

Stack

- 쌓아 올린다는 것
- 책을 쌓듯이 쌓아 올린 형태의 자료구조
- 정해진 방향으로 자료쌓기 가능
- top으로 정한 곳을 통해서만 접근 가능
- 삽입하는 연산 => **push** 삭제하는 연산 => **pop**
- 후입선출 구조 (LIFO, Last - In - First - Out)

tmi) 비어있는 스택에서 원소를 추출 => stack underflow

스택이 넘치는 경우 => stack overflow (유명한 '그' 싸이트 이름이 여기서 유래!)

스택의 특징인 후입선출(LIFO)을 활용하여 여러 분야에서 활용 가능하다.

- 웹 브라우저 방문기록 (뒤로 가기) : 가장 나중에 열린 페이지부터 다시 보여준다.
- 역순 문자열 만들기 : 가장 나중에 입력된 문자부터 출력한다.
- 실행 취소 (undo) : 가장 나중에 실행된 것부터 실행을 취소한다.
- 후위 표기법 계산
- 수식의 괄호 검사 (연산자 우선순위 표현을 위한 괄호 검사)

QUEUE

- 줄, 줄을 서서 기다리는 것
- 선입선출(FIFO, First in first out)
- 한쪽 끝에서 삽입 작업이, 다른 쪽 끝에서 삭제 작업이 양쪽으로 이루어짐.
- 삭제연산만 수행되는 곳을 프론트(front), 삽입연산만 이루어지는 곳을 리어(rear)로 정하여 각각의 연산작업만 수행
- QUEUE의 rear에서 이루어지는 삽입연산 => 인큐(enQueue) 삭제연산 => 디큐(dnQueue)
- 큐는 들어올 때 rear로 들어오지만 나올때는 front부터 빠지는 특성
- 접근방법 => 가장 첫 원소와 끝 원소로만 가능
- 가장 먼저 들어온 프론트 원소가 가장 먼저 삭제

큐는 주로 데이터가 입력된 시간 순서대로 처리해야 할 필요가 있는 상황에 이용한다.

- 우선순위가 같은 작업 예약 (프린터의 인쇄 대기열)

- 은행 업무
- 콜센터 고객 대기시간
- 프로세스 관리
- 너비 우선 탐색(BFS, Breadth-First Search) 구현
- 캐시(Cache) 구현

Graph

- 그래프는 노드(Node, 정점 -vertex-)와 노드와 노드를 연결하는 간선(edge)로 구성
- 간선은 방향을 가질수도, 무방향(undirected)일 수 있다
- 진입 차수 진입 차수(in-degree): 방향 그래프에서 외부에서 오는 간선의 수 (내차수 라고도 부름)
- 진출 차수(out-degree): 방향 그래프에서 외부로 향하는 간선의 수 (외차수 라고도 부름)
- 방향 그래프에 있는 정점의 진입 차수 또는 진출 차수의 합 = 방향 그래프의 간선의 수(내차수 + 외차수)

인접 행렬 방식


	a	b	c	d
a	0	0	1	0
b	0	0	1	1
c	1	1	0	0
d	0	1	0	1

abcd는 노드, 1은 간선을 의미한다

- NxN 불린 행렬(Boolean Matrix)로써 $matrix[i][j]$ 가 true라면 $i \rightarrow j$ 로의 간선이 있다는 뜻이다.
- V개의 노드를 표현하기 위해 $V \times V$ 만큼의 크기가 필요하므로 공간복잡도는 $O(V^2)$

✨코플릿 7번 문제 나름 수제 해설

Title



Date

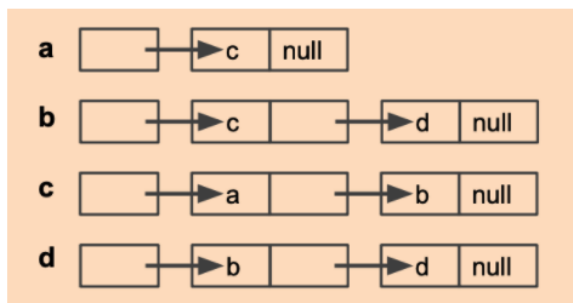
```

matrix      from    to
[0,1,0,0]   0       2
[0,0,1,0]
[0,0,0,1]   *matrix, from, to는 고정값
[0,1,0,0]

// from = 0
let arr = [from]; // arr = [0,1,0,0]
for (let i = 0; i < matrix[from].length; i++) {
  // i < 4
  if (matrix[from][i] == 1 && !arr.includes(i)) {
    // if문 Break 역할
    arr = [...arr, i];
    // arr = [0,1,0,0]
    // arr에 i 값이 있다면 false로 출력
    // 행렬에는 무조건 0,1 존재
    // 만약 0,1 둘다
    // 아래의 if문에 들어간다면 if문이
    // 무한루프.
    i = 0; matrix[from][0] = 0;
    i = 1; matrix[from][1] = 0;
    // if (some(matrix, 1, 2) == 1)
    //   [0,0,0,0]
    return true;
  }
  i = 2; matrix[from][2] = 0;
  i = 3; matrix[from][3] = 0;
}

```

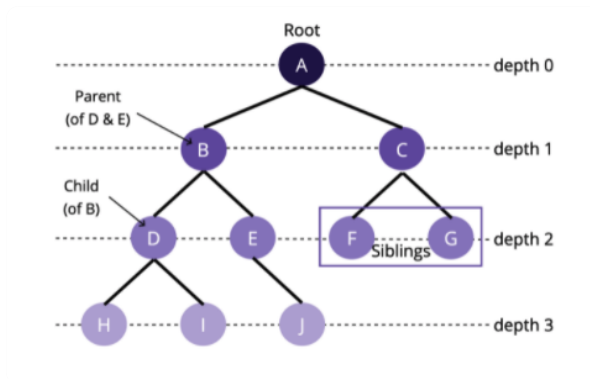
인접 리스트 방식



각 노드에서 간선이 어떤 노드와 연결되어있는지 연결리스트로 표현한다

- 모든 정점(혹은 노드)을 인접 리스트에 저장
- 즉, 각각의 정점에 인접한 정점들을 리스트로 표시한 것
- 무방향 그래프(Undirected Graph)에서 (a, b) 간선은 두 번 저장
- 트리에션 특정 노드(루트 노드)에서 모든 노드로 접근 가능 => Tree 클래스 불필요
- 그래프에션 특정 노드에서 모든 노드에 접근 불가능 => Graph 클래스 필요
- V개의 리스트(노드)에 간선 개수만큼의 원소가 들어있으므로 공간복잡도는 $O(V+E)$
- 일상생활 예시) 지도, 지하철 노선도 최단경로, 전기회로 소자들, 도로 등

Tree



- 노드로 구성된 계층적 자료구조.
- 최상위 노드(루트)를 만들고, 루트 노드의 child를 추가하고,
- 그 child에 또 child를 추가하는 방식으로 트리 구조를 구현할 수 있다.
- 간선(edge) : 노드와 노드를 연결하는 선
- 잎(leaf) : 자식이 없는 (말단)노드 *최하단 노드라는 의미가 아님
- 깊이(depth) : 루트를 기준으로 특정 노드까지의 거리(간선의 개수)
- 높이(height) : 루트를 기준으로 최하단 계층의 leaf 노드까지의 거리
- 레벨(level) : 깊이가 같은 노드끼리의 집합
- 그래프 구현 방식 중 인접 행렬 방식과 인접 리스트 방식의 차이는 무엇인가?
- 트리와 그래프의 차이점은 무엇인가?
- 트리는 방향 그래프, 그래프는 방향, 무방향 모두 가능.
- 트리는 사이클 불가능(비순환 그래프),
- 그래프는 사이클, 자체간선(self-loop)가능, 순환그래프, 비순환그래프 모두 존재.
- 트리는 루트노드 존재, 그래프는 루트노드가 없다.
- 트리는 계층모델, 그래프는 네트워크 모델.
- 트리는 노드가 n 개일 때 항상 $n-1$ 의 간선을 가짐.
- 그래프는 그래프에 따라 간선의 수가 다름. (없을수도 있다)
- 일상생활 예시) 이진트리, 이진탐색 트리, 균형 트리, 이진힙 등

BST

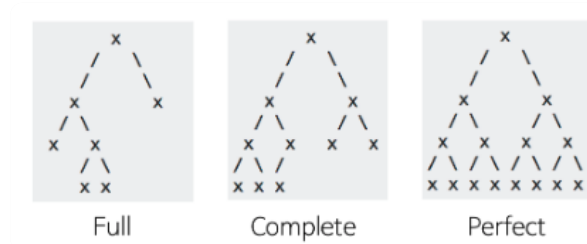
Binary Search Tree

- 이진 탐색 트리는 최대 2개의 자식만을 갖는다.
- 트리구조는 재귀적이므로, 각 자식 노드 역시 2개의 자식만을 갖는다.
- 노드의 값보다 작으면 왼쪽 자식으로,
- 노드의 값보다 크면 오른쪽 자식으로 삽입된다.

- 깊이 우선 탐색은 무엇인가?
- 루트노드에서 시작해서 현재 노드의 첫번째 자식부터, 그리고 그 자식의 자식부터
- 우선적으로 탐색하는 방법.
- 탐색이 끝나면 백트래킹으로 탐색하지 않은 노드를 찾는다.

이진 탐색 트리의 탐색 방법 3가지

- 전위 순회(Preorder Traversal): 부모 → 좌 → 우
- 중위 순회(Inorder Traversal): 좌 → 부모 → 우
- 후위 순회(Postorder Traversal): 좌 → 우 → 부모

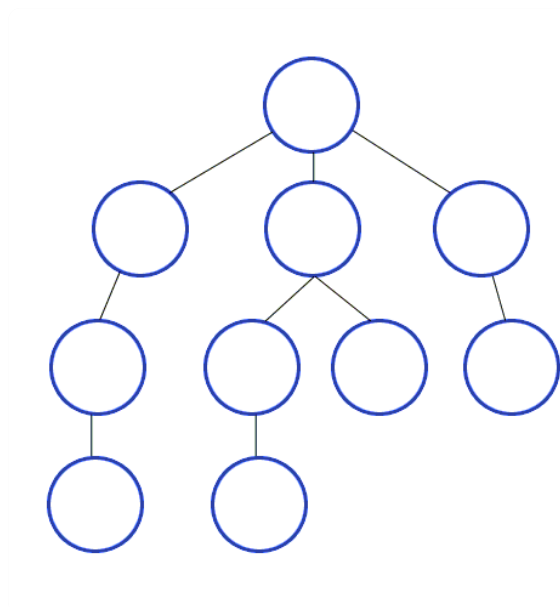


//참고링크

gmlwjd9405.github.io/2018/08/13/data-structure-graph.html

gmlwjd9405.github.io/2018/08/12/data-structure-tree.html

BFS (너비 우선 탐색)



최대한 넓게 이동한 다음, 더 이상 갈 수 없을 때 아래로 이동

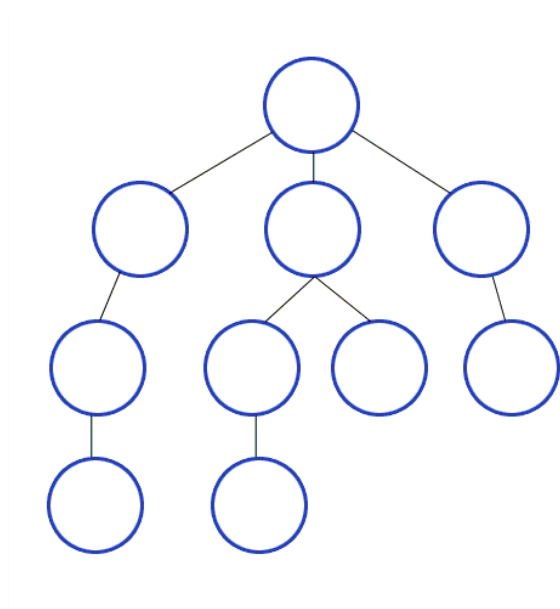
루트 노드(혹은 다른 임의의 노드)에서 시작 → 인접한 노드를 먼저 탐색하는 방법

시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법

주로 두 노드 사이의 최단 경로를 찾고 싶을 때 이 방법을 선택함

- 지구 상에 존재하는 모든 친구 관계를 그래프로 표현한 후 Sam과 Eddie사이에 존재하는 경로를 찾는 경우
 깊이 우선 탐색의 경우 - 모든 친구 관계를 다 살펴봐야 할지도 모름
 너비 우선 탐색의 경우 - Sam과 가까운 관계부터 탐색

DFS (깊이 우선 탐색)

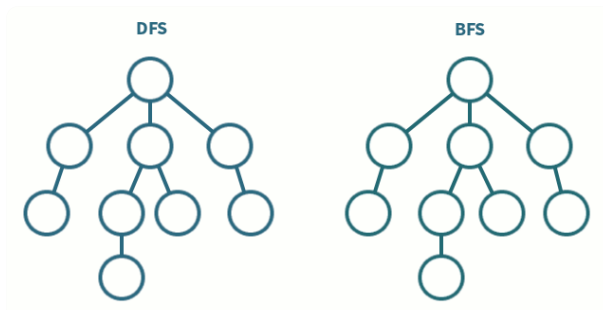


루트 노드(혹은 다른 임의의 노드)에서 시작해서 다음 분기(branch)로 넘어가기 전에 해당 분기를 완벽하게 탐색하는 방식

- 예를 들어, 미로찾기를 할 때 최대한 한 방향으로 갈 수 있을 때까지 쭉 가다가
 더 이상 갈 수 없게 되면 다시 가장 가까운 갈림길로 돌아와서
 그 갈림길부터 다시 다른 방향으로 탐색을 진행하는 것이 깊이 우선 탐색 방식이라고 할 수 있습니다.

1. 모든 노드를 방문하고자 하는 경우에 이 방법을 선택함
2. 깊이 우선 탐색(DFS)이 너비 우선 탐색(BFS)보다 좀 더 간단함
3. 검색 속도 자체는 너비 우선 탐색(BFS)에 비해서 느림

깊이 우선 탐색(DFS) 과 너비 우선 탐색(BFS) 비교



출처 <https://namu.wiki/w/BFS>

DFS(깊이우선탐색)	BFS(너비우선탐색)
현재 정점에서 갈 수 있는 점들까지 들어가면서 탐색	현재 정점에 연결된 가까운 점들부터 탐색
스택 또는 재귀함수로 구현	큐를 이용해서 구현

💡 DFS와 BFS의 시간복잡도

두 방식 모두 조건 내의 모든 노드를 검색한다는 점에서 시간 복잡도는 동일합니다.

DFS와 BFS 둘 다 다음 노드가 방문하였는지를 확인하는 시간과 각 노드를 방문하는 시간을 합하면 됩니다.

N은 노드, E는 간선일 때

인접 리스트 : $O(N+E)$

인접 행렬 : $O(N^2)$

일반적으로 E(간선)의 크기가 N^2 에 비해 상대적으로 적기 때문에

인접 리스트 방식이 효율적임

인접 행렬의 경우, 정점의 개수 N만큼 도는 2중 for문을 돌려 두 정점 간에 간선이 존재하는지를 확인해야 합니다.

이때 N의 제곱만큼 돌게 되므로 $O(N^2)$ 의 시간 복잡도가 됩니다.

인접 리스트로 구현된 경우, 존재하는 간선의 정보만 저장되어 있으므로 인접 행렬에서 사용한 2중 for문이 필요하지 않습니다. 다음 노드가 방문하였는지 확인할 때 간선의 개수인 E의 두 배만큼의 시간이 걸리고(1번에서 2, 6번이 방문하였는지를 확인하고 2번에서 1, 3, 6번을 방문하였는지 확인합니다. 이때 1번과 2번의 간선 하나에 두 번의 확인을 하기 때문에 두배만큼 시간이 걸립니다.) 각 노드를 방문할 때 정점의 개수인 N만큼 걸립니다. 따라서 $O(N+2 \cdot E)$ = $O(N+E)$ 가 됩니다. (시간 복잡도에서 계수는 삭제합니다.)

깊이 우선 탐색(DFS)과 너비 우선 탐색(BFS) 활용한 문제 유형/
응용

1) 그래프의 모든 정점을 방문하는 것이 주요한 문제

단순히 모든 정점을 방문하는 것이 중요한 문제의 경우 DFS, BFS 두 가지 방법 중 어느 것을 사용 하셔도 상관없습니다.

둘 중 편한 것을 사용하시면 됩니다.

2) 경로의 특징을 저장해둬야 하는 문제

예를 들면 각 정점에 숫자가 적혀있고 a부터 b까지 가는 경로를 구하는데 경로에 같은 숫자가 있으면 안 된다는 문제 등, 각각의 경로마다 특징을 저장해둬야 할 때는 DFS를 사용합니다. (BFS는 경로의 특징을 가지지 못합니다)

3) 최단거리 구해야 하는 문제

미로 찾기 등 최단거리를 구해야 할 경우, BFS가 유리합니다.

왜냐하면 깊이 우선 탐색으로 경로를 검색할 경우 처음으로 발견되는 해답이 최단거리가 아닐 수 있지만,

너비 우선 탐색으로 현재 노드에서 가까운 곳부터 찾기 때문에 경로를 탐색 시 먼저 찾아지는 해답이 곧 최단거리기 때문입니다.

이밖에도

- 검색 대상 그래프가 정말 크다면 DFS를 고려

- 검색대상의 규모가 크지 않고, 검색 시작 지점으로부터 원하는 대상이 별로 멀지 않다면 BFS

출처: <https://devuna.tistory.com/32> [튜나 개발일기]