

Algorithm HW3

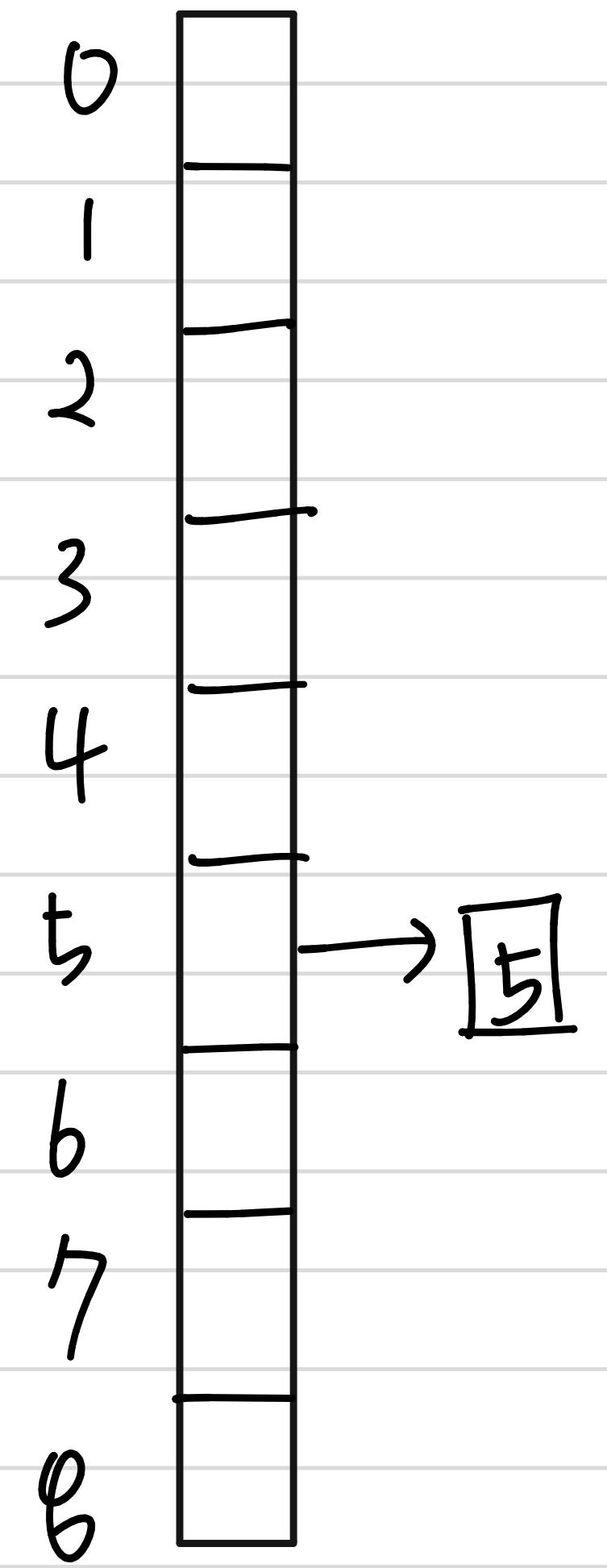
- Mechanical Engineering
- 2019-15638 주기영

#1. (교재 11.2-2)

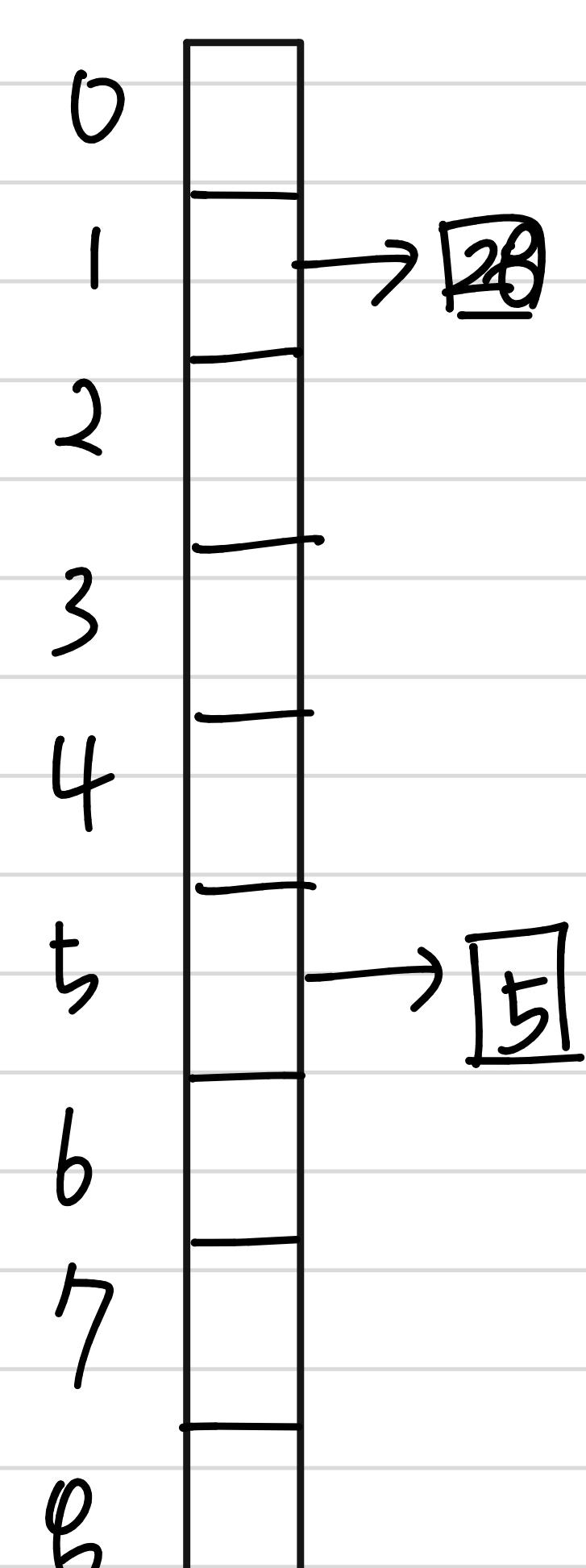
X chaining 방식에 의한 hash table 흐름 해설

각 slot에 Linked List가 존재하여 흐름의 일정을 통해 slot에서 Linked List에 삽입한다. 이때 Linked List의 맨 뒤에 삽입하는 것은 List 크기 n에 대해 시간복잡도가 O(n)이므로, head에 삽입하여 O(1)의 시간복잡도를 확보

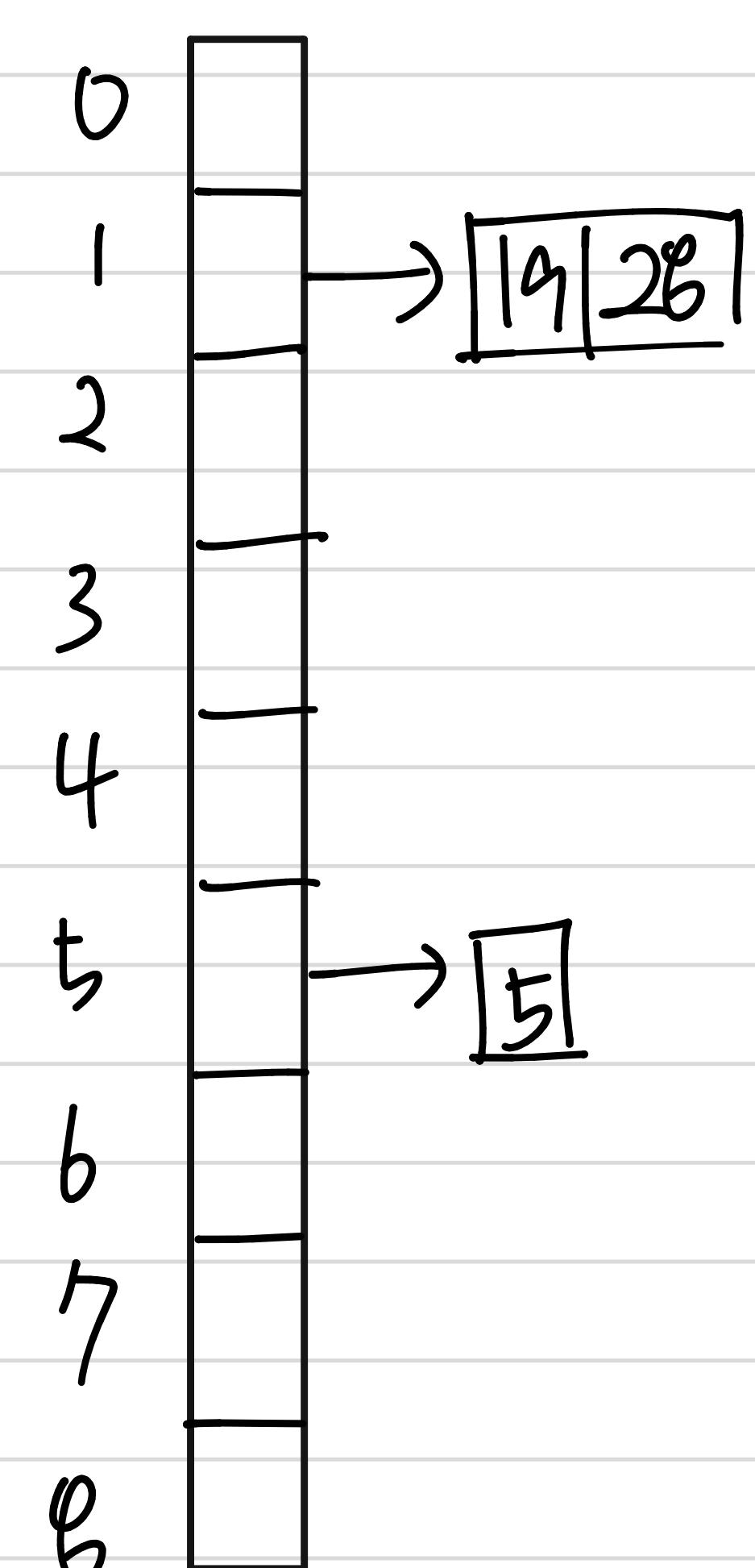
① 5 삽입 ($5\%9=5$)



② 28 삽입 ($28\%9=1$)

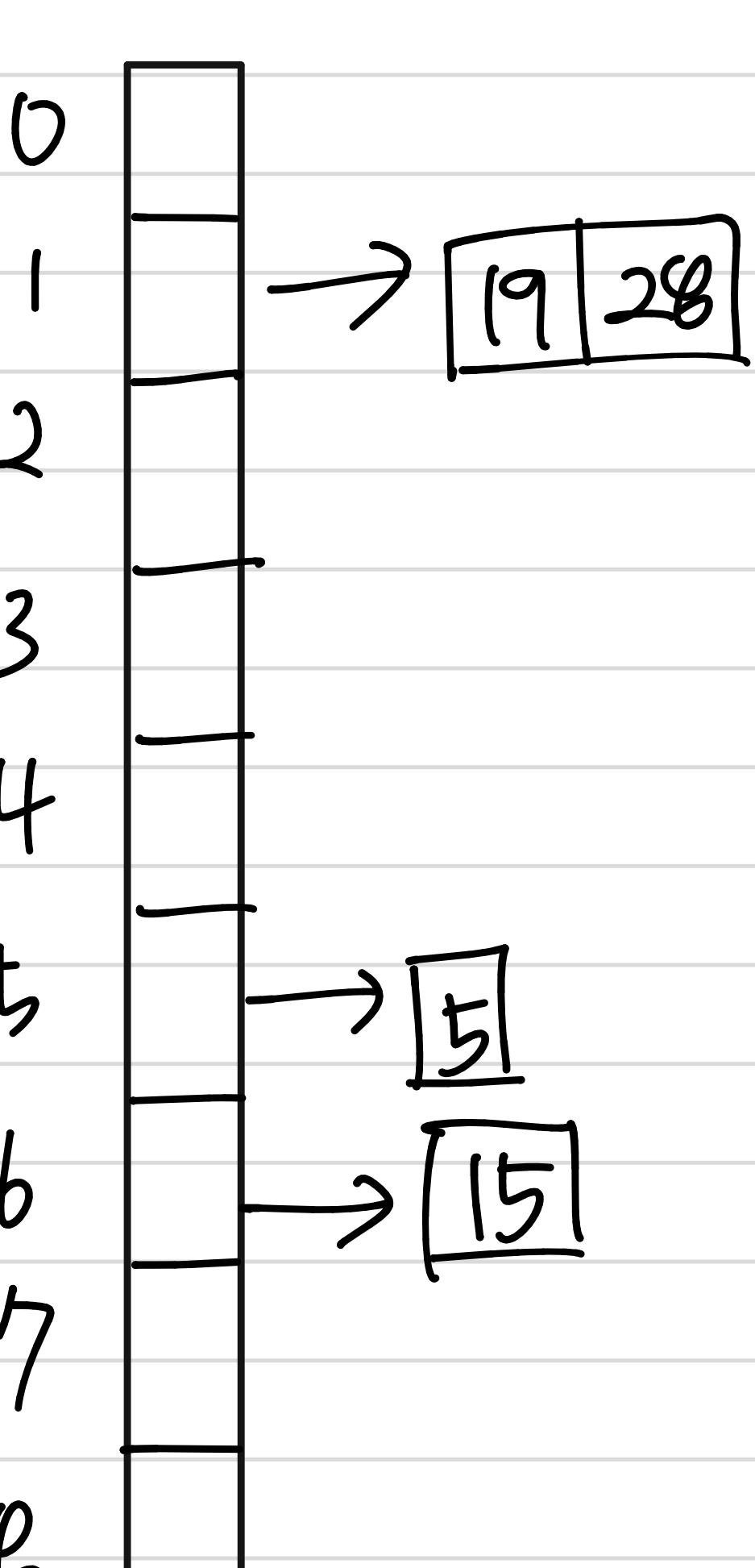


③ 19 삽입 ($19\%9=1$)



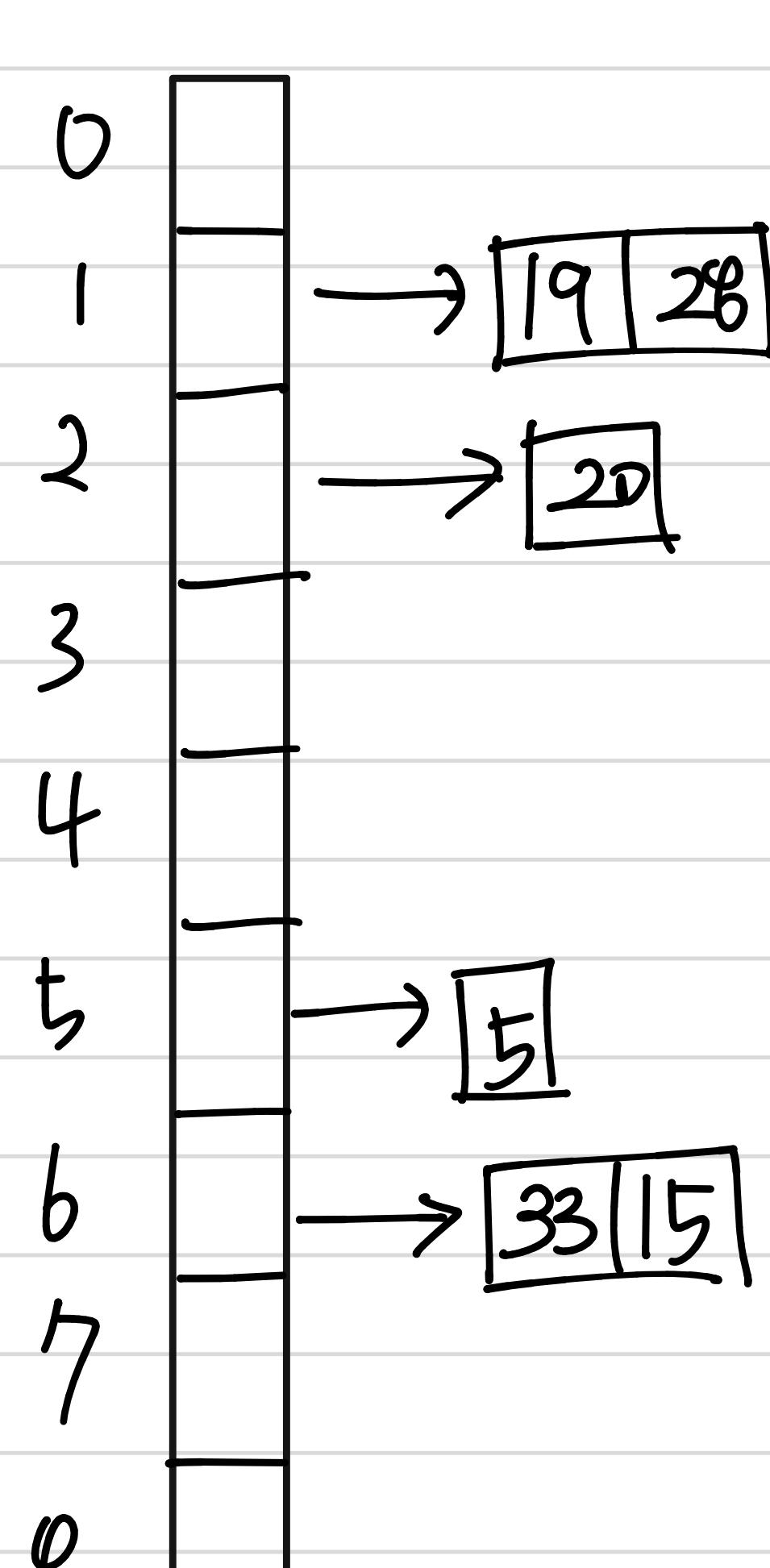
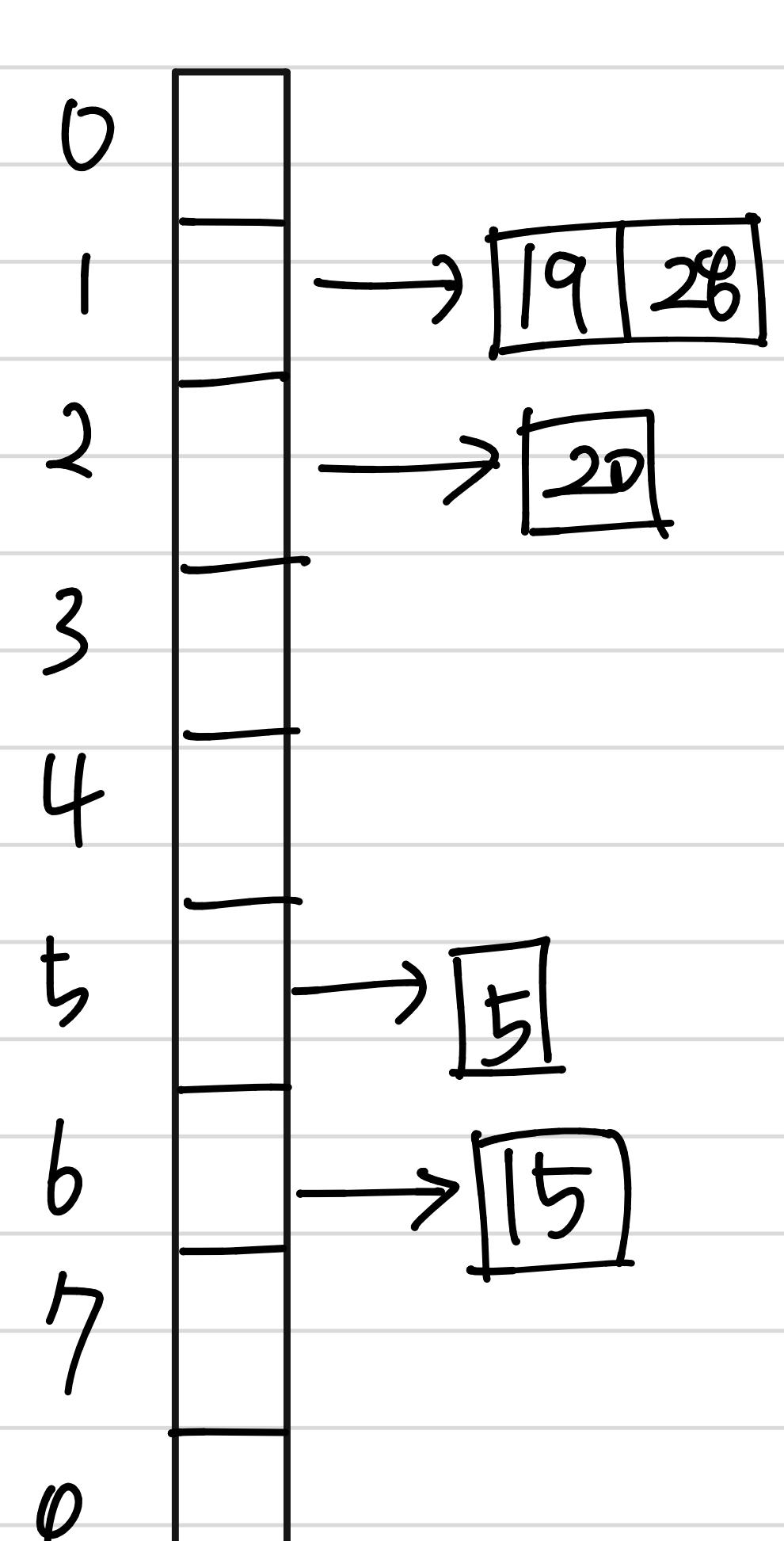
④ 15 삽입 ($15\%9=6$)

④ 15 삽입 ($15\%9=6$)

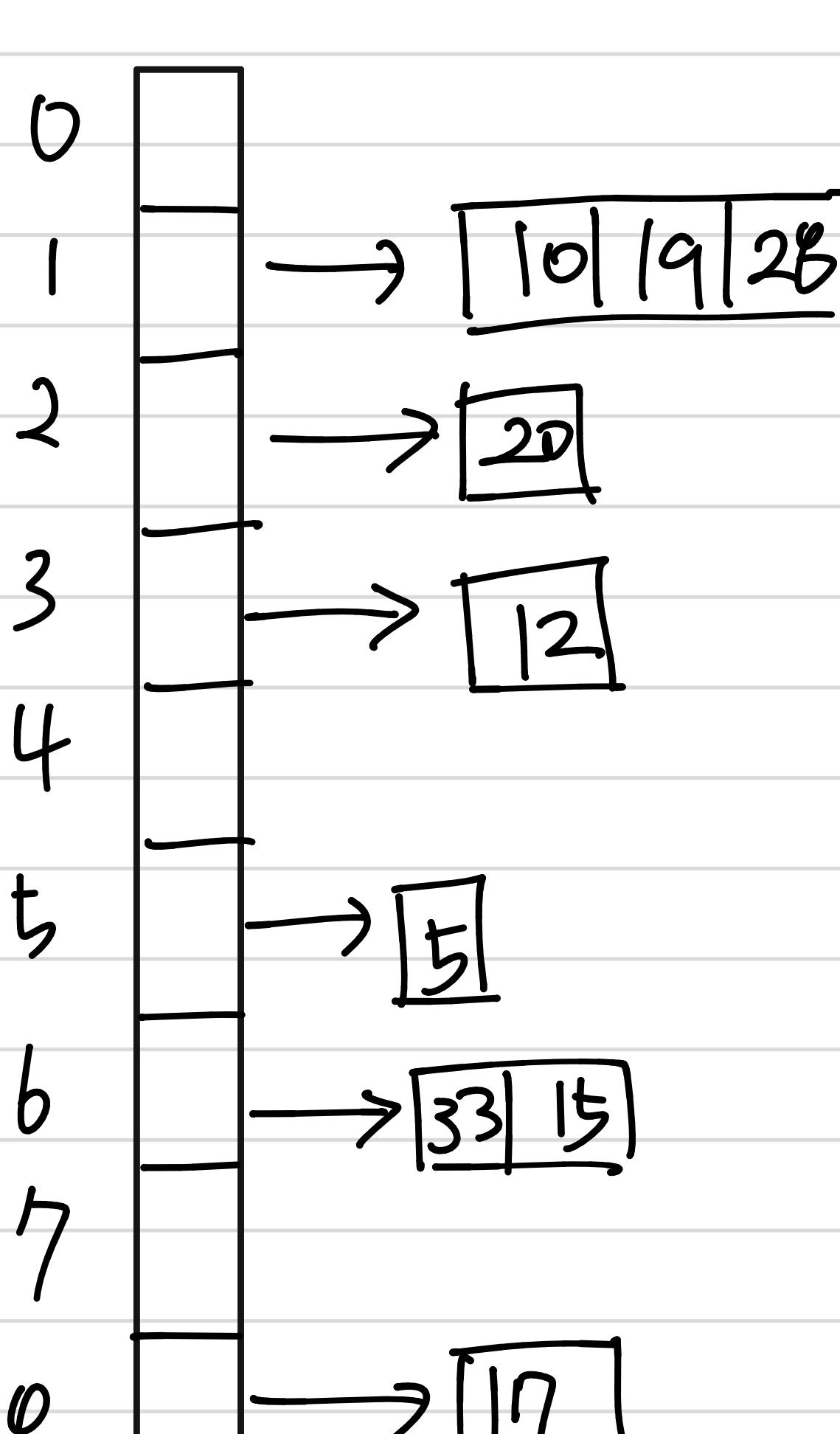
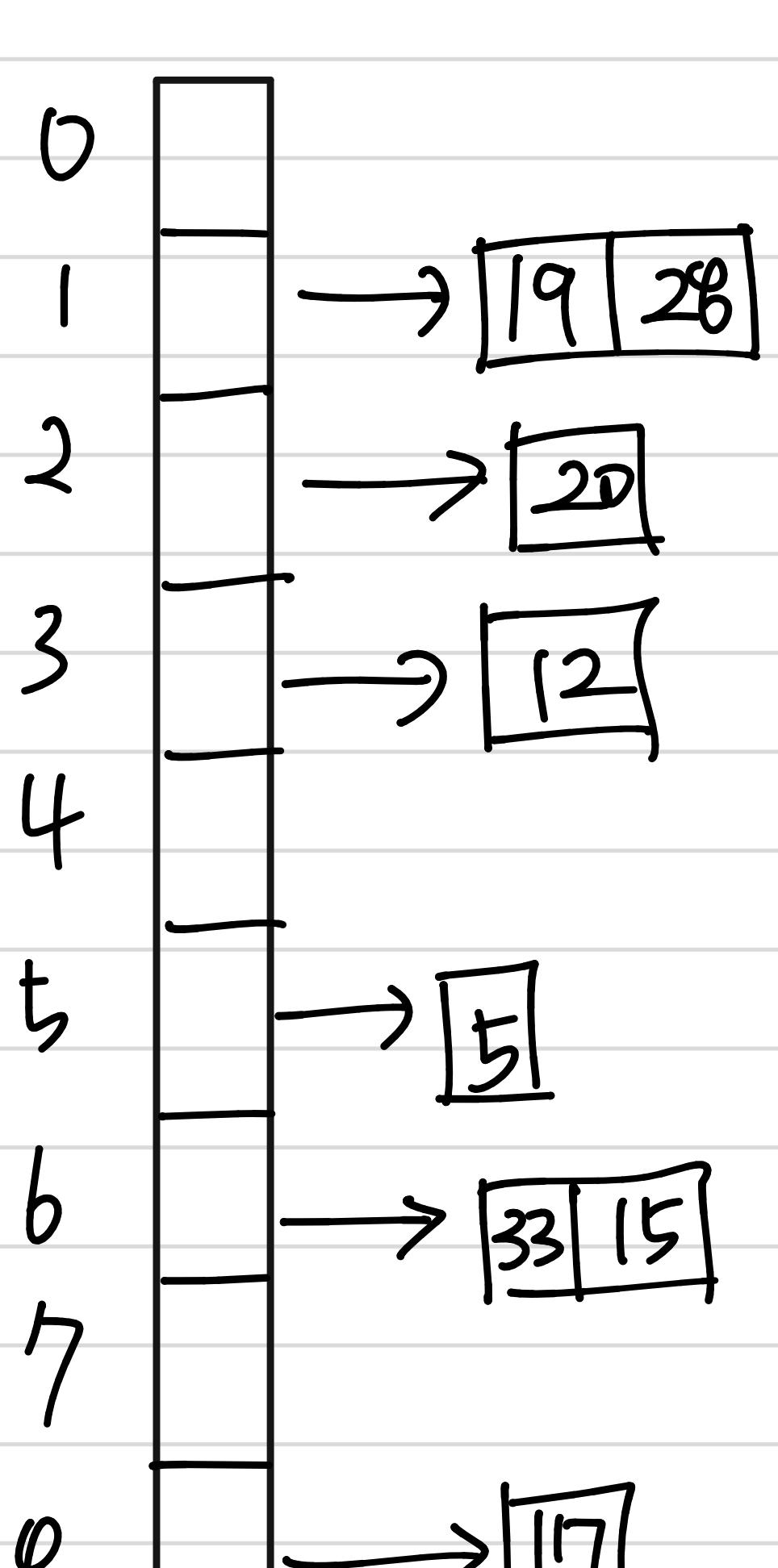
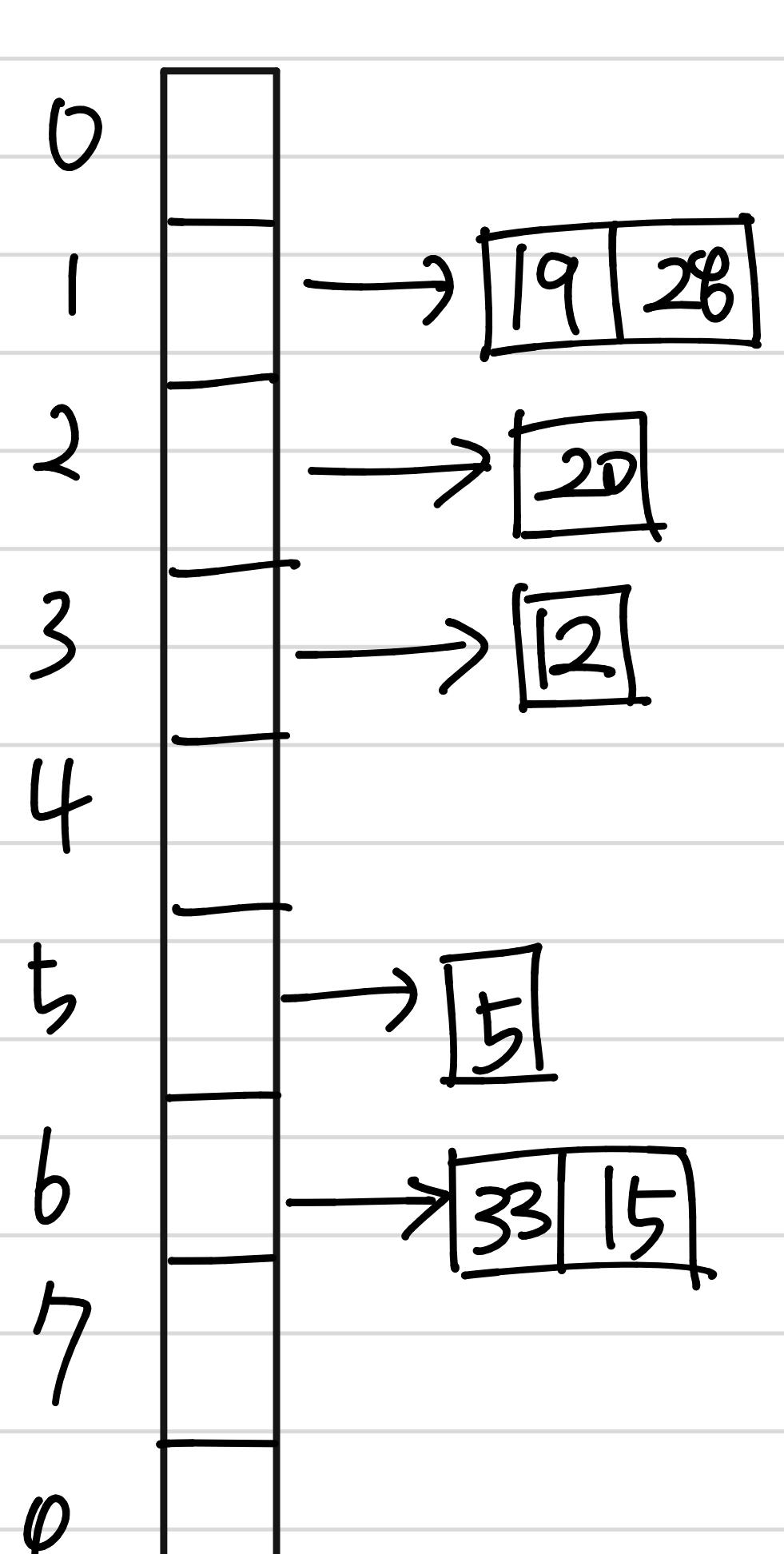


⑤ 20 삽입 ($20\%9=2$)

⑥ 33 삽입 ($33\%9=6$)



⑦ 12 삽입 ($12\%9=3$) ⑧ 17 삽입 ($17\%9=8$) ⑨ 10 삽입 ($10\%9=1$)



마지막 상태

#2. (11.4-1)

$h(k)=k$, $M=11$ 인 경우.

각각의 경우에서

T_n 는 번째 숫자가 들어가고 나서

Hash table의 모습을 의미한다.

(1) Linear probing 사용.

<10, 22, 31, 4, 15, 28, 17, 88, 59>

$$h(k, i) = (k+i) \bmod 11$$

last state
↓

slot	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
0		22	22	22	22	22	22	22	22
1								88	88
2									
3									
4				4	4	4	4	4	4
5						15	15	15	15
6							28	28	28
7								17	17
8									59
9						31	31	31	31
10	10	10	10	10	10	10	10	10	10

① $T_1 : 10\%11 = 10 \Rightarrow 10$ 을 slot 10에 Insert

② $T_2 : 22\%11 = 0 \Rightarrow 22$ 을 slot 0에 Insert

③ $T_3 : 31\%11 = 9 \Rightarrow 31$ 을 slot 9에 Insert

④ $T_4 : 4\%11 = 4 \Rightarrow 4$ 를 slot 4에 Insert

⑤ $T_5 : 15\%11 = 4 \Rightarrow$ (충돌 발생) | 번의 추가 탐색 후 slot 5에 삽입

⑥ $T_6 : 28\%11 = 6 \Rightarrow 28$ 을 slot 6에 Insert

⑦ $T_7 : 17\%11 = 6 \Rightarrow$ (충돌 발생) | 번의 추가 탐색 후 slot 7에 삽입

⑧ $T_8 : 88\%11 = 0 \Rightarrow$ (충돌 발생) | 번의 추가 탐색 후 slot 8에 삽입

⑨ $T_9 : 59\%11 = 4 \Rightarrow$ (충돌 발생) 4번의 추가 탐색 후 slot 9에 삽입

(history : 5 → 6 → 7 → 8)

$$h(5, 1) \uparrow h(5, 3) \uparrow h(5, 4)$$

h(5, 2)

last state

(2) Quadratic probing 사용

$$h(k, i) = (k+i+3i^2) \bmod 11$$

slot	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
0		22	22	22	22	22	22	22	22
1									
2									88 88
3								17	17 17
4				4	4	4	4	4	4
5									
6						28	28	28	28
7									59
8						15	15	15	15
9				31	31	31	31	31	31
10	10	10	10	10	10	10	10	10	10

① $T_1 : 10\%11 = 10 \Rightarrow 10$ 을 slot 10에 Insert

② $T_2 : 22\%11 = 0 \Rightarrow 22$ 을 slot 0에 Insert

③ $T_3 : 31\%11 = 9 \Rightarrow 31$ 을 slot 9에 Insert

④ $T_4 : 4\%11 = 4 \Rightarrow 4$ 를 slot 4에 Insert

⑤ $T_5 : 15\%11 = 4 \Rightarrow$ (충돌 발생) | 번의 추가 탐색 후 slot 8에 삽입

⑥ $T_6 : 28\%11 = 6 \Rightarrow 28$ 을 slot 6에 Insert

⑦ $T_7 : 17\%11 = 6 \Rightarrow$ (충돌 발생) 3번의 추가 탐색 후 slot 3에 삽입

⑧ $T_8 : 88\%11 = 0 \Rightarrow$ (충돌 발생) 4번의 추가 탐색 후 slot 2에 삽입

⑨ $T_9 : 59\%11 = 4 \Rightarrow$ (충돌 발생) 2번의 추가 탐색 후 slot 1에 삽입

- ① history $b \rightarrow 9 \rightarrow$ ③ 순서대로 $h(17.1), h(17.2), h(17.3)$
 ② history $4 \rightarrow 3 \rightarrow 8 \rightarrow 8 \rightarrow 3 \rightarrow 4 \rightarrow 0 \rightarrow$ ③ 순서대로 $h(88.1) \dots h(88.8)$
 ④ history $8 \rightarrow$ ① 순서대로 $h(59.1), h(59.2)$

(2) Double hashing 사용

$$h(k, i) = (K + (1 + k \bmod (m-1))i) \bmod m$$

Last State
↓

slot	T ₁	T ₂	T ₃	T ₄	T ₅	T ₆	T ₇	T ₈	T ₉
0	22	22	22	22	22	22	22	22	22
1									
2									59
3					17	17	17		
4		4	4	4	4	4	4		
5			15	15	15	15	15		
6				28	28	28	28		
7					88	88			
8									
9		31	31	31	31	31	31	31	
10	10	10	10	10	10	10	10	10	10

① T₁: $10 \% 11 = 10 \Rightarrow 10$ 을 slot 10에 Insert

② T₂: $22 \% 11 = 0 \Rightarrow 22$ 을 slot 0에 Insert

③ T₃: $31 \% 11 = 9 \Rightarrow 31$ 을 slot 9에 Insert

④ T₄: $4 \% 11 = 4 \Rightarrow 4$ 를 slot 4에 Insert

⑤ T₅: $15 \% 11 = 4 \Rightarrow$ (충돌 발생) 2번의 추가 탐색 후 slot 5에 삽입

⑥ T₆: $28 \% 11 = 6 \Rightarrow 28$ 을 slot 6에 Insert

⑦ T₇: $88 \% 11 = 6 \Rightarrow$ (충돌 발생) 1번의 추가 탐색 후 slot 3에 삽입

⑧ T₈: $88 \% 11 = 0 \Rightarrow$ (충돌 발생) 2번의 추가 탐색 후 slot 7에 삽입

⑨ T₉: $59 \% 11 = 4 \Rightarrow$ (충돌 발생) 2번의 추가 탐색 후 slot 2에 삽입

$$h(15, 1) = (15+6) \% 11 = 10$$

$$h(15, 2) = (15+12) \% 11 = 5 \rightarrow \text{비어있는 slot}$$

$$h(17, 1) = (17+8) \% 11 = 3 \rightarrow \text{비어있는 slot}$$

$$h(88, 1) = (88+9) \% 11 = 9$$

$$h(88, 2) = (88+18) \% 11 = 7 \rightarrow \text{비어있는 slot}$$

$$h(59, 1) = (59+10) \% 11 = 3$$

$$h(59, 2) = (59+20) \% 11 = 2 \rightarrow \text{비어있는 slot}$$

#3. (교재 22.3 - 5)

Classification of edges

1. Tree edges : DFS Tree Eπ 내에 존재하는 edge

2. Back edges : Gπ 내에서 임의의 vertex u와 자손 v를 연결하는

Tree edge가 아닌 edge

3. Forward edges : Gπ 내에서 임의의 vertex u와 조상 v를 연결하는

edge로, Self-loop도 역행 간선으로 간주한다.

4. Cross edges : 모든 다른 edge를 의미

DFS Algorithm 내에서

u의 간선 (u, v)을 살피면

v의 색에 따른 간선 분류

- 1. WHITE는 Tree 간선
- 2. GRAY는 역행 간선
- 3. BLACK은 순행 또는 교차간선

Theorem 22.7 (Parenthesis 정리)

graph $G = (V, E)$ 의 DFS에서 서로 다른 두 정점 u, v에 대해서

다음의 세 가지 상황만이 존재할 수 있다.

1. $[u.d, u.f], [v.d, v.f]$ 가 겹치는 블록 X, G_{π} 에서 서로의 자손이 아니다.
2. $[u.d, u.f]$ 가 $[v.d, v.f]$ 를 완전히 포함, G_{π} 에서 v가 u의 자손
3. 2의 반례 상황.

(증명) $u.d < v.d$ 인 상황 가정

① If $v.d < u.f$: u가 gray일 때, v가 발견되었으므로, v가 u의 자손

또한 이 경우에 v가 u의 자손이므로, 재귀 함수 특성에 의해 자동적으로 $v.f < u.f$

② If $u.f < v.d$: u가 black이 된 후로 v가 발견되었으므로, 서로가 서로의 자손이 아니다.

and $u.d > v.d$ 또한 대칭성을 가지고 Parenthesis 정리가 증명된다.

다음의 정리, 개념들로 a, b, c를 증명하자

a. $u.d < v.d < v.f < u.f \Leftrightarrow (u, v)$ 가 tree edge or forward edge

(\Rightarrow) Parenthesis 정리에서 2에 해당한다. 따라서 G_{π} 에서 v가 u의 자손에 해당한다. v가 u의 자손이므로, edge의 정의에 따라서 Back edge / cross edge 가 될 수 없고, tree edge거나 forward edge이다.

(\Leftarrow) (u, v) 가 tree edge거나 forward edge이므로, G_{π} 내에서 v가 u의 자손이다. DFS 내에서 u는 v보다 먼저 발견되어, u가 GRAY 상태에서 v가 발견되었으므로 $u.d < v.d < v.f < u.f$ 가 성립한다.

b. $v.d \leq u.d < u.f \leq v.f \Leftrightarrow (u, v)$ 가 back edge

(\Rightarrow) Parenthesis 정리에서 3에 해당한다. G_{π} 에서 u가 v의 자손에 해당하며, v가 u의 자손이므로, edge의 정의에 따라서 (u, v) 는 back edge이다.

(\Leftarrow) (u, v) 가 back edge이므로, G_{π} 내에서 u가 v의 자손이다. DFS 내에서 v는 u보다 먼저 발견된다. v가 GRAY 상태에서 u가 발견되어, u가 v의 자손이므로, u가 먼저 black이 된다. 즉, $v.d \leq u.d < u.f \leq v.f$ 가 성립한다.

c. $v.d < v.f < u.d < u.f \Leftrightarrow (u, v)$ 가 cross edge

(\Rightarrow) Parenthesis 정리에서 1에 해당한다. G_{π} 에서 서로의 자손에 해당하지 않는다. tree edge, back edge, forward edge는 G_{π} 내에서 조상과 자손을 연결하는 edge이므로, (u, v) 는 cross edge에 해당한다.

(\Leftarrow) (u, v) 가 cross edge이므로, u와 v는 G_{π} 내에서 서로의 자손에 해당하지 않는다. v가 u보다 먼저 발견되었으므로, $v.d < u.d$ 이며, 또한 서로의 자손에 해당하지 않으므로, v가 gray인 상태에서 u를 발견할 수 있고, v가 black이 된 후에 u가 발견된다. 따라서 $v.d < v.f < u.d < u.f$ 가 성립함을 보일 수 있다.

#4. (22-3). Euler tour $G=(E,V)$ a. for each vertex $v \in V$ 에 대해 $\text{in-degree}(v) = \text{out-degree}(v)$ $\Leftrightarrow G$ has Euler tour ($P \leftrightarrow Q$ 형태)

(\Leftarrow) 이 방향의 대수를 먼저 증명하자. 만약 어떤 vertex의 in degree \neq out degree라고 가정하고, 이 vertex를 u 라고 해보자. u 가 tour의 start vertex라고 하면 in degree와 out degree가 달라 바이어에 u 로 돌아올 수 없다.

u가 경유 vertex라고 하면 두 가지 case가 존재

(Case 1) $\text{in-degree}(u) > \text{out-degree}(u)$: u 로 들어가는 모든 간선을 통해 순회할 때 들어가는 간선이 나가는 간선보다 많으므로, u 에 갇히게 된다.(Case 2) $\text{in-degree}(u) < \text{out-degree}(u)$: u 로 들어가는 모든 간선을 이용하여 tour를 마치고, 나오면 나서서 나가는 남은 간선을 순회하지 못하므로, Euler tour가 존재할 수 없다.

(\Rightarrow)

u_1 edge = 1, u_2 edge = 2, u_3 edge = 2
 \Rightarrow 두 가지 경우 모두 Euler tour 존재

$|E|$ 의 크기가 k 개 이하이며, P 조건을 만족하는 graph는 Euler tour을 가진다고 가정하자. $V = [v_1 \dots v_n]$ 이라고 가정.

G 에서 임의의 꼭짓점 v_1 을 선택할 때, v_1 을

$v_1 - \dots - v_i$ 인 임의의 path 항상 찾을 수 있고, 이러한 path에

사용된 edge의 집합은 E' 이라고 하자. G 에서 E' 을 모두 제거하고, 연결된 edge가 없는 vertex를 제거하면 G 는 $G_1 \dots G_m$ 으로 나뉘며, $G_1 \dots G_m$ 모두 서로소 집합이며 P 조건을 만족시키는 연결 graph가 됨을 알 수 있다.

이제, $G_1 \dots G_m$ 은 모두 edge 개수가 k 개 이하므로, Euler tour을 가진다.

앞서 살펴본 path q 에 대해서 q 는 $G_1 \dots G_m$ 중 적어도 하나의 vertex를 모두 지나는데 이는 q 가 $G_1 \dots G_m$ 중 제거 않는 graph G_k 가 존재한다면 G_k 는 G 로부터 원래 꾀로 떨어진 graph이기 때문에 G 가 연결 graph라는 사실에 모순된다.

よし $G_1 \dots G_m$ 을 지나는 하나의 vertex를 $a_1 \dots a_m$ 이라고 가정하고, $G_1 \dots G_m$ Euler tour을 $C_1 \dots C_m$ 이라고 하자.

마지막으로, G 에서 q 를 따라 이동하면서 $a_1 \dots a_m$ 을 만날 때마다 $C_1 \dots C_m$ 을 추가하여 순회하면 이는 G 의 Euler path가 된다.

즉, 귀납법에 따라서 $P \rightarrow Q$ 임을 증명하였다.

b. Euler tour을 구하는 알고리즘

Euler tour가 존재한다는 보장이 있다면 주어진 상황에서 아직 가지 않은 edge를 선택하여 tour한다면 항상 Euler tour을 만들어낼 수 있다.

따라서 다음과 같이 Algorithm을 작성할 수 있다. 어떤 vertex에서

$G=(E,V)$ 출발하여도 항상 Euler tour

를 만들 수 있는 성질이 있음.

(이는 G 에서 증명함)

Euler tour(G):

- each $e \in E$, $e.\text{visited} = \text{False}$
- $L \rightarrow \text{tour List}$
- $(u,v) \in E$ 인 arbitrary edge 선택
- append u to L , $(u,v).\text{visited} = \text{True}$
- while V 로부터 나가는 unvisited edge $(v,w) \in E$
- $(v,w).\text{visited} = \text{True}$
- $V = w$
- append V to L

Time complexity : $O(|E|)$

#5 (23.1-1)

정의 23.1 : $G=(V,E)$ 는 graph로, A 는 G 의 최소신장트리에 포함되는 edge set 으로 A 에 일때, (u,v) 가 G 의 cut인 $(S, V-S)$ 를 cross하는 경량 간선인 경우에 (u,v) 는 A 의 safety edge이다. 즉, 최소 신장 트리를 구성하는 과정에서 (u,v) 를 A 에 추가하는 것이 가능하다.)

(u,v) is $G=(V,E)$ 의 minimum weight edge

최소 신장 Tree를 구성하는 과정에서 번 첫 관계를 생각하자. 먼저 $|V|$ 개의 vertex 중 u, v 를 포함하는데 u, v 가 나머지로 graph의 일부인 cut인 $(S, V-S)$ 를 생각할 수 있다. 즉, 일반성을 잊지 않고 $S \subseteq U, V-S \subseteq V$ 최소신장트리를 만들기 위한 티의 부분집합 A 가 처음에 공집합이자고 할 때, A 에 (u,v) 를 추가 23.1-1에 의해서 안전하게 포함시킬 수 있다.

이후에 Safety edge를 계속하여 추가해면서 A 를 최소신장트리로 만들면 A 는 (u,v) 를 포함하는 G 의 최소신장트리가 된다.

따라서 (u,v) 를 포함하는 G 의 최소신장트리를 항상 만들 수 있음을 증명하였다.

#6. (24.2-1)

DAG - SHORTEST-PATHS (G, w, s):

- topologically sort the vertices of G
- INITIALIZE-SINGLE-SOURCE(G, s)
- for each vertex u , taken in topologically sorted order
- for each vertex $v \in G.\text{adj}[u]$.
- RELAX(u, v, w)

먼저 G 를 topological sorting 한다.

DFS를 실행하면 finishing time의 역순으로 정렬하면

$r-s-t-x-y-z$ 이다.

Line 2를 수행한 상태

(d)	r s t x y z	(T)	r s t x y z
0 00 00 00 00 00		NIL NIL NIL NIL NIL NIL	

Line 3에서 $u=r$ 인 for문이 수행된 이후 상태

(d)	r s t x y z	(T)	r s t x y z
0 5 3 00 00 00		NIL r r NIL NIL NIL	

$r \rightarrow s$ relax ($0+5 < 00$) $r.d + w(r,s) < s.d$

$r \rightarrow t$ relax ($0+3 < 00$) $r.d + w(r,t) < t.d$.

Line 3에서 $u=s$ 인 for문이 수행된 이후 상태

(d)	r s t x y z	(T)	r s t x y z
0 5 3 11 00 00		NIL r r NIL NIL NIL	

$s \rightarrow x$ relax ($5+6 < 00$) $s.d + w(s,x) < x.d$

Line 3에서 $u=t$ 인 for문이 수행된 이후 상태

(d)	r s t x y z	(T)	r s t x y z
0 5 3 10 7 5		NIL r r t t t	

$t \rightarrow x$ relax ($3+7 < 11$) $t.d + w(t,x) < x.d$

$t \rightarrow y$ relax ($3+4 < 00$) $t.d + w(t,y) < y.d$

$t \rightarrow z$ relax ($3+2 < 00$) $t.d + w(t,z) < z.d$

Line 3에서 $u=2$ 인 상태에서 f_{value} 가 수행된 이후 상태

(d) $r \ s \ t \ x \ y \ z$
 $0 \ 5 \ 3 \ 10 \ 7 \ 5$ relax 발생X (π로한 변화X)

Line 3에서 $u=5$ 인 상태에서 f_{value} 가 수행된 이후 상태

(π) $r \ s \ t \ x \ y \ z$
 $0 \ 5 \ 3 \ 10 \ 7 \ 5$ relax 발생X (π로한 변화X)

Line 3에서 $u=3$ 인 상태로는 수행을 해도 상관없지만 위상정렬상

마지막 순서이므로, relax가 발생하지 않는다.

Result

(d) $r \ s \ t \ x \ y \ z$ (π) $r \ s \ t \ x \ y \ z$
 $0 \ 5 \ 3 \ 10 \ 7 \ 5$ NIL r r t t t

#7 (교재 24.3-3)

check correctness

DIJKSTRA(G, w, s):

1. INITIALIZE-SINGLE-SOURCE(G, s)

2. $S = \emptyset$

3. $Q = G.V$

4. while $|Q| > 1$

5. $u = \text{EXTRACT-MIN}(Q)$

6. $S = S \cup \{u\}$

7. for each $v \in G.\text{Adj}[u]$,

8. RELAX(u, v, w)

먼저 4행을 바탕으로 전의 다익스트라 algorithm이 정확하게 shortest path를 계산한다고 가정하자.

바로 전과 다른 점은 Q 에 남아 있는 임의의 node v 에 대해서 Line 5-8 과정을 수행하지 않는다는 것이다.

Q 의 마지막 node v 는 graph 전체에서 algorithm이 수행될 때, S 에 포함되지 않는 유일한 set이며, 다익스트라 알고리즘의 4-8 Loop에 대해서 다음의 Loop invariant가 성립한다.

Loop Invariant: Loop 시작 전 S 에 있는 모든 vertex에 대해서
 $v.d = f(S, v)$, 즉 최단거리가 보장되어 있다.

따라서 Q 에 남아 있는 경우에 S 에는 v 를 제외한 vertex가 존재하고, vertex들의 shortest path가 보장되어 있다는 사실을 알 수 있다.

① $S \rightarrow u$ 인 path가 존재하지 않는 경우

Line 8에서 어떤 vertex도 u 를 RELAX할 수 없기 때문에 $u.d$ 는 update되지 않은 상태로 계속 ∞ 의 값으로 남아 있다.

② $S \rightarrow u$ 의 shortest path인 $S \rightarrow g \rightarrow u$ 가 존재할 때,

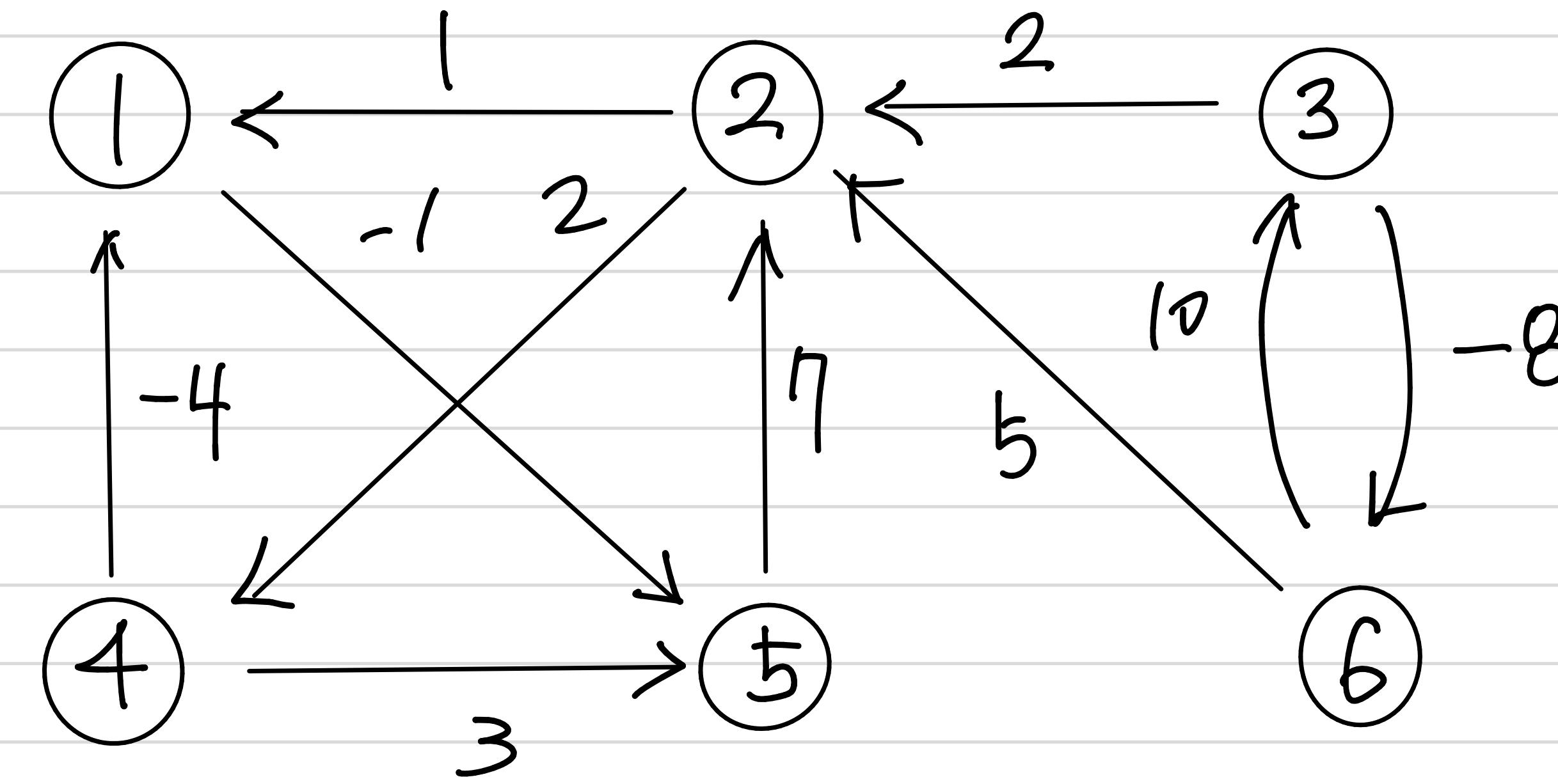
Line 5에서 ∞ 의 minimum d를 갖는 vertex가 가진 Loop를 생각할 때,

인접 vertex인 u 를 Line 8에서 RELAX 하면, 이를 수행한 뒤에

$u.d = f(S, u)$ 를 보장할 수 있다.

: 수행한 다익스트라 알고리즘 또한 모든 vertex의 shortest path를 정확하게 계산하여 correct하다.

#8 (교재 25.3-1) python 사용



① vertex 0을 추가하고, 0으로부터 모든 vertex로의 0인 간선을 추가한 graph G' 을 생성하고, Bellman-Ford Algorithm을 이용하여 $f(0, u)$ ($1 \leq u \leq 6$)을 모두 계산한다.

$h(u) = f(0, u)$ 로 설정한다. 그에 따라 h 는 다음과 같은 값을 갖는다.

----- h value -----
 $[-5, -3, 0, -1, -6, -8]$

순서대로 $h[1], h[2], \dots, h[6]$ 이다.

② 음수인 가중치를 없애기 위해서 $\hat{w}(u, v) = w(u, v) + h[u] - h[v]$ 이용. \hat{w} 는 다음과 같이 계산된다.

----- w_hat value -----
 $[[\inf \inf \inf \inf 0. \inf]$
 $[3. \inf \inf 0. \inf \inf]$
 $[\inf 5. \inf \inf \inf 0.]$
 $[0. \inf \inf \inf 8. \inf]$
 $[\inf 4. \inf \inf \inf \inf]$
 $[\inf 0. 2. \inf \inf \inf]]$

행렬의 (i, j) 요소
 $\hat{w}(i, j)$ 를 의미한다.
(ex) $\hat{w}(1, 5) = 6$
 $\hat{w}(2, 1) = 3$.

\inf 가 아닌凡是 $\hat{w}[1][5] = 0, \hat{w}[2][1] = 3, \hat{w}[2][4] = 0$
 $\hat{w}[3][2] = 5, \hat{w}[3][6] = 0, \hat{w}[4][1] = 0, \hat{w}[4][5] = 8$
 $\hat{w}[5][2] = 4, \hat{w}[6][2] = 0, \hat{w}[6][3] = 2$ 이다. |

③ 음수인 가중치가 없으므로, 모든 vertex를 시작점으로 하여 다익스트라 알고리즘을 적용하면 All pair shortest path인 $\hat{f}(i, j)$ ($1 \leq i, j \leq 6$)을 계산할 수 있으며, 실제 $f(u, v)$ 는 다음과 같이 h 를 이용하여 다시 복원할 수 있고, d_{uv} matrix를 다음과 같이 나타낼 수 있다.

복원식: $f(u, v) = \hat{f}(u, v) + h(v) - h(u)$

----- d value -----
 $[[0. 6. \inf 8. -1. \inf]$
 $[-2. 0. \inf 2. -3. \inf]$
 $[-5. -3. 0. -1. -6. -8.]$
 $[-4. 2. \inf 0. -5. \inf]$
 $[5. 7. \inf 9. 0. \inf]$
 $[3. 5. 10. 7. 2. 0.]]$

행렬의 (i, j) 요소가
 d_{ij} 를 의미한다.
(ex) $d_{12} = 6$
 $d_{14} = 8$
 $d_{15} = -1$

사용한 code 참조함,

```

1 import numpy as np
2
3 inf_replacement = np.inf
4
5
6 2 usages
7 def Initialize_singleSource(G, s):
8
9     for v in G:
10         v[0] = 100000
11         v[1] = 'NIL'
12
13     G[s][0] = 0
14
15 2 usages
16 def Relax(G, u, v, w):
17     if G[v][0]>G[u][0] + w[v][v]:
18         G[v][0] = G[u][0] + w[u][v]
19         G[v][1] = u
20
21 1 usage
22 def Bellman(G, w, s, E):
23
24     Initialize_singleSource(G, s)
25     for _ in range(len(G)-1):
26         for u, v in E:
27             Relax(G, u, v, w)
28
29
30 def dajjkstra(G, w, s):
31
32     Initialize_singleSource(G, s)
33     visited = [False]*7
34
35     visited[s]=True
36
37     Q = []
38
39     Q.append([s, 0])
40
41     while Q:
42
43         x, d = min(Q, key=lambda x: x[1])
44         Q.remove([x, d])
45
46         visited[x] = True
47
48         for j in range(1, 7):
49             if w[x][j] != 100000 and not visited[j]:
50                 Relax(G, x, j, w)
51                 Q.append([j, G[j][0]])
52
53     return
54
55 G = [[0]*2 for _ in range(7)]
56 E = [[2, 1], [2, 4], [4, 1], [1, 5], [4, 5], [5, 2], [6, 2], [3, 2], [3, 6], [6, 3]]
57
58 w = np.ones((7,7)).astype(int) * 100000
59
60 w[2][1] = 1
61 w[2][4] = 2
62 w[4][1] = -4
63 w[1][5] = -1
64 w[4][5] = 3
65 w[5][2] = 7
66 w[6][2] = 5
67 w[3][2] = 2
68 w[6][3] = 10
69 w[3][6] = -8

```

```

67     for j in range(1, 7):
68         w[0][j]=0
69         E.append([0, j])
70
71     Bellman(G, w, s, E)
72     h = [G[i][0] for i in range(7)]
73
74     print()
75     print("----- h value -----")
76     print()
77     print(h[1:])
78
79     w_hat = np.zeros((7, 7)).astype(int)
80
81     for i in range(7):
82         for j in range(7):
83             w_hat[i][j] = w[i][j] + h[i] - h[j]
84
85     w_hat[w_hat>10000]=100000
86     w_hat = np.where(w_hat == 100000, inf_replacement, w_hat)
87
88     w_hat_clone = np.copy(w_hat)
89     w_hat_clone = np.delete(w_hat_clone, obj=0, axis=0) # 1행 삭제
90     w_hat_clone = np.delete(w_hat_clone, obj=0, axis=1) # 1열 삭제
91
92     for i in range(7):
93         w_hat[i][0]=0
94         w_hat[0][i]=0
95
96     print()
97     print("----- w_hat value -----")
98     print()
99     print(w_hat_clone)
100
101 curr = [[0]]*7
102
103 for i in range(1,7):
104     dajjkstra(G, w_hat, i)
105     curr[i] = [G[i][0] for i in range(7)]
106
107 for i in range(1, 7):
108     for j in range(1, 7):
109         curr[i][j] = curr[i][j] +h[j] - h[i]
110
111 curr.remove([0])
112 curr = np.array(curr)
113 curr = np.delete(curr, obj=0, axis=1)
114
115 curr[curr>10000] = 100000
116 curr = np.where(curr == 100000, inf_replacement, curr)
117 for i in range(6):
118     curr[i][i]=0
119
120 print()
121 print("----- d value -----")
122 print()
123 print(curr)
124

```