (index >= tree.size() || tree[index] == -1) return -1;
    if (tree[index] == key) return index;

    if (key < tree[index]) return searchBST(tree, key, 2 * index); // 왼쪽 자식
    return searchBST(tree, key, 2 * index + 1); // 오른쪽 자식
}
```

이진 탐색 트리 | 02 연산

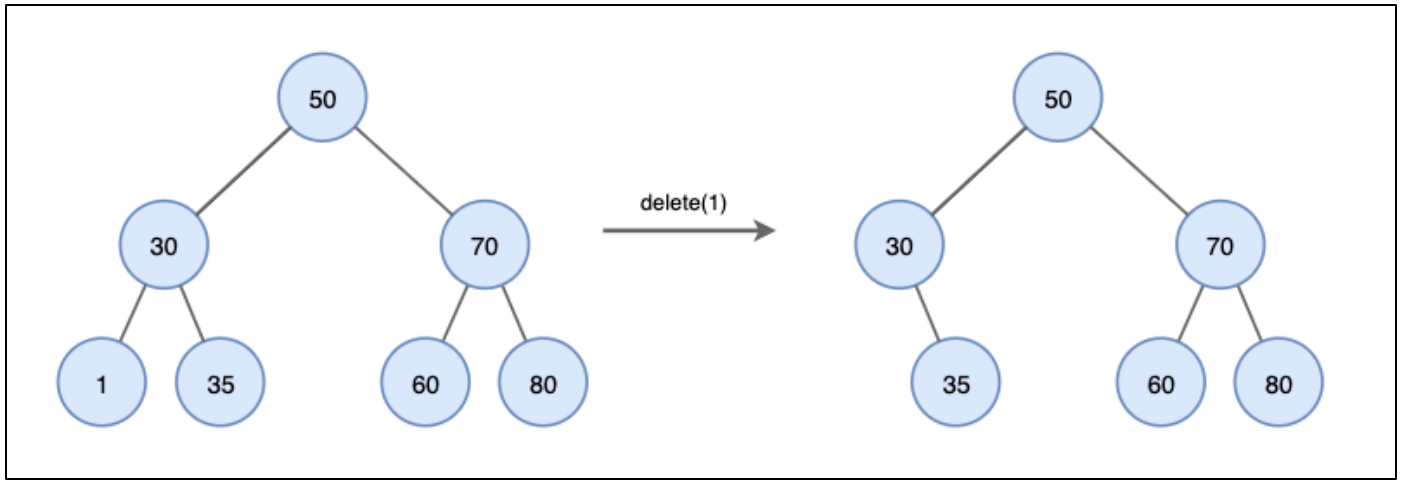
- **삽입**
  - 정의 : 트리에 새로운 값을 추가하는 연산이다. 중복은 허용되지 않는다.
  - 동작 방식 : 넣을 위치를 탐색하고, 새 키는 항상 리프 노드에 삽입된다.
  - 시간 복잡도 : 평균적으로 O(log n)의 연산이다. 편향된 트리일 경우 O(n)이다.
  - 구현
    - 현재 노드가 비어있다면, 노드를 삽입한다.
    - 키 < 현재 노드이면, 왼쪽 서브트리에 삽입한다.
    - 키 > 현재 노드이면, 오른쪽 서브트리에 삽입한다.
    - 키 = 현재 노드이면, 삽입하지 않고 종료한다.

```
2  void insertBST(vector<int>& tree, int key, int index = 1) {
3
4      if (tree[index] == -1) {
5          tree[index] = key;
6          return;
7      }
8
9      if (key < tree[index]) {
10         insertBST(tree, key, 2 * index);
11     } else if (key > tree[index]) {
12         insertBST(tree, key, 2 * index + 1);
13     }
14 }
```

이진 탐색 트리 | 02 연산

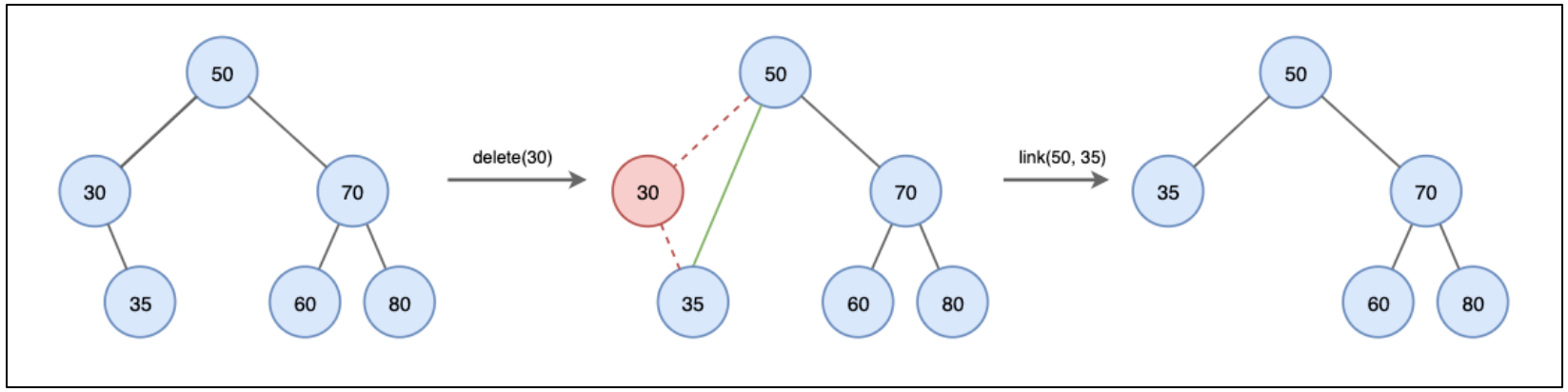
- **삭제** *가장 까다로운 연산!
  - 정의 : 트리에서 특정 키를 가진 노드를 제거한다.
  - 동작 방식 : 3가지 케이스로 나누어 처리한다.
  - 시간 복잡도 : 평균적으로 O(log n)의 연산이다. 편향된 트리일 경우 O(n)이다.

  - Case 1) 삭제할 노드가 리프노드인 경우 : 그냥 제거하면 된다.



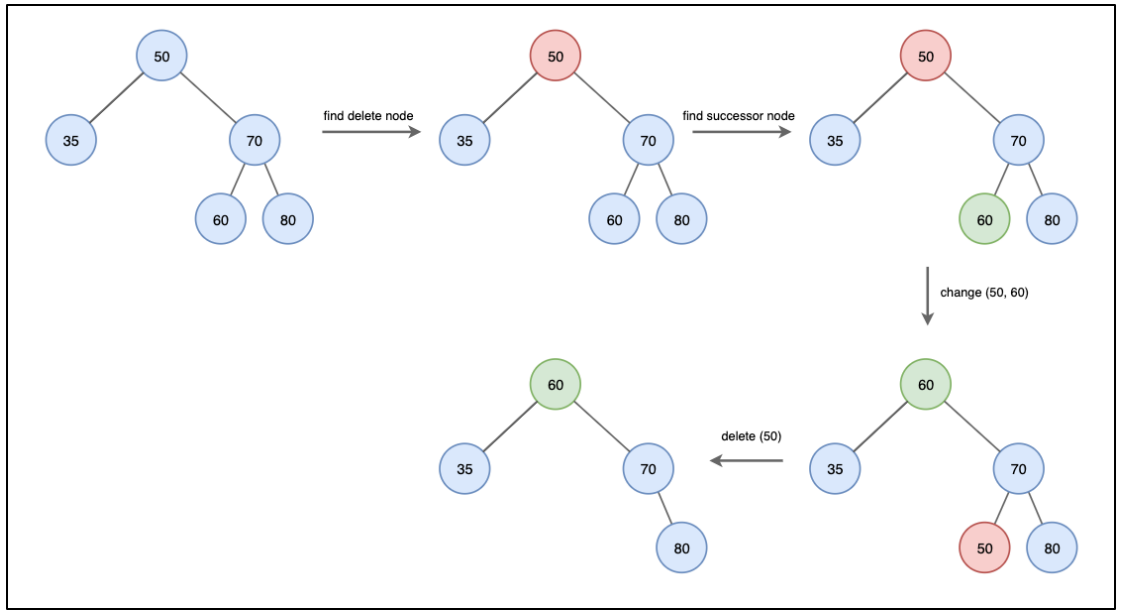
이진 탐색 트리 | 02 연산

- **삭제** *가장 까다로운 연산!
  - Case 2) 삭제할 노드에 자식이 하나만 있는 경우 : 자식 노드를 노드의 부모에 직접 연결하고 노드를 삭제한다.



이진 탐색 트리 | 02 연산

- **삭제** *가장 까다로운 연산!
  - Case 3) 삭제할 노드에 자식이 2개 있는 노드 : 삭제할 노드를 찾고, 그것의 '후계자' 노드를 찾는다. 값을 바꾸고 '후계자' 노드를 삭제한다.
    - 어떤 노드의 왼쪽 서브트리는 해당 노드보다 작은 값만 존재하고, 오른쪽 서브트리는 해당 노드보다 큰 값만 존재한다.
    - 왼쪽 서브트리에서 가장 오른쪽에 있는 노드, 오른쪽 서브트리에서 가장 왼쪽에 있는 노드가 '후계자' 노드가 될 수 있다.
    - 이 후계자 노드를 삭제할 노드의 위치에 덮어쓰고, 후계자 노드(리프 노드)를 삭제한다.



이진 탐색 트리 | 03 std::set<typename>

- set은 균형 이진 탐색 트리 기반으로, 중복 없는 원소들을 정렬된 상태로 저장하는 자료구조이다.
  - 중복을 허용하는 균형 이진 탐색 트리 자료구조를 원한다면, multiset을 이용하면 된다.
  - 참고) unordered_set은 해시 테이블을 기반으로 하며, 탐색/삽입/삭제 모두 amortized O(1)이다. 최악의 경우, O(n)이다.
- 모든 연산이 최악의 경우에도 O(log n)의 시간복잡도를 가진다.

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> s;

    // 삽입
    s.insert(5);
    s.insert(3);
    s.insert(7);
    s.insert(5); // 중복 → 삽입 안 됨

    // 탐색
    if (s.count(3)) std::cout << "3 있음\n";

    // 삭제
    s.erase(5);

    // 순회 (자동 오름차순 정렬)
    for (int x : s) {
        std::cout << x << ' ';
    }
    // 출력: 3 7

    return 0;
}
```

이진 탐색 트리 | 04 std::map<key, value>

- map은 균형 이진 탐색 트리 기반으로, 중복 없는 키-값 쌍을 저장하며 정렬된 상태를 유지하는 자료구조이다.
  - 참고) unordered_map은 해시 테이블을 기반으로 하며, 탐색/삽입/삭제 모두 amortized O(1)이다. 최악의 경우, O(n)이다.
- 모든 연산이 최악의 경우에도 O(log n)의 시간복잡도를 가진다.

```cpp
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> m;

    // 삽입
    m["apple"] = 3;
    m["banana"] = 5;
    m.insert({"orange", 2});

    // 접근
    std::cout << m["apple"] << '\n';     // 3
    std::cout << m.at("banana") << '\n'; // 5

    // 존재 확인
    if (m.count("grape")) {
        std::cout << "grape 있음\n";
    } else {
        std::cout << "grape 없음\n";
    }

    // 순회 (정렬된 순서대로)
    for (const auto& pair : m) {
        std::cout << pair.first << " : " << pair.second << '\n';
    }

    return 0;
}
```

이진 탐색 트리 | 05 예제 : 백준 1269번 대칭 차집합

- **문제 요약**
  - 자연수를 원소로 갖는 공집합이 아닌 두 집합 A와 B가 있다.
  - 이때, 두 집합의 대칭 차집합의 원소의 개수를 출력하는 프로그램을 작성하시오.
  - 두 집합 A와 B가 있을 때, (A-B)와 (B-A)의 합집합을 A와 B의 대칭 차집합이라고 한다.
- **입력**
  - 집합 A, B가 주어진다.
- **출력**
  - 대칭 차집합의 원소의 개수를 출력한다.

이진 탐색 트리 | 05 예제 : 백준 1269번 대칭 차집합

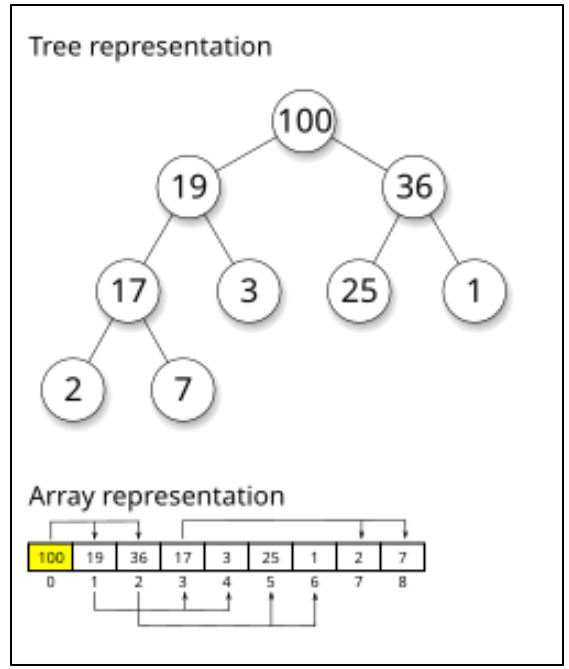
- **구현**
  - stl의 set을 그대로 이용하면 된다.

```cpp
#include <bits/stdc++.h>
using namespace std;

set<int> A;
set<int> B;
int a, b, res;

int solution(set<int> a, set<int> b){ // a - b
  int ans = 0;
  for(auto iter : a){
    if(b.find(iter) == b.end()) ans++;
  }
  return ans;
}

int main(){
  ios::sync_with_stdio(false);
  cin.tie(0); cout.tie(0);

  cin >> a >> b;
  for(int i = 0; i < a; i++){ int tmp; cin >> tmp; A.insert(tmp); };
  for(int i = 0; i < b; i++){ int tmp; cin >> tmp; B.insert(tmp); };
  res += solution(A, B);
  res += solution(B, A);
  cout << res << "\n";
  return 0;
}
```

우선순위 큐(힙)

우선순위 큐(힙) | 01 개요

- 힙은 Complete Binary Tree의 일종으로, 가장 큰/작은 값만 빠르게 찾을 수 있는 자료구조 이다.
  - 힙은 최대 힙과 최소 힙으로 나뉜다.
  - 최대 힙 : 각 노드는 자식 노드보다 크거나 같음. 즉, 루트에는 항상 최댓값이 있음.
    즉, A[i] ≥ A[2 * i] AND A[i] ≥ A[2 * i + 1]
  - 최소 힙 : 각 노드는 자식 노드보다 작거나 같음. 즉, 루트에는 항상 최솟값이 있음.
    즉, A[i] ≤ A[2 * i] AND A[i] ≤ A[2 * i + 1]
  - (나중에 알아봄) C++에서는 사용자가 직접 우선순위를 결정할 수 있다.



- 힙은 높이가 O(log n)인 완전 이진 트리로, 주요 연산의 시간 복잡도는 다음과 같다.
  - 삽입 : O(log n)
  - 삭제 : O(log n)
  - top 조회 : O(1)

우선순위 큐(힙) | 02 std::priority_queue<typename>

- priority_queue는 큐처럼 쓰지만, 넣은 순서가 아니라 우선순위가 가장 높은 값부터 꺼내는 구조이다.
  - 기본은 최대 힙으로 큰 값부터 나온다. 최소 힙을 사용하고 싶으면 다음과 같이 선언하면 된다.

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue<int> pq; // 최대 힙
    // priority_queue<int, vector<int>, greater<int>> min_pq; // 최소 힙

    pq.push(5);
    pq.push(1);
    pq.push(10);

    while (!pq.empty()) {
        cout << pq.top() << " ";  // 가장 큰 값부터 출력됨
        // cout << min_pq.top() << " ";  // 가장 작은 값부터 출력됨
        pq.pop();
    }
    return 0;
}
```

우선순위 큐(힙) | 03 예제 (1) : 백준 11286번 절댓값 힙

- **문제요약**
  - 절댓값 힙은 다음과 같은 연산을 지원하는 자료구조이다.
    - 배열에 정수 x (x ≠ 0)를 넣는다.
    - 배열에서 절댓값이 가장 작은 값을 출력하고, 그 값을 배열에서 제거한다. 절댓값이 가장 작은 값이 여러개일 때는, 가장 작은 수를 출력하고, 그 값을 배열에서 제거한다.
- **입력**
  - 첫째 줄에 연산의 개수 N(1≤N≤100,000)이 주어진다.
  - 다음 N개의 줄에는 연산에 대한 정보를 나타내는 정수 x가 주어진다.
  - 만약 x가 0이 아니라면 배열에 x라는 값을 넣는(추가하는) 연산이고, x가 0이라면 배열에서 절댓값이 가장 작은 값을 출력하고 그 값을 배열에서 제거하는 경우이다. 입력되는 정수는 -2^31보다 크고, 2^31보다 작다.
- **출력**
  - 입력에서 0이 주어진 회수만큼 답을 출력한다. 만약 배열이 비어 있는 경우인데 절댓값이 가장 작은 값을 출력하라고 한 경우에는 0을 출력하면 된다.

우선순위 큐(힙) | 03 예제 (1) : 백준 11286번 절댓값 힙

- **문제풀이**
  - stl에서 제공하는 우선순위 큐는 사용자가 우선순위 기준을 직접 정의할 수 있다.
  - 이를 활용해서 문제의 규칙을 구현할 수 있다.
  - 문제의 규칙
    1. 절댓값이 작은 값 우선
    2. 절댓값이 같으면 더 작은 수, 즉 음수 우선

우선순위 큐(힙) | 03 예제 (1) : 백준 11286번 절댓값 힙

- **구현**
  - Compare : 정렬 기준을 정의하는 함수 객체로, 작을수록 우선순위가 높다고 판단되도록 정의해야 함. 즉, Compare 함수는 두 요소 a, b를 비교해서 'b가 a보다 우선순위가 높으면 true를 반환'해야 한다.
    b가 a보다 더 먼저 나와야 한다면, true를 반환하면 된다.
  - 그렇기에, 절댓값이 같으면 a > b를 반환하는 것임.
  - 그렇기에 abs(a) > abs(b)를 반환하는 것임.

```cpp
#include <iostream>
#include <queue>
#include <cmath>
using namespace std;

struct Compare {
    bool operator()(int a, int b) {
        if (abs(a) == abs(b)) return a > b;
        return abs(a) > abs(b);
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n;
    cin >> n;
    priority_queue<int, vector<int>, Compare> pq;

    while (n--) {
        int x;
        cin >> x;
        if (x != 0) {
            pq.push(x);
        } else {
            if (pq.empty()) cout << "0\n";
            else {
                cout << pq.top() << '\n';
                pq.pop();
            }
        }
    }
}
```

우선순위 큐(힙) | 03 예제 (2) : 백준 1715번 카드 정렬하기

- **문제요약**
  - 정렬된 두 묶음의 숫자 카드가 있다고 하자. 각 묶음의 카드의 수를 A, B라 하면 보통 두 묶음을 합쳐서 하나로 만드는 데에는 A+B 번의 비교를 해야 한다. 이를테면, 20장의 숫자 카드 묶음과 30장의 숫자 카드 묶음을 합치려면 50번의 비교가 필요하다.
  - 매우 많은 숫자 카드 묶음이 책상 위에 놓여 있다. 이들을 두 묶음씩 골라 서로 합쳐나간다면, 고르는 순서에 따라서 비교 횟수가 매우 달라진다. 예를 들어 10장, 20장, 40장의 묶음이 있다면 10장과 20장을 합친 뒤, 합친 30장 묶음과 40장을 합친다면 (10 + 20) + (30 + 40) = 100번의 비교가 필요하다. 그러나 10장과 40장을 합친 뒤, 합친 50장 묶음과 20장을 합친다면 (10 + 40) + (50 + 20) = 120 번의 비교가 필요하므로 덜 효율적인 방법이다.
- **입력**
  - 첫째 줄에 N이 주어진다. ($1 \le N \le 100,000$) 이어서 N개의 줄에 걸쳐 숫자 카드 묶음의 각각의 크기가 주어진다. 숫자 카드 묶음의 크기는 1,000보다 작거나 같은 양의 정수이다.
- **출력**
  - 첫째 줄에 최소 비교 횟수를 출력한다.

우선순위 큐(힙) | 03 예제 (2) : 백준 1715번 카드 정렬하기

- **문제풀이**
  - 작은 카드 뭉치 2개를 선택해서 합치는 과정을 카드 뭉치가 1개가 될때까지 반복하면 된다.
  - 이는 그리디적 요소가 들어간 문제로, 일반화된 논리로 증명이 가능하다.
    - 두 카드 묶음을 합친 비용은 그 합이 그대로 누적 비용이 됨.
    - 이 합쳐진 결과는 다음 번 합산에서도 더 큰 비용을 발생시킬 수 있음 (누적됨).
    - 따라서 초기 단계에서 큰 값끼리 먼저 합치면, 그 결과 묶음도 커지고 다음 합산 비용에도 불리한 영향을 끼침.
    - 반면, 작은 값끼리 먼저 합치면, 초기 비용도 작고 이후의 비용 누적 효과도 적음.

우선순위 큐(힙) | 03 예제 (2) : 백준 1715번 카드 정렬하기

- 구현

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    int n;
    cin >> n;

    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        pq.push(x);
    }

    int total = 0;
    while (pq.size() > 1) {
        int a = pq.top(); pq.pop();
        int b = pq.top(); pq.pop();
        int sum = a + b;
        total += sum;
        pq.push(sum);
    }

    cout << total << '\n';
    return 0;
}
```

서로소 집합

서로소 집합 | 01 개요

- **서로소 집합(Disjoint Set)이란, 서로 교집합이 없는 집합들의 모임**을 뜻한다. 즉, 어떤 원소도 두 개 이상의 집합에 동시에 속할 수 없다.
  - {1, 2}, {3, 4}, {5}는 서로소 집합이지만, {1, 2}, {2, 3}은 서로소 집합이 아님 (2가 두 집합에 있음)

- 친구 관계를 생각해보면, 내 친구의 친구는 친구(?)이다. 즉, 누가 누구랑 친구인지 따라 여러 개의 서로 연결된 집단이 만들어진다.
  - 이 집단들을 구분하기 위해 "같은 집단인가?" 또는 "두 집단을 하나로 합칠까?" 같은 연산이 필요하다.
  - 이걸 효율적으로 처리하기 위한 자료구조가 서로소 집합이다.

- '서로소 집합' 자료구조는 **2가지 연산**을 제공한다.
  - "같은 집단인가?"라는 질문을 해결할 수 있는 **find(x) 함수**
  - "두 집단을 하나로 합치는" **union(x, y) 함수**를 제공한다.

서로소 집합 | 02 연산

- '서로소 집합' 자료구조를 구현하기 위해서는 **parent 배열과 2개의 함수**가 필요하다.

- **parent[x] = y**
  - x의 **'루트'** 노드는 y라는 정보를 담은 배열이다. 이를 통해 같은 집합에 속해있는지, 그렇지 않은지 확인할 수 있다.

- **find(x)**
  - x가 속한 집합의 루트 노드를 반환하는 연산이다.
  - 루트 노드를 따라가면 자기 자신을 루트로 갖는 노드가 존재함.
  - 보통 트리 구조로 표현하며, 이 루트 노드를 통해 집합을 식별할 수 있다.

- **union(x, y)**
  - x와 y가 속한 서로 다른 집합을 하나로 합치는 연산이다.
  - find(x)와 find(y)를 먼저 해서 루트를 찾고, 루트 노드가 같지 않다면, 한 쪽 루트를 다른 쪽에 붙인다.

서로소 집합 | 03 구현

- 서로소 집합은 보통 트리 형태로 집합이 표현된다.
- 처음에는 각각이 루트 노드인 상태에서 시작한다.
- **int Find(x) : x의 루트 노드(부모가 아님!)를 반환한다.**
  - x의 부모 노드가 x라면, x는 루트 노드이다.
  - x의 부모 노드가 x가 아니라면, parent[x]의 루트 노드를 반환하는 함수를 함수를 호출한다.
- **void Union(int a, int b) : a를 포함하는 집합, b를 포함하는 집합을 합친다.**
  - a의 루트 노드와 b의 루트 노드를 찾는다.
  - 루트 노드가 같지 않다면(서로 다른 집합이라면), 한쪽 루트를 다른 쪽에 연결한다.

```
1   int parent[MAX];  // MAX는 원소의 수 (예: 100000)
2
3   for (int i = 1; i <= n; i++) {
4       parent[i] = i;
5   }
6
7   int Find(int x) {
8       if (parent[x] == x) return x;        // 자기 자신이 루트
9       else return find(parent[x]);         // 부모 따라 계속 탐색
10  }
11
12  void Union(int a, int b) {
13      a = find(a);
14      b = find(b);
15      if (a != b) parent[b] = a;  // 한쪽 루트를 다른 쪽에 연결
16  }
```

서로소 집합 | 04 최적화

- find(x) 연산은 'x의 루트노드를 찾는 연산'이다. 이 연산은 트리의 레벨이 클수록 느려지게 된다. 서로소 집합에서 가장 이상적인 트리의 모습은 루트 노드에 모든 자식 노드들이 연결되어 있는, 즉 트리의 높이가 1인 트리일 것이다. 이를 위한 최적화 방법을 생각할 수 있다.

1. 경로 압축

```
18  int Find(int x) {
19      if (parent[x] == x) return x;
20      return parent[x] = Find(parent[x]);
21  }
```

2. 랭크 기준, 사이즈 기준 합치기

무작정 한쪽 루트를 다른 쪽에 합치면 트리의 깊이가 증가할 수 있다. 그렇기에, 작은 트리를 큰 트리에 붙이는 전략이 효율적이다.

랭크 기준 : 랭크를 기준으로, 사이즈를 기준으로 어떤 노드를 루트 노드로 삼을지 결정할 수 있다.

사이즈 기준 : 작은 집합을 큰 집합에 붙이는게 이득이고, 낮은 랭크를 가진 집합을 큰 랭크 집합에 붙이는게 이득이다.

경로 압축과 랭크(사이즈) 기준 합치기를 동시에 적용하면 **O(1)에 가까운 성능**을 낸다.

서로소 집합 | 05 예제 : 백준 1717번 집합의 표현

문제

초기에 $n+1$개의 집합 $\{0\}, \{1\}, \{2\}, \dots, \{n\}$이 있다. 여기에 합집합 연산과, 두 원소가 같은 집합에 포함되어 있는지를 확인하는 연산을 수행하려고 한다.

집합을 표현하는 프로그램을 작성하시오.

입력

첫째 줄에 n, m이 주어진다. m은 입력으로 주어지는 연산의 개수이다. 다음 m개의 줄에는 각각의 연산이 주어진다. 합집합은 $0\ a\ b$의 형태로 입력이 주어진다. 이는 a가 포함되어 있는 집합과, b가 포함되어 있는 집합을 합친다는 의미이다. 두 원소가 같은 집합에 포함되어 있는지를 확인하는 연산은 $1\ a\ b$의 형태로 입력이 주어진다. 이는 a와 b가 같은 집합에 포함되어 있는지를 확인하는 연산이다.

출력

1로 시작하는 입력에 대해서 a와 b가 같은 집합에 포함되어 있으면 " YES " 또는 " yes "를, 그렇지 않다면 " NO " 또는 " no "를 한 줄에 하나씩 출력한다.

서로소 집합 | 05 예제 : 백준 1717번 집합의 표현

- 구현

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int MAX = 1'000'001;
int parent[MAX];
int setSize[MAX];

int find(int x) {
    if (parent[x] != x)
        parent[x] = find(parent[x]);
    return parent[x];
}

void union_sets(int a, int b) {
    int ra = find(a);
    int rb = find(b);
    if (ra == rb) return;

    if (setSize[ra] < setSize[rb]) {
        parent[ra] = rb;
        setSize[rb] += setSize[ra];
    } else {
        parent[rb] = ra;
        setSize[ra] += setSize[rb];
    }
}
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, m;
    cin >> n >> m;

    for (int i = 0; i <= n; ++i) {
        parent[i] = i;
        setSize[i] = 1;
    }

    while (m--) {
        int op, a, b;
        cin >> op >> a >> b;
        if (op == 0) {
            union_sets(a, b);
        } else {
            if (find(a) == find(b))
                cout << "YES\n";
            else
                cout << "NO\n";
        }
    }
}
```

최소스패닝트리

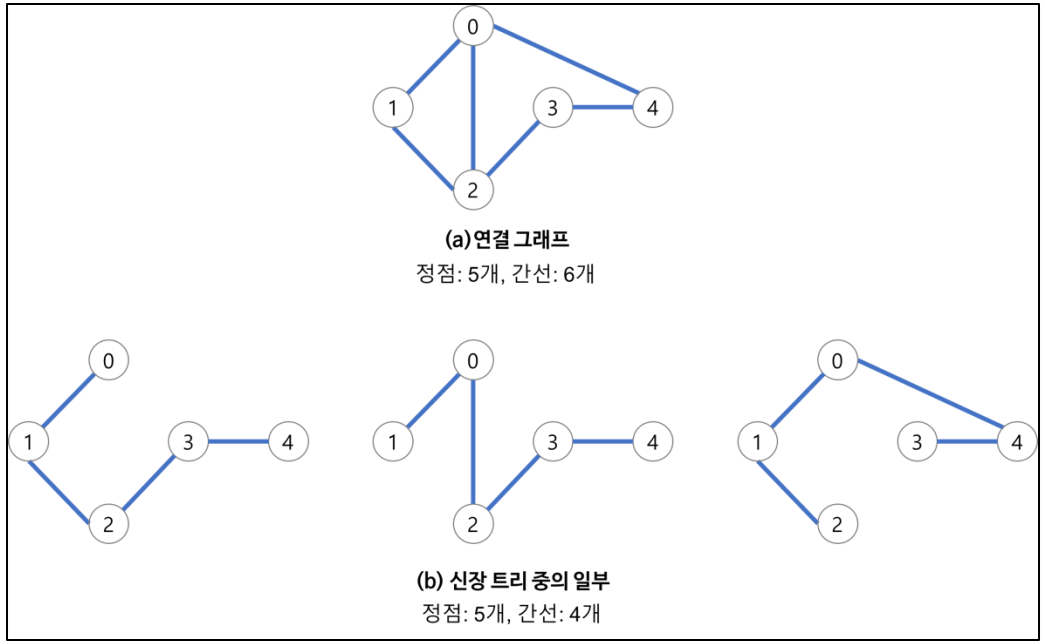
최소스패닝트리 | 01 개요

- **스패닝 트리란?**

  무방향 연결 그래프에 있는 **모든 정점을 포함**하되, **최소한의 간선으로만 연결**해 놓은 트리를 말한다.

  스패닝 트리는 하나가 아니고 다른 스패닝 트리가 존재할 수 있다.

  그 중에서 최소 스패닝 트리(이하, MST)는 가중치가 최소인 것을 의미한다.



최소스패닝트리 | 01 개요

- **MST의 정의**
  - MST는 가중치 있는 무방향 연결 그래프에서 만들 수 있는 **스패닝 트리 중 간선들의 가중치 합이 최소인 것**을 말한다!
  - 참고) MST도 유일하지 않을 수 있다.

- **MST를 구하기 위한 질문**
  - 모든 정점을 연결하면서 가중치의 총합을 최소로 하려면 어떤 간선을 선택해야 할까?
    - 가장 가중치가 작은 간선부터 하나씩 선택(그리디)하면서, 사이클이 생기지 않도록 선택해 나가면 된다!

- **MST 알고리즘**
  - MST를 구하는 대표적인 알고리즘에는 Kruskal's Algorithm과 Prim's Algorithm이 있는데, Kruskal's Algorithm만 다룰 예정이다.

최소스패닝트리 | 02 Kruskal's Algorithm

- **시간복잡도 : O(E log E) = 간선 정렬 O(Elog E) + 서로소 집합의 O(log V) 연산을 O(E)번 수행**
- **알고리즘 작동방식**

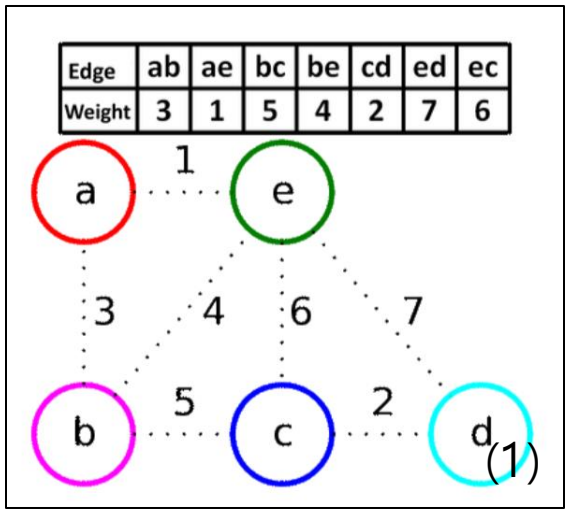
  스패닝 트리에 간선을 하나씩 포함시키는 방식으로 그리디 알고리즘을 기반으로 작동한다.

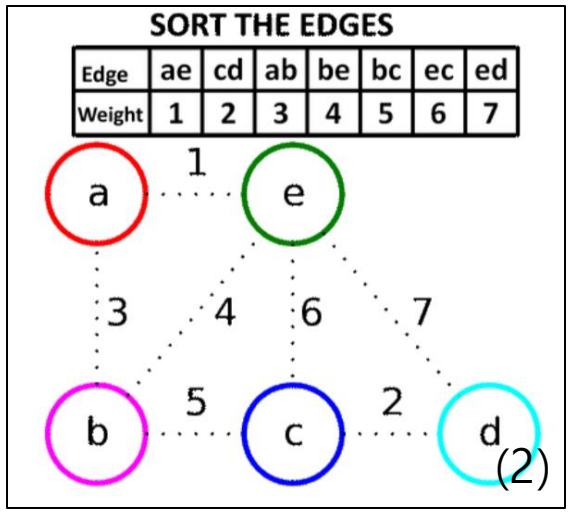
  1. 모든 간선을 가중치를 기준으로 오름차순 정렬
  2. 가중치가 작은 간선부터 보면서, 사이클이 생기지 않는다면 해당 간선을 MST를 구성하는 간선의 집합에 추가

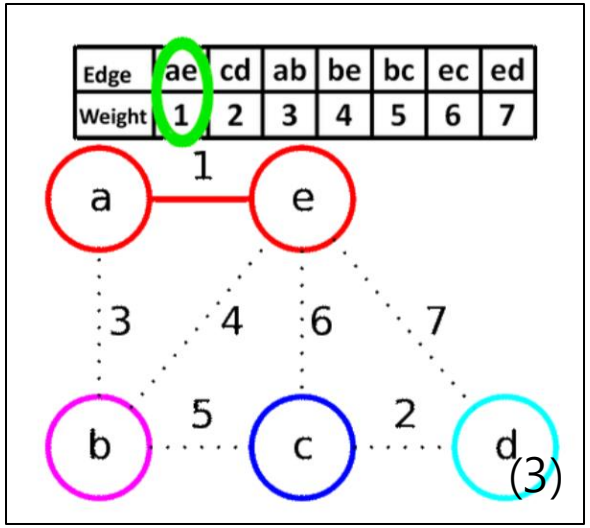
  <사이클 유무 판단 방법>

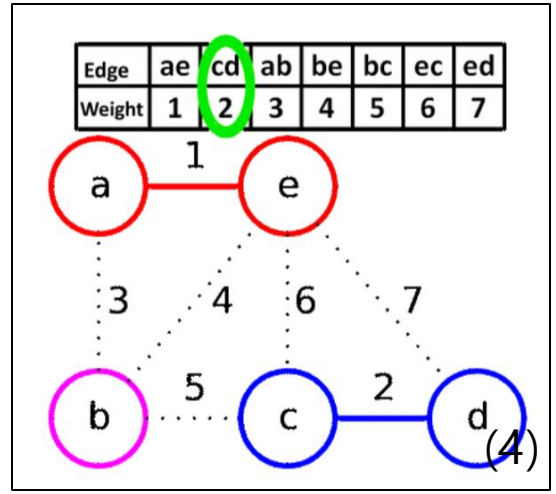
  간선 e = (u, v)에 대해. 정점 u와 v가 다른 집합에 있으면 연결했을 때, 사이클이 생기지 않음. → 서로소 집합 사용!

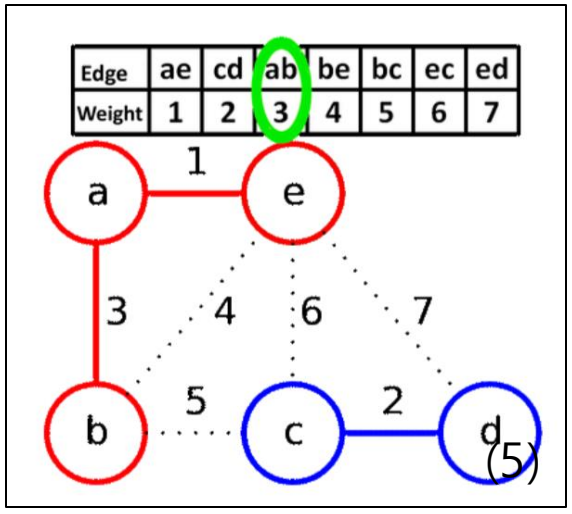
최소스패닝트리 | 02 Kruskal's Algorithm

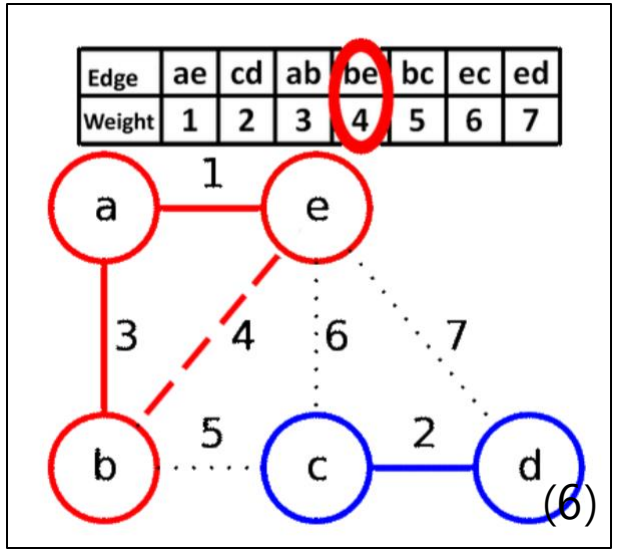


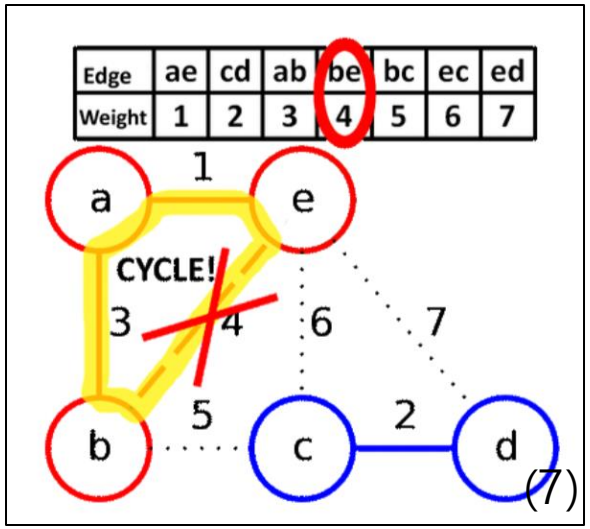












●●●

최소스패닝트리 | 02 Kruskal's Algorithm

- **구현**

    1. 모든 간선을 가중치 기준으로 정렬
    2. 각 정점을 **서로소 집합(disjoint-set)**으로 초기화
    3. 간선을 가중치 순으로 보며:
        - 두 정점이 서로 다른 집합에 속해 있다면, 해당 간선을 MST에 포함하고 두 집합을 합친다.
        - 간선 개수가 V - 1개가 되면 종료한다.

    *실제 구현은 문제를 통해 알아봅니다!

최소스패닝트리 | 03 예제 : 백준 1197번 최소 스패닝 트리

- **문제요약**
  - 그래프가 주어졌을 때, 그 그래프의 최소 스패닝 트리를 구하는 프로그램을 작성하시오.
  - 최소 스패닝 트리는, 주어진 그래프의 모든 정점들을 연결하는 부분 그래프 중에서 그 가중치의 합이 최소인 트리를 말한다.
- **입력**
  - 첫째 줄에 정점의 개수 V($1 \le V \le 10,000$)와 간선의 개수 E($1 \le E \le 100,000$)가 주어진다. 다음 E개의 줄에는 각 간선에 대한 정보를 나타내는 세 정수 A, B, C가 주어진다. 이는 A번 정점과 B번 정점이 가중치 C인 간선으로 연결되어 있다는 의미이다. C는 음수일 수도 있으며, 절댓값이 1,000,000을 넘지 않는다.
- **출력**
  - 첫째 줄에 최소 스패닝 트리의 가중치를 출력한다.

최소스패닝트리 | 03 예제 : 백준 1197번 최소 스패닝 트리

- 문제풀이

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Edge {
    int u, v, weight;
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

int V, E, parent[10001];

int Find(int x) {
    if (x == parent[x]) return x;
    return parent[x] = Find(parent[x]); // 경로 압축
}

void Union(int a, int b) {
    a = Find(a);
    b = Find(b);
    if (a != b) parent[b] = a;
}
```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> V >> E;

    vector<Edge> edges;
    for (int i = 0; i < E; i++) {
        int A, B, C;
        cin >> A >> B >> C;
        edges.push_back({A, B, C});
    }

    // 간선 정렬
    sort(edges.begin(), edges.end());

    // Union-Find 초기화
    for (int i = 1; i <= V; i++) parent[i] = i;

    int mst_weight = 0;
    int count = 0;
    for (Edge& e : edges) {
        if (find(e.u) != find(e.v)) {
            Union(e.u, e.v);
            mst_weight += e.weight;
            count++;
            if (count == V - 1) break; // MST 완성
        }
    }

    cout << mst_weight << '\n';
    return 0;
}
```

문제

- **기초**

  백준 11279번 최대 힙

  백준 1976번 여행 가자

- **심화**

  백준 5639번 이진 검색 트리

  백준 1922번 네트워크 연결

감사합니다!