

# TREES

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 4 . 18

주 편시 주의할 점

④ splitting      null을 연산하고 있는 것은 아닌지 확인해보기 / parent가 중심이 되어 연산 진행

- 주어진 key 값을 기준으로 2개의 binary search tree로 split. ( tlt: bigger than given key, ltl: smaller than given key )

[ 주어진 key보다 cur key가 크면 → 그 노드의 right subtree로 second tree ]

[ 주어진 key 뒷 cur key가 작으면 → 그 노드의 left subtree로 first tree ]

- time complexity

## Iterators

### Binary Tree

① postorder Iterator

```
class PostIter {
    Stack s;
    public PostIter(Node r) {
        s = new Stack();
        pushNextDesc(r);
    }
    public void pushNextDesc(Node n) {
        while (n != null) { // 가장 끝 node는 마지막 노드의 children 처리
            s.push(n);
            if (n.left == null) n = n.right; else n = n.left;
        }
    }
    public int getData() {
        if (atEnd()) return -1;
        return s.peek().data;
    }
    public void next() {
        if (atEnd()) return;
        Node prev = s.pop();
        if (!s.isEmpty()) { // 만약 둘 다 끝나면 맨 끝값 빼고 다시 첫 노드를 전달
            if (s.peek().left == prev) pushNextDesc(s.peek().right);
        }
    }
    public boolean atEnd() {
        return s.isEmpty();
    }
}
```

② inorder iterator

```
class InIter {
    Stack s;
    public int getData() {
        if (atEnd()) return -1;
        return s.peek().data;
    }
    public boolean atEnd() {
        return s.isEmpty();
    }
    public void pushLeft(Node n) {
        while (n != null) {
            s.push(n);
            n = n.left;
        } // 각각 node만 찾으ing
    }
    public InIter(Node r) {
        s = new Stack();
        pushLeft(r);
    }
    public void next() {
        if (atEnd()) return;
        Node n = s.pop();
        if (n.right != null) pushLeft(n.right);
    }
}
```

③ preorder iterator

```
class PreIter {
    Stack s;
    public int getData() {
        if (atEnd()) return -1;
        return s.peek().data;
    }
    public boolean atEnd() {
        return s.isEmpty();
    }
    public PreIter(Node r) {
        s = new Stack();
        if (r != null) s.push(r);
    }
    public void next() {
        if (atEnd()) return;
        Node cur = s.pop();
        if (cur.right != null) s.push(cur.right);
        if (cur.left != null) s.push(cur.left);
    }
}
```

### Level order traversal

: 각 노드를 레벨 순으로 순회.

→ 이전의 traversal 과의 차이: queue 사용. ( 이런 방법들은 stack 사용)

#### pseudo code

```
initialize queue
queue.enqueue(root);
while queue != empty
    n ← queue.dequeue();
    if (n!=NULL)
        print n.data
        queue.enqueue(leftchild);
        queue.enqueue(rightchild);
```

# HEAPS

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

## Priority Queues

- operation : insert, query max, delete max, query min, delete min

### Heaps

- complete binary tree
- children의 key 값보다 parent의 key 값이 더 크다.
- array representation 사용.
  - : complete이 아니면 array를 할당해놓고 만드는 공간이 더 큼

#### Basic heap operations

##### ① Create a heap : allocating a new array

( 가장 : max # of array가 정해짐 )

- time complexity :  $O(1)$

##### ② inserting a new item

- 제일 마지막에 삽입하고 힙의 성질을 만족하기 위해 성립될 때까지 bottom-up 실행

- time complexity : insert  $O(1)$  + bottom-up  $O(h) \Rightarrow O(n)$

##### ③ Querying the maximum └ root of the heap

- time complexity : finding  $O(1)$

##### ④ deleting the maximum

- 제일 마지막 노드를 가져와서 root에 넣은 뒤 top-down Heapify 진행

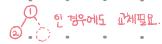
#### \* Heapify?

한 노드의 두 subtree가 모두 Heap일 때 root node도 힙의 성질을 지킬 수 있도록 swap

How? children 둘 중에 적다면 힙과 표준.

- time complexity : insert  $O(1)$  + heapify  $O(h) \Rightarrow O(n)$

#### \* 구현할 때 주의!!



Index 부여가능지 확인

##### ⑤ union : 두개의 heap을 하나의 heap으로 합침

→ 배열을 합치고 buildHeap 연보드함  
└ 자료구조 구현으로 관계성 확보(우선순위)

##### ⑥ delete

[ 제거 대상 element가 비교로 property 있는 맨션 필요 (heapify / bottom-up)]

decreaseKey to -∞ delete minimum

## Heap Sort

### ① Heapify from the bottom

- Heapify를 적용시키려면 subtree Heap이라 하므로 bottom부터 실행

- leaf node는 실행해도 바로 리턴되므로 마지막 노드의 parent 브터 Heapify 진행

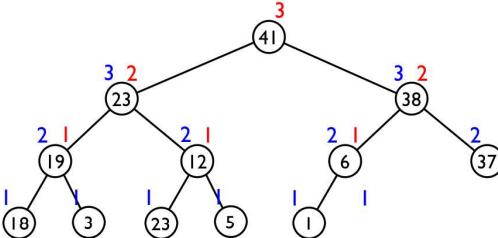
### ② repeatedly delete the maximum

- Heap의 맨 위에 하나의 delete maximum 수행 후 array에 남는 값은 오름차순 정렬되어 있음.

- time complexity :  $\frac{n}{2} \lceil \log_2 n \rceil = O(n \log n) + \text{heapify } O(n) \Rightarrow O(n \log n)$

\* Heapify는 왜  $O(n)$ ?

$S$  = Heapify 연산 수행 시간이라 하면



- 파란색 숫자들의 합 =  $S$

- 빨간색 숫자들의 합 = 25

$$S = 25 - S = (\text{root노드의 } 2S - S) + (\text{다른 노드들의 } 2S - S)$$

└ 드리의 노이

$$= \log_2 n + 2 \cdot \lceil \frac{n}{2} \rceil \Rightarrow O(n)$$

# GRAPH

Curve of forgetting study planner

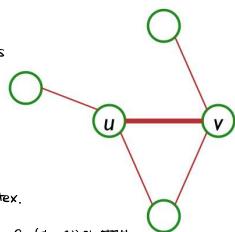
1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

## Graphs

### Terminology

- Graph  $G = (V, E)$
- $V$ : Vertices (singular: vertex)  $\rightarrow V(G)$ : set of vertices
- $E$ : Edges  $\rightarrow E(G)$ : set of edges
- 그래프에서 edge로 연결되어 있으면 adjacent하다.
- ex)  $u$ 와  $v$  adjacent. edge  $(u, v)$ 로 연결되어 있다.
- cycle: a path starts and ends at the same vertex.
- path: length  $n$  from  $u$  to  $v \Rightarrow e_1 = (u, x_1), e_2 = (x_1, x_2) \dots e_n = (x_{n-1}, v)$ 의 연결선.
- simple path: vertex 중복되지 않는 path.
- simple cycle: 길이 2 이상 vertex 중복이 아닌 cycle.
- acyclic: simple cycle이 없는 graph.
- degree of vertex: incident한 edge #
- Subgraph:  $G' = (V'; E')$  일 때  $V' \subseteq V$ 이고,  $E' \subseteq E$  일 때  $G'$ 은  $G$ 의 subgraph



- vertex-induced subgraph: specify vertex first  $\rightarrow$  연결된 all edges 포함
- edge-induced subgraph: specify edge first  $\rightarrow$  연결된 all vertices 포함
- weighed graph: each edge가 numerical weight로 annotated 되어 있는 graph
- connected graph: every pair of vertices on path이 있는 graph.  
↳ unconnected 일 때 그 subgraph들은 각각 connected component.

### Tree & Graph

- tree: an acyclic connected graph.

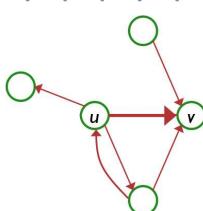
- [rooted tree]
- [unrooted tree]

- X tree / graph의 차이 (degree)

- [tree: children 수]

- [graph: vertex와 연결된 edge 수]

- spanning tree: Graph  $G$ 의 모든 vertices를 모두 포함하는 subgraph.



### Directed Graphs

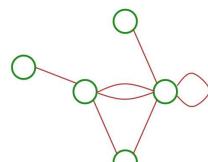
- directed graph (digraphs): edge는 ordered pair of vertices
- adjacent from/to:  $<u, v>$  ( $u$  to  $v$ ),  $<v, u>$  ( $v$  to  $u$ )  $\rightarrow$  head to tail
- degree

  - [in-degree: vertex  $v$ 에 들어오는 방향으로의 edge 수]
  - [out-degree: vertex  $v$  들어가는 방향으로의 edge 수]

- strongly-connected:  $u \rightarrow v, v \rightarrow u$ 의 direct pattern 존재 하는 경우.
- underlying graph: directed graph의 orientation을 무시한 undirected graph.

### Multigraph & pseudograph

- multigraph: 같은 pair of vertices 사이로 edge가 여러개인 경우.
- loop: an edge to itself
- pseudograph: loop를 사용할 수 있는 multigraph.



### Internal representation

#### Adjacency matrix (인접행렬)

만약 행렬  $(i, j)$  edge가 존재하면  $A[i][j] = 1$ , 존재하지 않으면  $A[i][j] = 0$   
인 경우

weighted graph의 경우 edge에 special value 있으면 weight 저장

#### Adjacency List (인접리스트)

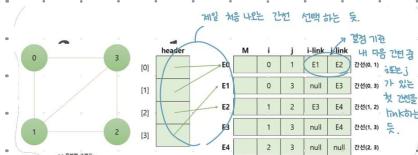
한번의 vertex에서 다른 vertex들을 전부 노드에 저장하는 linked list로 연결하고 이를 하나의 배열에 저장

weighted graph의 경우 weight도 같이 노드에 저장하는지

inverse adjacency list: store edges to vertex  $i$

부록: duplicate copies of each edge

#### Adjacency multilist



간선을 노드마다 linked list로 만들 / 숨기고 잘 쓰자 봄음

### Graph Search

#### Depth first search (DFS)

adjacent vertex에 recursive한 traversal

- ① vertex 0  $\rightarrow$  vertex 1
- ② vertex 1  $\rightarrow$  vertex 3
- ③ vertex 3  $\rightarrow$  vertex 4
- ④ vertex 1  $\rightarrow$  vertex 4 (already visited.)

DFS를 수행하면 all connected component 찾을 수 있음

running time:  $O(|V|^2)$ ,  $O(|V|+|E|)$

#### Breadth first search (BFS)

queue 사용

enqueue를 할 때 visiting vertex 추가

visit vertices in nondecreasing order of shortest path lengths from  $v$

running time:  $O(|V||E|)$

### Minimum Spanning Trees

Weighted graph의 total edge cost를 최소화하는 spanning tree

\*  $G$  is connected, acyclic 이면  $|E|=|V|-1$

#### Key Lemma

Graph  $G$ 의 nonempty SubSet인  $S$ 가 있을 때 edge  $e$ 를  $S$ 와  $S^c$ 의 endpoint,  $S^c$ 에 다른 endpoint로 가는 edge 중 minimum cost edge이면 every MST는  $e$ 를 포함한다.

증명) MST에 포함되는 edge는 다른 MST와는 차이가  $\neq$   $\infty$ 인 MST ( $T'$ )

$\rightarrow T'$ 은 비단을 통해 증명

#### Kruskal's algorithm

empty graph  $T$ 를 초기화

graph  $G$ 의 edges를 nondecreasing order로 정렬한 edge를 포함한 이후 cycles가 있는다면 허용하지 않음

$|V|-1$ 개의 edge가 되면 종료

모든 iteration을 통과해 edge 수가  $|V|-1$ 이거나 적어도 2개인 MST가 존재하지 않는 것을 보여준다. (edge cost distinct)

i) cycle 체크를 does/break로 한다면:  $O(|E| \log |E| + |E||V|) = O(|E||V|)$

ii) cycle 체크를 union-find로 한다면: endpoint 2개의 find연산으로 same subset이면 cycle 형성

$\rightarrow O(|E| \log |E| + |E||V|) = O(|E||V|)$  : no improvement

iii) cycle 체크를 weighted union-find로 한다면

$\rightarrow O(|V| + (|V| + |E|) + |E| \log |E| + |E| \log |V|)$

$\rightarrow O(|V| + |E| \log |V|)$

$\rightarrow O(|E| \log |V|)$  if  $G$  is connected

# GRAPH

Curve of forgetting study planner

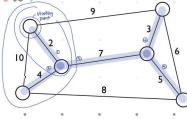
1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

## Prim's algorithm

$V(T) \cup V(G) \setminus V(T)$  사이의 edge 중 minimum cost edge를 테이블에

이용해 편해는 edge 경로까지 알면  $n$ 는 disconnected이다.



T의 edge 끝부분은  $|V|-1$ 의 원형 구조로 분포

(= Key Lemma 그대로)

- running time :  $O(|V|(|V|+|E|)) \Rightarrow O(|V|^2|E|)$

### [Implementation]

- using a heap (of edges) : min heap of edges cost

: 해당 vertex 쪽으로 도착하는 candidate가 되면 adjacent edges를 heap에 insert하고 그에 minimum cost edge를 찾아서

### Algorithm

여기서 문제는?

만약 candidate가 되었으면, 다른 candidate와 차이를 하거나 heap에 minimum 관계를 두었으면 이를 delete 해야 한다는 점이다.

$\Rightarrow$  ghost edge로 남겨두고 minimum에 맞을 때 candidate가 아니면 그때 지워주는거임.

- running time :  $O(|V|(|V|+|E|)\log |E|) \Rightarrow O(|E|\log |V|)$  : connected

## Shortest Path

### Dijkstra's algorithm

Weighted (di)graph 이고 모든 edges weight가 nonnegative일 때 vertex set에서 vertex를 shortest path를 찾는다

### [Key observation]

vertex  $s$ 에서 path의 second-to-last vertex를  $a$ 로 하면 shortest path의  $a \rightarrow t$ 를 하면 shortest path이다

### \* Dijkstra's algorithm 풀이

set  $S$  : shortest paths of vertices의 집합

$d[V]$  : shortest path from  $s$  to  $v$  ( $d[s] = 0$ ) // 실제 shortest path는  $s$ 와  $S$ 에 있는 vertex들 중 2nd-to-last를 정한 것을 대

의 shortest path // direct path가 있는 경우 바로 초기화

$prev[v]$  : shortest path의 2nd-to-last vertex

$d[v]$  : 처음 찾은 vertex를 업데이트해  $s$ 에 포함된 가중치 (Singleton 처리) 간접 : 이미 세 번째 vertex로 업데이트해버린  $d[v]$ 를

Update 새롭게 (4번 add는 vertex) 2nd-to-end vertex의 경우와 함께  $d[v]$ 를 업데이트

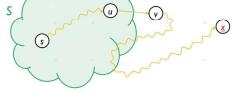
> If  $d[x] = \min_{v \in S} d[v]$ ,  $d[x]$  is shortest length of an  $s-x$  path

> Let  $P$  be an arbitrary path from  $s$  to  $x$ :

$P$  has to "cross" the boundary from  $S$  to  $V \setminus S$ :

let  $s \in P$  be the first arc on the boundary

> (length of the  $s-v$  prefix of  $P$ )  $\geq d[v] \geq d[x]$



이것은 infinity boolean flag를 사용하는 safest한 방법으로 이해하여 사용

- running time :  $O(|V|^2)$

path는 찾았을 때마다 찾을 수 있음

### Min-Heap

decreasekey operation 추가.

key값을 주면 값으로 바꾸고 property 만족할 때까지 parent와 swap 진행 / root 될 때까지

- running time :  $O(\log n)$

### Improvement

< Prim : min heap of edges : cost x change

< Dijkstra : min heap of vertices,  $d[v]$  : value can change (decrease only.)

$d[V]$ 으로 minheap을 만들고,  $d[V]$ 가 첫 번째 vertex를 최종 목적지로 각각

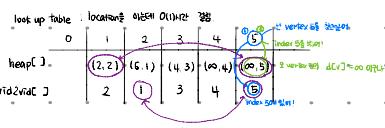
decreasekey operation으로  $d[V]$ 를 update 진행

$d[V]$ 의 최솟값 찾는 연산 : deleteMin 연산  $|V|$ 번

$d[V]$  update : decreasekey 연산  $|E|$ 번

$\Rightarrow$  최종 : decreasekey를 하면서 vertex의 index를 큐로 → search entire heap 할 때

: need to maintain index of vertex in heap



$\Rightarrow$  running time :  $O((|V|+|E|)\log |V|)$ ,  $O(|E|\log |V|)$  weakly connected

## Topological Sort

directed graph에서 전체 vertex를 방문하는 순서를 결정

→ pre-registered 혹은 vertex를 찾아서 미리나감 / empty일 때 더해지

But :acyclic인 Order를 찾을 수 없음

Directed Acyclic Graphs (DAGs)에서 topological order가 가능함

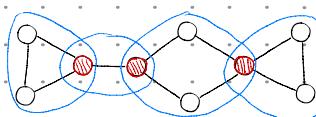
## Biconnected graphs

- articulation point (cut vertex) : 지나가는 때 graph를 끊어버리는 vertex

- biconnected graph : articulation point가 없는 graph

- biconnected component : maximal biconnected subgraph

additional vertex 추가 시 still biconnected인 vertices가 있음을



### DFS Tree

- Tree edges : DFS를 거쳐가는 edge

- Back edge : DFS를 거쳐가는 edge

- Cross edge : directed graph에서의 edge

### Articulation Point 찾기

-  $dfn[v]$  : vertex  $v$ 의 DFS 번호

-  $low[v] = \min\{dfn[v], low[z] \mid z \in v \text{ child}, dfn[z] < v\}$

root node (nil) 2nd to last, non-root node :  $low[v] \geq dfn[v]$  :  $v$ 는 child로 cut vertex

## Finding biconnected components

The main routine calls `findBC(root, dummy)`, with `dfn[]` initialized as zero

procedure `findBC(v, parent)`

set `dfn[v] = visit this vertex` (visit number)

`low[v] = dfn[v]` (lowest visit number)

for each incident edge  $(v, x)$  do

if  $x = \text{parent}$  or  $dfn[v] < dfn[x]$  then continue

push  $(v, x)$  (edge stack) (parent edge)

if  $x$  has been visited then

update `low[v]` to `low[x]` (parent edge)

else (new edge) (parent edge)

findBC( $x, v$ ) (parent edge)

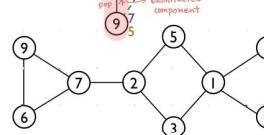
update `low[v]` to `low[x]` (parent edge)

if  $low[x] \geq dfn[v]$  then

repeat

pop and output an edge

until the edge is  $(v, x)$



# UNION-FIND

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

## Union-Find

maintains family of disjoint sets  $S_i$

### Operations

- Union ( $i, j$ ) :  $S_i, S_j$ 를 합하는 연산 (전체조선, 루트의 값을 알고 있어야 함 : root node value)
- find( $x$ ) :  $x$ 을 포함하는 set의 root를 찾아서 return (전체조선 :  $x$ 을 포함하는 set은 현재도 유일함)

## Representation (tree는 pointer로 구현)

Union : 다른 tree를 끌어온 tree의 subtree로 만듬

-  $O(1)$  time

- i,j를 모르는 경우, 그는 내부 component를 증명해 Find연산 후 실행해야 함

### Find

- 주어진 값이 포함된 tree의 root 리턴

-  $O(n)$  time

### Internal representation

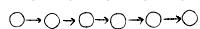
각자 자신의 parent를 가지고 있어야 함 (→비정선인)

parent가 없으면 -1로 선언 initialization도 모두 -1로 선언

Initialization :  $O(n)$  time

## Weighted Union

### Worst case of Union-Find



### Representation

- Union : 원 union 연산과 동일하지만, 더 작은 cardinality의 tree를 subtree로 만든다.

Cardinality update 필요함

- Find : 원 연산과 동일

### Internal representation

- parent 저장 array

- cardinality 저장 array : initialize by -1

### Improvement

maximum height of a tree :  $\lceil \log_2 k \rceil$  ( $k$ : cardinality)

proof) ①  $k=1$  일 때  $h=0$  :  $\lceil \log_2 1 \rceil = 0$

②  $k \geq 2$  일 때 union 전의 상태를 보면  $k=k_1+k_2$ ,  $k_1 < k_2$

$$\begin{aligned} \lceil \log_2 k \rceil &\geq \left\lceil \log_2 k_1 \right\rceil + \left\lceil \log_2 k_2 \right\rceil \\ \lceil \log_2 \frac{k}{2} \rceil &\geq \left\lceil \log_2 k_1 \right\rceil \end{aligned}$$

$$\lceil \log_2 k \rceil - 1 = \text{overall union 이후 } k_1 \text{에 있어 height 증가} : \lceil \log_2 k \rceil - 1 + 1 = \lceil \log_2 k_2 \rceil$$

union 이후  $k_2$ 에 있어 height 증가 :  $\lceil \log_2 k_2 \rceil$

(maximum height of subtree which cardinality is  $k$ ) :  $\lceil \log_2 k \rceil$

$\Rightarrow$  Find연산 수행시간 :  $O(\log n)$

## Improvement

### path compression

find연산을 반복적으로 호출해 하면 비효율적.  $O(n \log n)$  빈번

remember하기 [ ① storing : examine 시간 더 걸릴지도

[ ② change the node to the child of its root : 자리는 모든 node에 대해서

같은 연산 수행

- find연산 진행하면서 O(1) 연산 수행 :  $O(n \log n)$  빈번 X