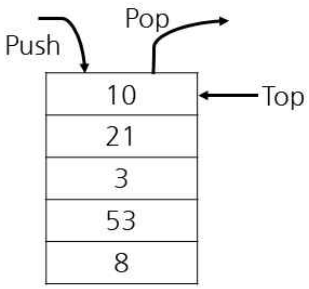
1. 복잡도(Complexity)
- 컴퓨터의 자원은 한정적이기 때문에, 데이터를 다룰 효율적인 알고리즘을 선택할 기준이 필요
- 공간 복잡도(Space Complexity)
  □ 해당 알고리즘(또는 함수)에 추가로 사용되는 메모리
- 시간 복잡도(Time Complexity)
  □ 해당 알고리즘에 사용되는 연산의 횟수
  □ Big-O 표기법 사용
- 점근 표기법

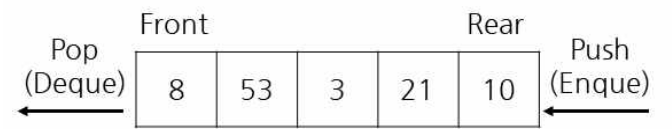
	표현	의미	예시
O(Big-O)	$f = O(g)$	$f \leqq g$	$n \subset O(n^2)$
Ω(Big-Omega)	$f = \Omega(g)$	$f \geqq g$	$n^3 \subset \Omega(n^2)$
Θ(Big-Theta)	$f = \Theta(g)$	$f = g$	$n^2 = \Theta(n^2)$

2. 배열(Array)
- Index, Value의 한 쌍으로 이루어진 자료구조
- 같은 데이터 타입을 가진 선형 구조, 연속된 메모리 공간에 데이터를 저장
- Index 값을 알면 해당 데이터에 쉽게 접근이 가능(탐색이 쉬움)
- 삽입/삭제에 시간이 오래 걸림
- 순차적인 데이터의 저장, 한정된 데이터 개수, 탐색을 주로 하는 경우에 사용

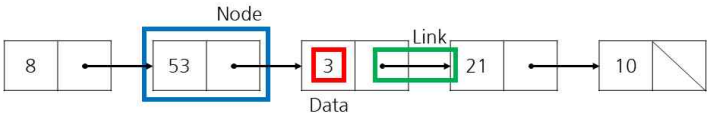
3. 스택(Stack)
- 마지막에 삽입한 데이터가 가장 먼저 삭제되는 LIFO(Last-In, First-Out) 구조
- 가장 위쪽의 데이터인 Top에만 접근 가능
- 깊이 우선 탐색(DFS), 재귀 함수에 사용
- 메모리의 스택 영역에는 지역 변수와 매개 변수가 저장



4. 큐(Queue)
- 가장 먼저 삽입한 데이터가 먼저 삭제되는 FIFO(First-In, First-Out) 구조
- 가장 앞에 있는 데이터인 Front에만 접근 가능
- 삽입이 일어나는 쪽을 Rear, 삭제가 일어나는 쪽을 Front라고 부름
- Buffer, 너비 우선 탐색(BFS), CPU Scheduling 등에 사용

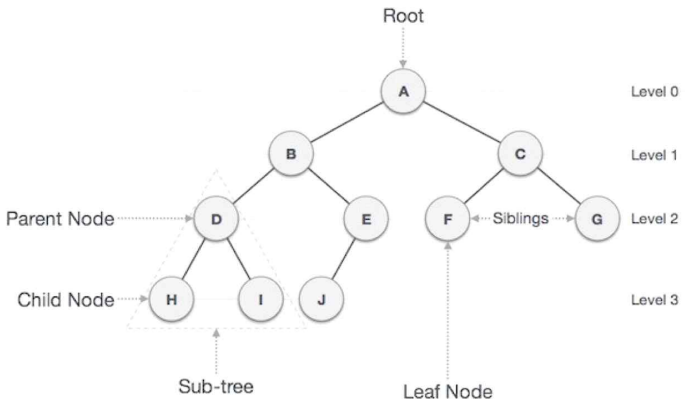


5. 리스트(List)
- 배열과 같이 데이터를 선형적으로 저장하는 자료구조
- 크기가 가변적이며 연결 리스트를 주로 사용
- 노드는 데이터와 링크로 구성



6. 트리(Tree)
- 노드들로 이루어진 자료구조로, 그래프의 한 종류
- 용어

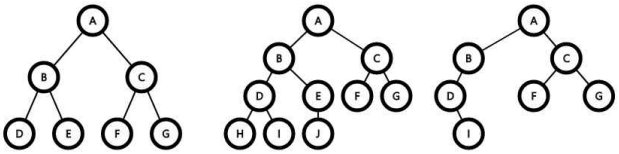
이름		의미
루트 노드		부모가 없는 노드, 트리는 오직 하나의 루트 노드를 가짐
리프 노드		자식이 없는 노드
내부 노드		리프 노드가 아닌 노드
형제 노드		같은 부모를 가지는 노드
깊이		루트에서 해당 노드까지 도달하기 위해 거치는 간선의 수
레벨		특정 깊이를 가지는 노드의 집합
차수	노드	해당 노드가 가진 자식의 수
	트리	트리에서 가장 큰 차수
간선		노드를 연결하는 선
높이		트리에서 가장 값이 큰 깊이



- 표현 방법
  □ N-Link : 각 노드는 최대 N개의 자식을 가짐
  □ Left Child-Right Sibling : 각 노드는 가장 왼쪽 자식과 오른쪽에 위치한 형제의 포인터 주소를 가짐
- 이진 트리(Binary Tree)
  □ 레벨 i에서의 최대 노드 수는 $2^{i-1}(i \geqq 1)$
  □ 깊이가 k일 때 최대 노드 수는 $2^k - 1(k \geqq 1)$
  □ 포화 이진 트리(Full Binary Tree)
    * 리프 노드를 제외한 모든 노드가 2개의 자식을 보유

□ 완전 이진 트리(Complete Binary Tree)

* 리프 노드가 왼쪽부터 모두 쌓여있는 트리



포화 이진 트리 완전 이진 트리 일반 이진 트리

□ 구현

* 배열을 이용

= i번째 노드의 자식은 $i*2, i*2+1$

= 트리의 루트 인덱스는 1

* 연결리스트를 이용

= 구조체 형태로 데이터, 왼쪽. 오른쪽 자식의 링크를 가짐

□ 탐색(Traversal)

* 전위 순회(Preorder) : 자신→왼쪽→오른쪽
* 중위 순회(Inorder) : 왼쪽→자신→오른쪽
* 후위 순회(Postorder) : 왼쪽→오른쪽→자신

- 힙(Heap)

□ 완전 이진 트리(CBT) 구조

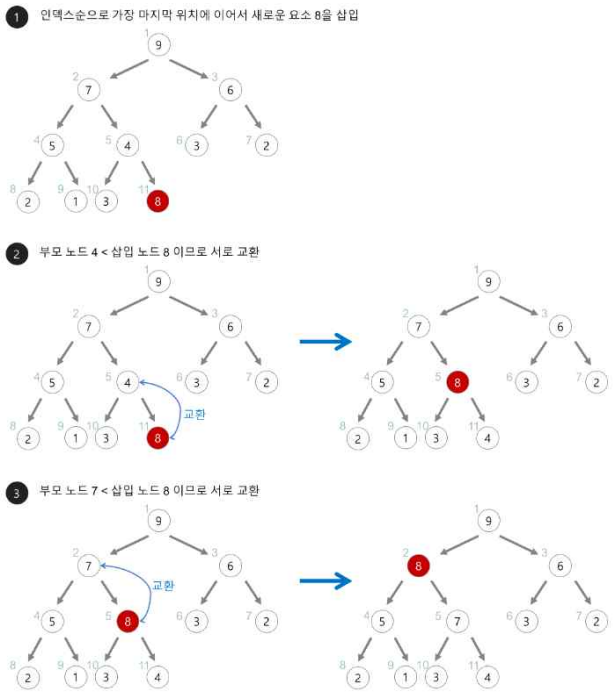
□ Max Heap : 반드시 하위 노드보다 큰 값을 가짐

□ Min Heap : 반드시 하위 노드보다 작은 값을 가짐

□ 우선순위 큐(Priority Queue)를 구현할 때 주로 사용

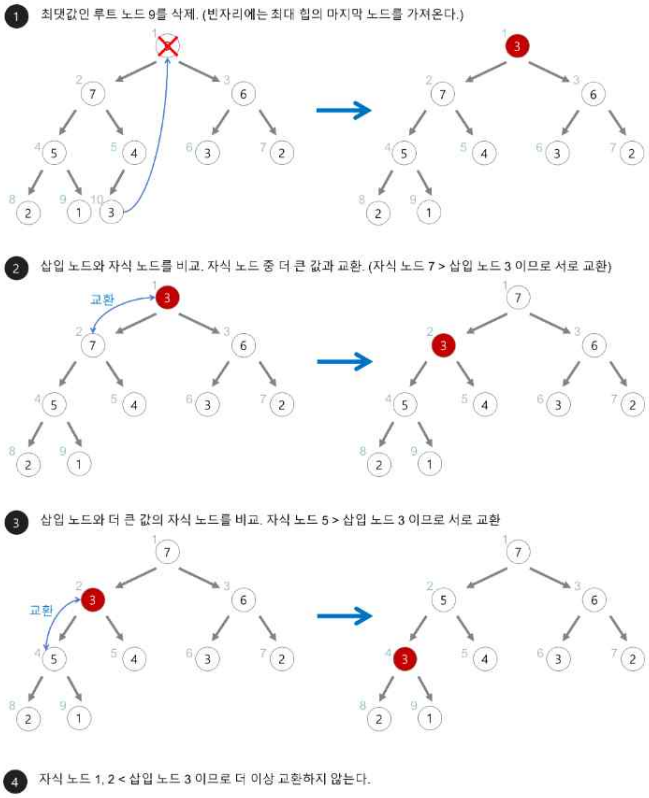
□ 삽입 연산

* 완전 이진 트리를 만족하는 위치에 새로운 노드 삽입
* 부모 노드와 값을 비교하며 값을 변경



□ 삭제 연산

* 루트 노드의 값을 삭제하고, 마지막 노드를 옮김
* 자식 노드와 값을 비교하며 값을 변경



7. 그래프(Graph)

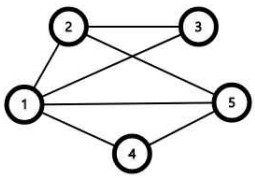
- 정점(Vertex)과 간선(Edge)으로 구성된 집합

이름	의미
정점(노드)	위치를 나타냄
간선	정점을 잇는 선
인접	간선으로 이어진 정점 간의 관계
경로	인접한 정점으로 이루어진 집합
단순 경로	경로에서 반복되는 정점이 없는 경우
싸이클	단순 경로에서 시작 정점과 종료 정점이 같은 경우
오일러 경로	그래프 상의 모든 간선을 한 번씩 방문하는 경로

- 구현

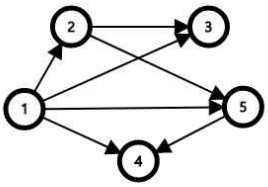
□ 인접 행렬(Adjacency Matrix)

* N개의 정점을 가진 그래프
* (x, y)가 연결되어 있으면 1로, 아니라면 0으로 표기
* (x, x)의 값은 항상 0
* 방향이 없는 그래프는 대각선을 기준으로 대칭
* 사용하는 메모리가 큼(정점의 크기 N → N^2)
* 방향이 없는 그래프



	1	2	3	4	5
1	0	1	1	1	1
2	1	0	1	0	1
3	1	1	0	0	0
4	1	0	0	0	1
5	1	1	0	1	0

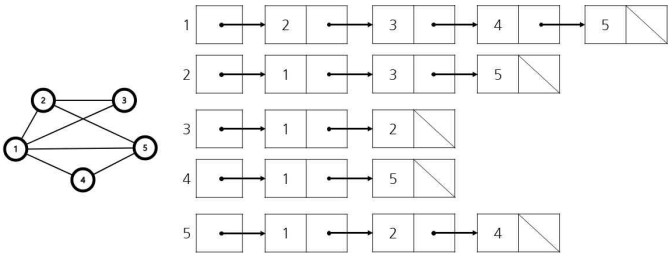
* 방향이 있는 그래프



	1	2	3	4	5
1	0	1	1	1	1
2	0	0	1	0	1
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	1	0

□ 인접 리스트

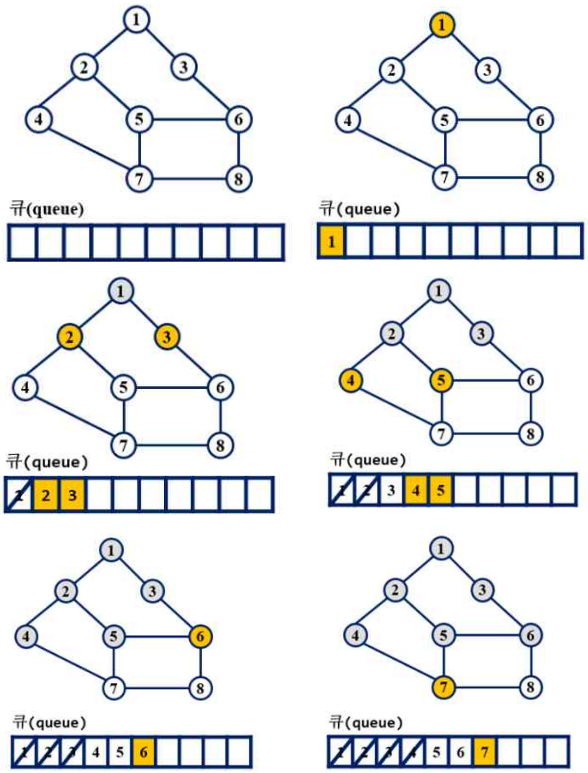
* 해당 정점과 연결된 정점을 연결 리스트에 저장
* 연결 여부를 확인하기 위해 순차 탐색이 필요
* 삽입 연산이 빠르고 메모리 소모가 적음

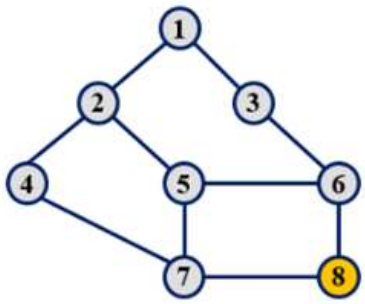


- 그래프 순회

□ 너비 우선 탐색(Breadth First Search)

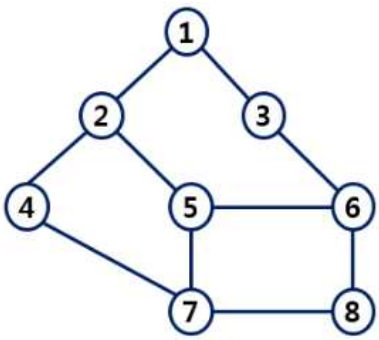
* 같은 깊이에 있는 모든 정점을 방문 후, 다음 깊이를 방문
* 큐를 이용
* 과정
  1. 해당 정점을 방문
  2. 큐에서 해당 정점을 삭제하고, 정점의 이웃 중 방문하지 않은 정점을 큐에 삽입
  3. 큐가 비면 탐색을 종료

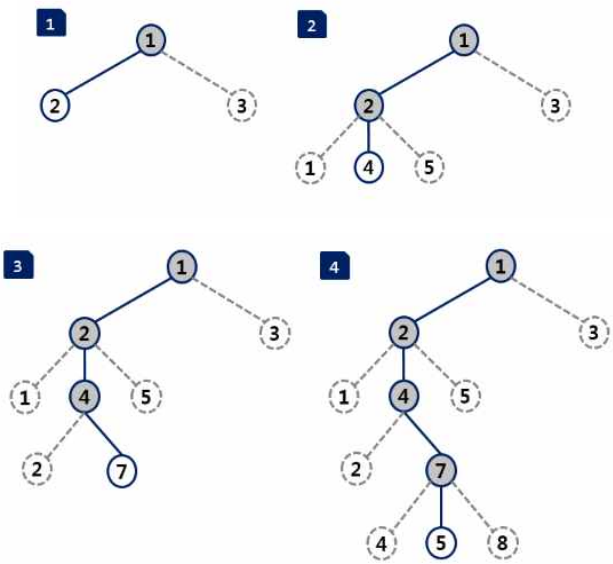


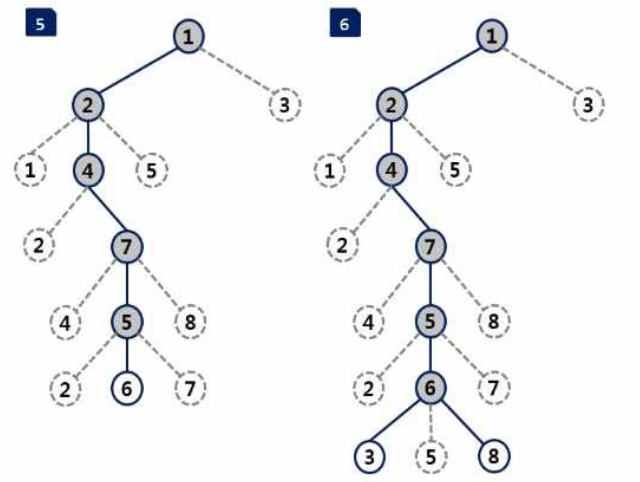


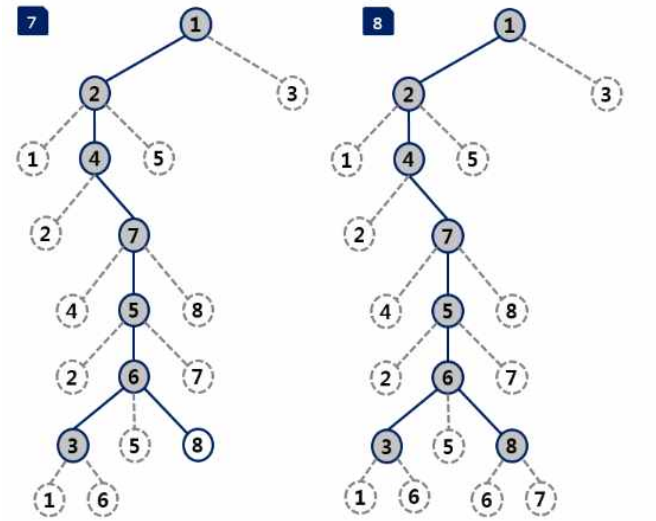
□ 깊이 우선 탐색(Depth First Search)

* 더 나아갈 곳이 없을 때까지 깊이 들어간 후, 다시 돌아와 다른 정점을 방문
* 스택이나 재귀 함수를 이용
* 과정
  1. 해당 정점을 방문
  2. 이웃 정점 중 방문하지 않은 정점으로 이동
  3. 해당 정점에서 방문할 수 있는 정점이 없으면, 이전 정점으로 돌아가 2를 반복
  4. 이전으로 돌아가도 방문할 정점이 없으면 탐색 종료

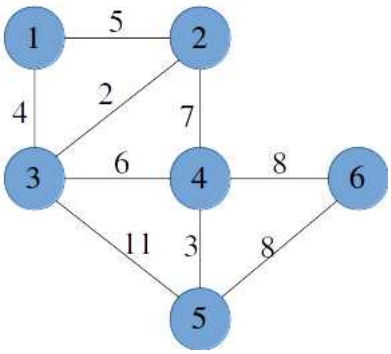


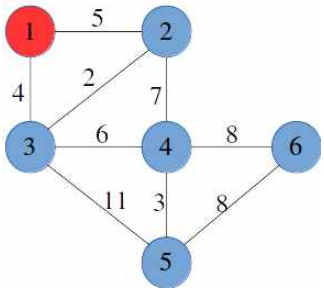




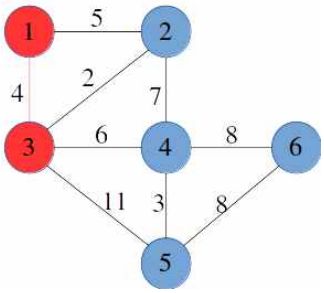


- 최소 신장 트리(Minimum Spanning Tree)
  - □ 그래프의 간선에 가중치(Weight)를 추가
  - □ 그래프에서 싸이클을 형성하는 간선을 제거하면 트리 구조와 같아서 트리라고 이름을 붙임
  - □ 프림 알고리즘(Prim's Algorithm)
    - * 정의
      - = 최소 가중치를 선택하는 것이 어려움
      - = 다음 간선을 선택하는 것에 따라 모양이 다름
    - * 과정
      1. 임의의 정점을 루트로 선택
      2. MST에 속한 모든 정점들에서 가장 낮은 가중치를 가진 간선을 선택하여 추가(싸이클 형성 X)
      3. 모든 정점을 연결할 때까지 반복

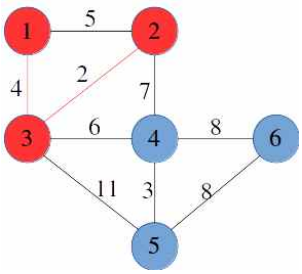




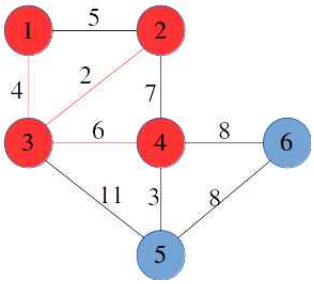
1을 루트로 선택



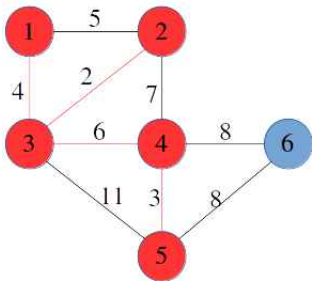
가중치가 최소(4)
노드 3 연결



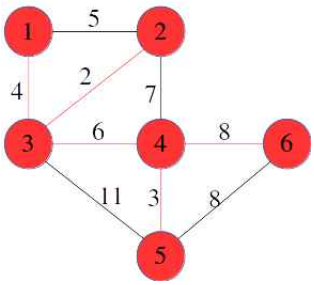
가중치가 최소(2)
노드 2 연결



가중치가 최소(6)
노드 4 연결



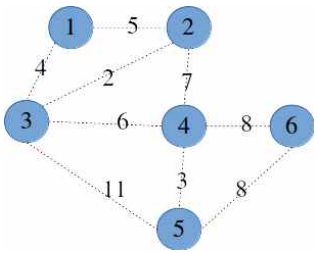
가중치가 최소(3)
노드 5 연결



가중치가 최소(8)
노드 6 연결 // 종료

□ 크루스칼 알고리즘(Kruskal's Algorithm)

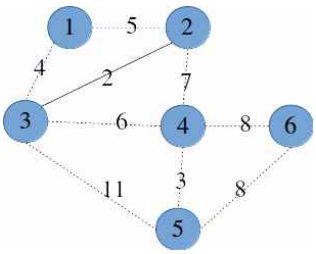
* 정의
  - = 싸이클을 형성하는지 확인하는 것이 어려움 (Union-Find 알고리즘 이용)
* 과정
  1. 그래프 내의 모든 간선을 가중치의 오름차순으로 정렬
  2. 정렬된 간선을 차례대로 선택해, 해당 간선의 정점끼리 연결되어 있지 않다면 MST에 추가 (싸이클 형성 X)



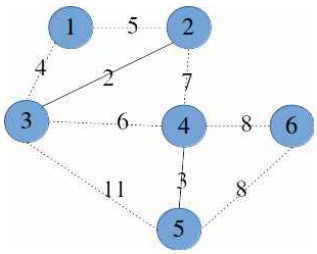
MST 초기화

2,3	4,5	1,3	1,2	3,4	2,4	4,6	5,6	3,5
2	3	4	5	6	7	8	8	11

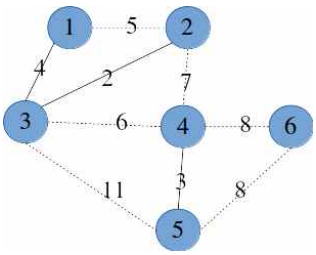
가중치에 따라 정렬



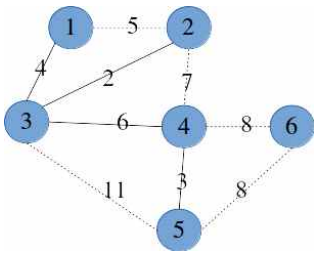
최소 가중치(2) 선택



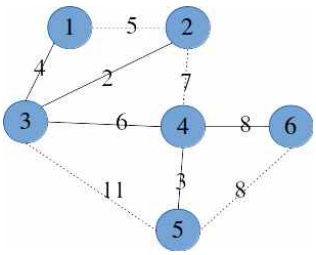
다음 가중치(3) 선택



다음 가중치(4) 선택



1과 2는 연결되어 있으므로
다음 가중치(6) 선택



2와 4는 연결되어 있으므로
다음 가중치(8) 선택 // 종료

□ Union-Find 알고리즘
  * Disjoint set(서로소 집합)을 이용
  * 과정 : (a, b)가 같은 그룹이라면 False 반환, 다른 그룹이라면 a와 b를 같은 그룹으로 만듦

- 위상 정렬(Topological Sort)
  □ 정점 간의 관계를 기준으로 정렬을 수행
  □ 그래프의 방향성이 있어야 하며, 싸이클이 없어야 함(DAG)
  □ a→b의 관계일 경우, a는 b로 이동할 필요조건

□ 정렬 방식
  * 진입 간선의 개수에 따라

1. 진입 간선이 없는 정점 중 하나를 선택, 리스트의 헤드에 추가하고, 해당 정점과 진출 간선을 제거
2. 그래프에 정점이 모두 없어질 때까지 반복

  * DFS를 이용
1. DFS를 이용해, 가장 깊숙이 위치한 정점을 리스트의 마지막에 추가하고, 해당 정점과 진입 간선을 제거
2. 그래프에 정점이 모두 없어질 때까지 반복

8. 정렬(Sorting)

- 버블 정렬(Bubble Sort)
  □ 인접한 두 개의 원소를 비교하여 자리를 교환하는 방식
  □ 정렬이 이루어지는 과정이 거품이 올라오는 것과 비슷해 지어진 이름
  □ 첫 번째 원소부터 마지막 원소까지 반복하며, 한 단계가 끝나면 가장 큰(작은) 원소가 마지막에 위치
  □ 구현이 간단하며, 데이터를 하나씩 비교하기 때문에 정밀한 비교가 가능하지만, 비교 횟수가 많아 성능은 좋지 않음

- 삽입 정렬(Insertion Sort)
  □ 이미 정렬이 되어 있는 부분에 새로운 원소를 삽입하는 방식
  □ 정렬이 되어 있는 부분의 가장 뒤부터 비교하며 위치를 찾음
  □ 버블 정렬의 비교 횟수가 많다는 단점을 보완

- 선택 정렬(Selection Sort)
  □ 전체 원소 중에서 해당 위치에 들어가야 하는 원소를 선택하여 자리를 교환하는 방식
  □ 가장 작은 값을 첫 번째, 두 번째로 작은 값을 두 번째, … 반복
  □ 정렬을 위한 비교 횟수는 많으나, 교환 횟수는 적음
  □ 이미 정렬된 데이터를 역순으로 정렬할 때 가장 적합함

- 합병 정렬(Merge Sort)
  □ Divide and Conquer(분할 정복)을 이용한 정렬
  □ 더 쪼개지지 않을 때까지 분할을 반복

□ 각 부분 집합의 첫 번째 원소부터 비교하며 병합
□ 데이터 전체를 저장할 추가 저장 공간이 필요

- 퀵 정렬(Quick Sort)
□ Divide and Conquer(분할 정복)을 이용한 정렬
□ Merge Sort는 동일한 크기로 분할하지만, Quick Sort는 임의의 크기로 분할
□ 임의의 원소(Pivot)을 기준으로 작은 값을 왼쪽, 큰 값을 오른쪽으로 옮긴 뒤, 각각을 다시 정렬
□ 안정성이 없지만, 속도가 빨라 가장 많이 사용됨

- 힙 정렬(Heap Sort)
□ 주어진 데이터를 이용하여 힙을 만든 후, 삭제 연산을 통해 정렬하는 방식
□ 내림차순은 Max Heap, 오름차순은 Min Heap 이용

- 기수 정렬(Radix Sort)
□ 자릿수에 따라 정렬을 하는 방식
□ 1의 자리, 10의 자리, 100의 자리, ... 의 순으로 각 자리의 값이 같은 원소끼리 부분 집합을 만들어 정렬
□ 정수나 문자열과 같은 특수한 경우에만 사용 가능

- 각 정렬의 특징

Name	Best	Avg	Worst	Space	Stable
Bubble	n^2	n^2	n^2	0	T
Selection	n^2	n^2	n^2	0	T
Insertion	n	n^2	n^2	0	T
Heap	$nlog_2n$	$nlog_2n$	$nlog_2n$	0	F
Merge	$nlog_2n$	$nlog_2n$	$nlog_2n$	n	T
Quick	$nlog_2n$	$nlog_2n$	n^2	$2n+2$	F
Radix	$d \times n$	$d \times n$	$d \times n$	$3d+3$	F

9. 해싱(Hashing)
- 용어
□ 충돌(Collision) : 서로 다른 데이터에 대해 동일한 해시 값을 가짐
□ 클러스터(Cluster) : 특정 범위에 데이터가 몰리는 경우

- 해시 함수(Hash Function)
□ 나눗셈법(Division Method)
* 인덱스 : 입력값 % 테이블의 크기
* 크기는 2의 제곱 수와 거리가 먼 소수를 사용
□ 자릿수 접기
* 각 자릿수를 더해 해시 값을 계산

- 충돌 해결
□ 개방 해싱(Open Hashing)
* 새로운 공간을 할당
* 체이닝(Chaining) : 해시 값에 만들어진 연결 리스트에 데이터를 삽입
= 해시 테이블은 탐색을 효율적으로 하기 위해 만들어졌으나, 리스트 탐색으로 인해 비효율적
□ 폐쇄 해싱(Closed Hashing)
* 주어진 해시 테이블 안에서 해결
= 충돌이 발생하면 새로운 주소를 탐색하여 삽입
* 선형 탐사(Linear Probing)
= 인덱스를 +1씩 더해가며 데이터 삽입
= 클러스터 발생 확률이 높음
= 탐색 시간이 오래 걸림
* 제곱 탐사(Quadratic Probing)
= 충돌이 발생한 횟수에 대해 제곱한 수를 더함
= 2차 클러스터가 발생할 확률이 높음
* 이중 해싱(Double Hashing)
□ 재해싱(Rehashing)
* 해시 테이블에 데이터가 많아질 경우, 테이블을 재할당

10. 탐색(Search)
- 순차 탐색(Sequential Search)
□ 선형 탐색(Linear Search)이라고도 부름
□ 자기 구성 순차 탐색 : 탐색의 효율을 높이기 위해 자주 사용되는 항목을 앞으로 빼내는 방식
* 전진 이동법 : 가장 최근을 제일 앞으로
* 전위법 : 사용될 때마다 한 칸씩 앞으로
* 빈도 계수법 : 사용 횟수에 따라 정렬

- 이진 탐색(Binary Search)
□ 정렬된 데이터에서 사용 가능
□ 탐색 범위를 반씩 줄여나감 → $O(\log_2 n)$
□ 과정
1. 집합의 중간에 있는 값을 선택
2. 값을 비교하여 같으면 종료, 작으면 왼쪽에서 다시 탐색, 크다면 오른쪽에서 탐색
3. 왼쪽 인덱스 > 오른쪽 인덱스라면 데이터가 없는 경우

- 이진 탐색 트리(Binary Search Tree)
□ 왼쪽 자식은 부모보다 작고, 오른쪽 자식은 큼

- 레드 블랙 트리(Red Black Tree)
□ 이진 탐색 트리의 데이터 편향을 해결하기 위해 사용