

Algorithm HW3

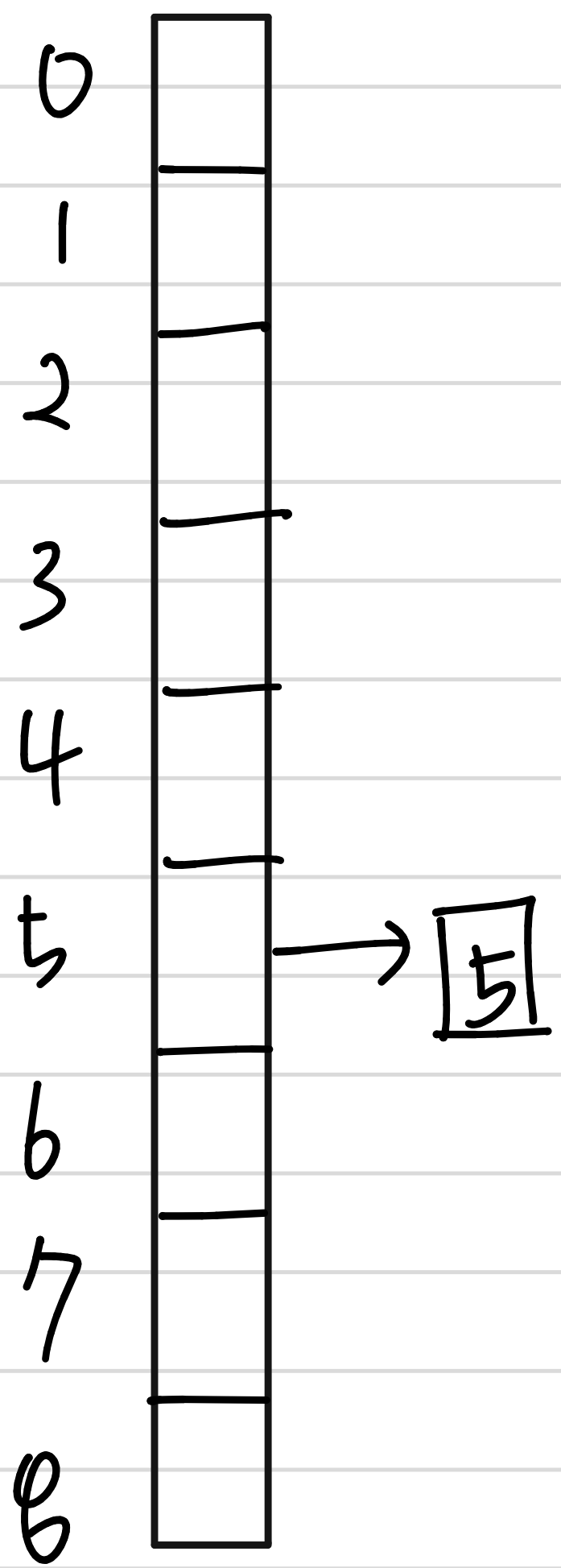
- Mechanical Engineering
- 2019-15838 주 기영

#1. (교재 11.2-2)

* chaining 방식에 의한 hash table 충돌 해결

각 slot에 Linked List가 존재하여 충돌이 일어날 때 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

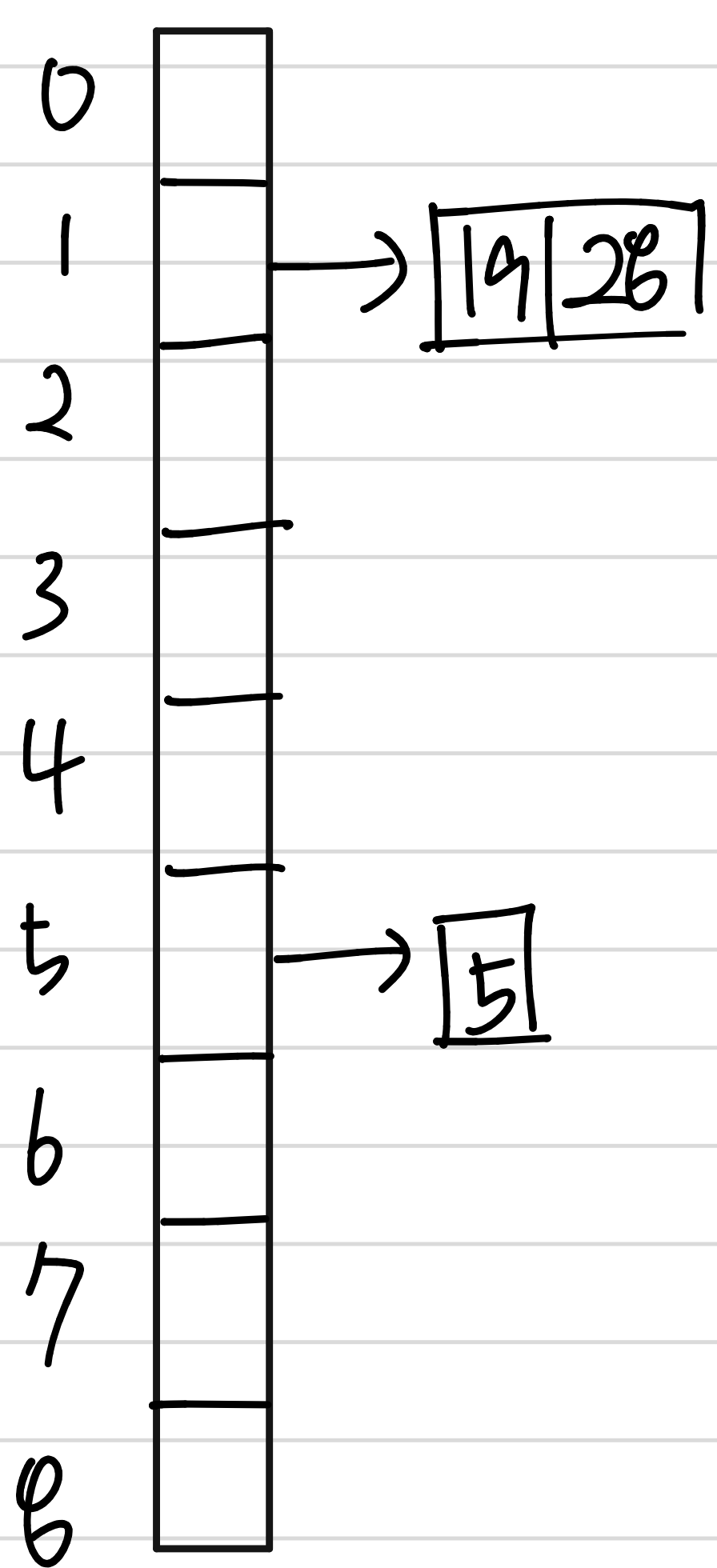
① 5 삽입 ($5\%9=5$)



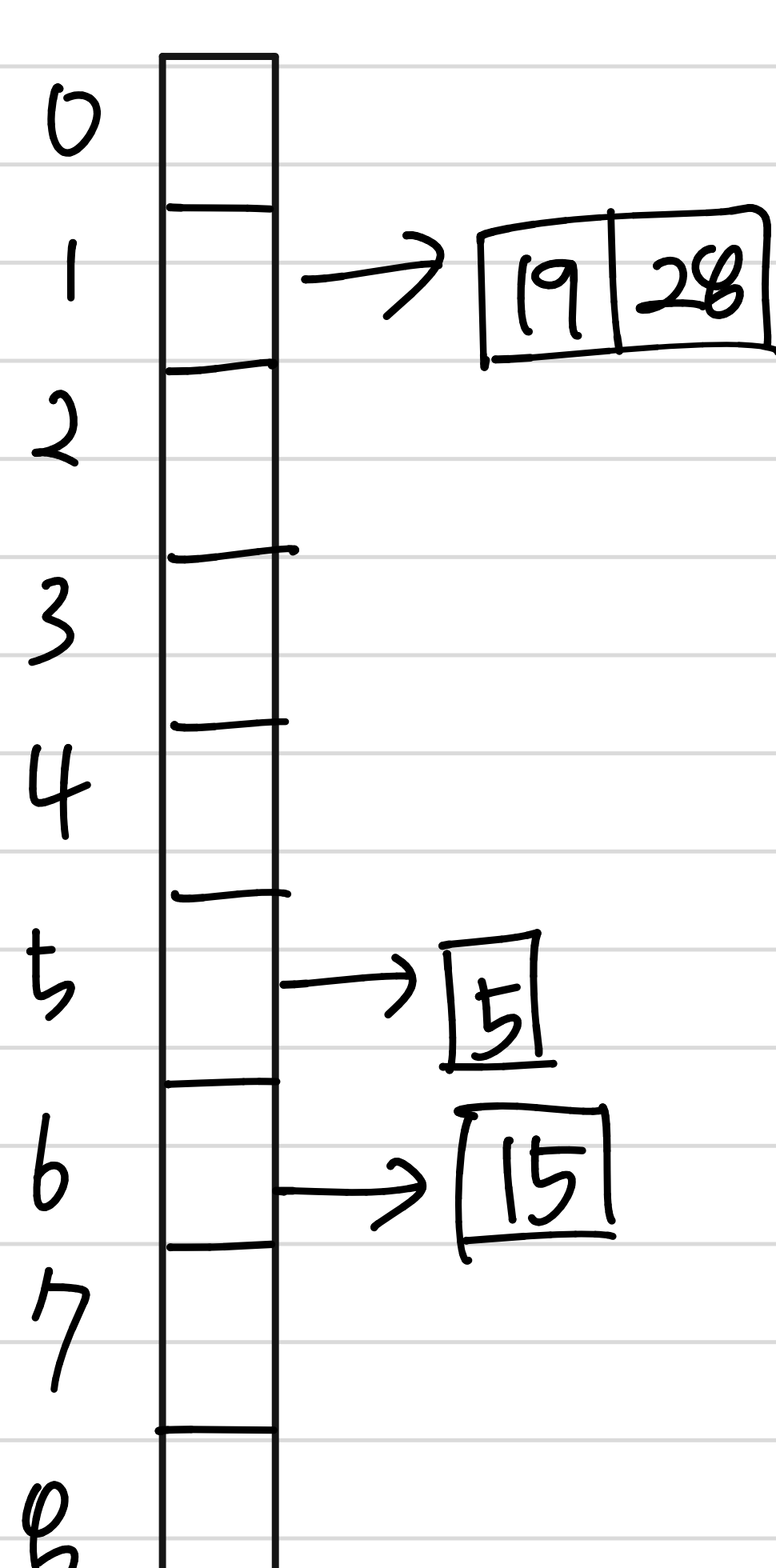
② 28 삽입 ($28\%9=1$)



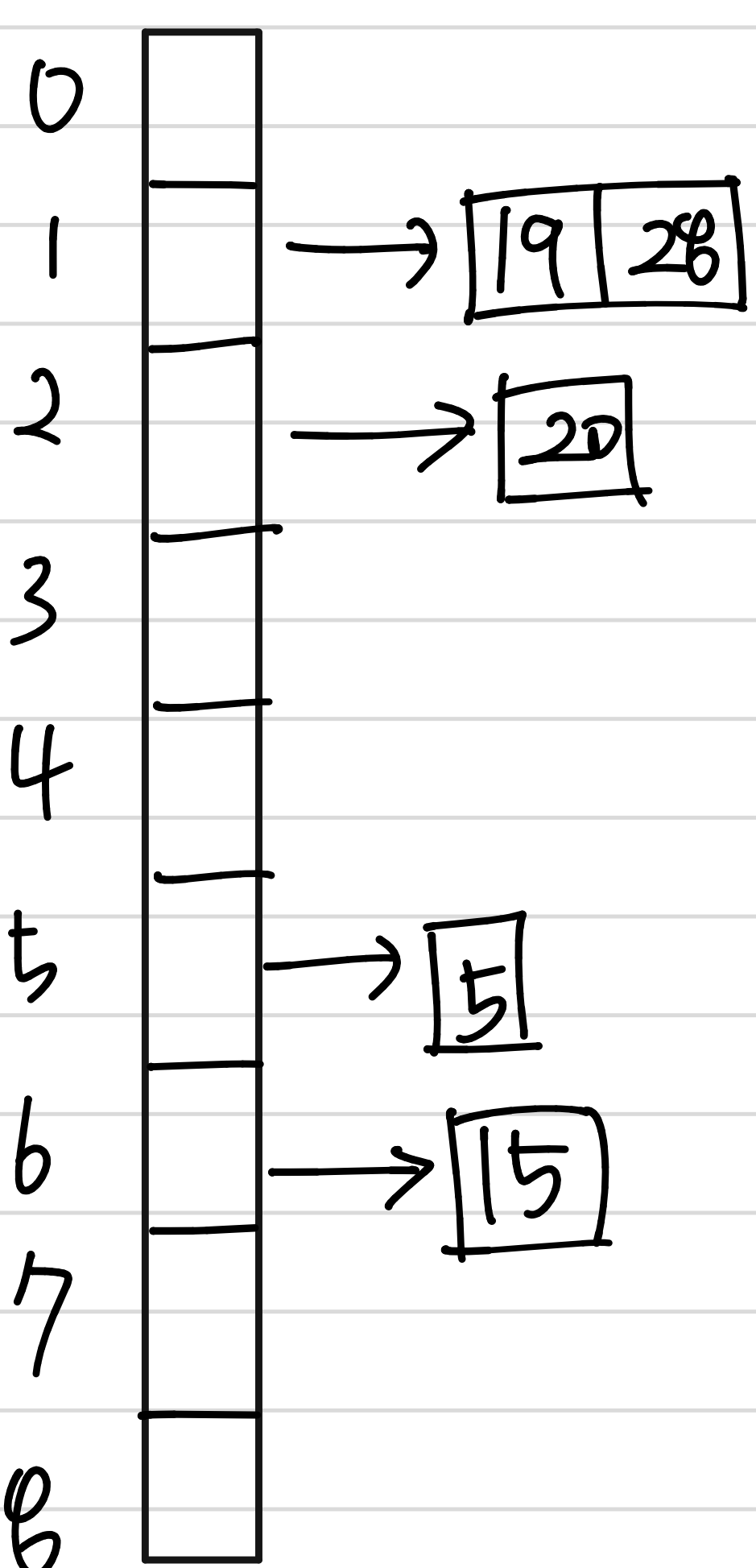
③ 19 삽입 ($19\%9=1$)



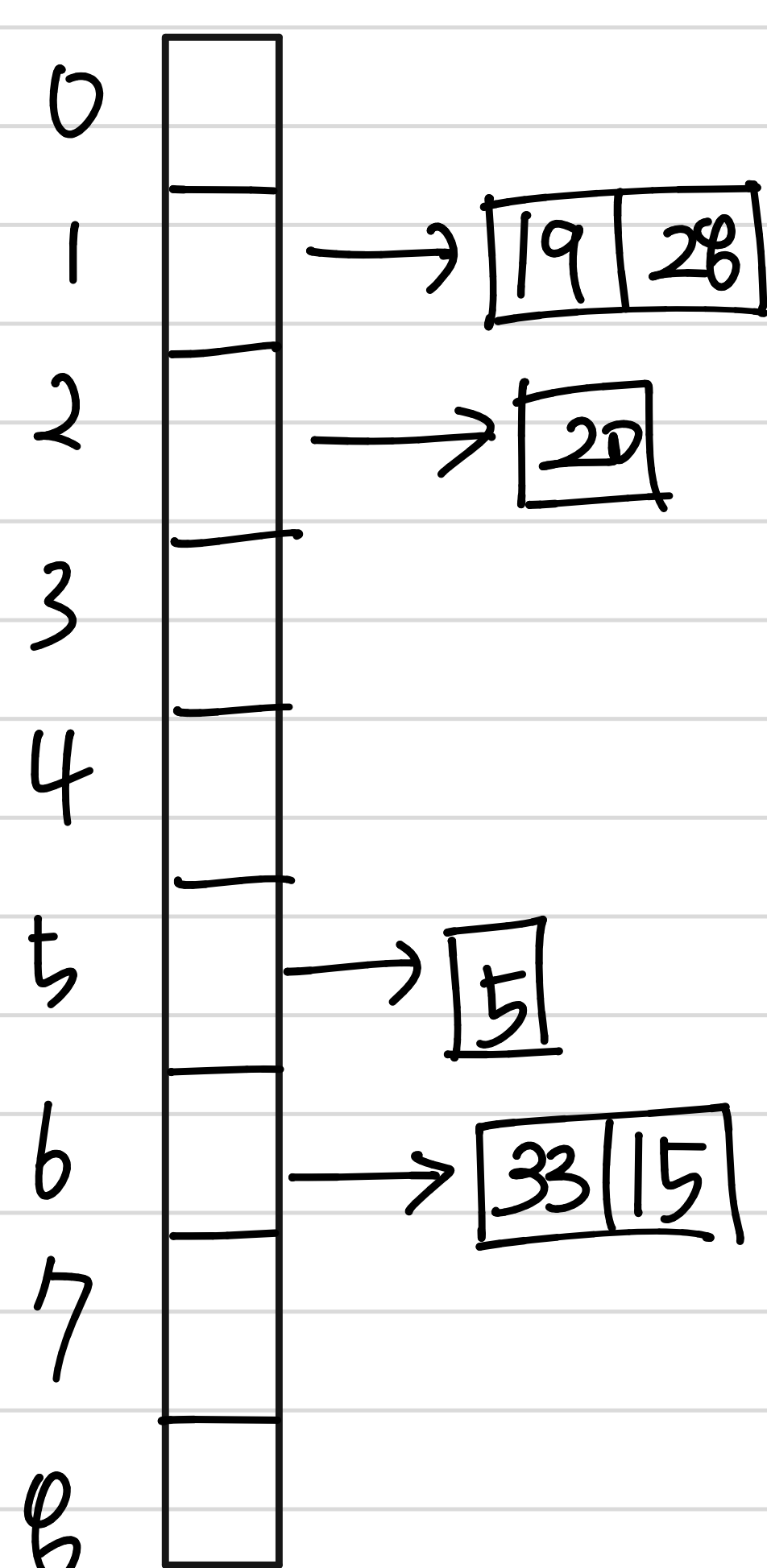
④ 15 삽입 ($15\%9=6$)



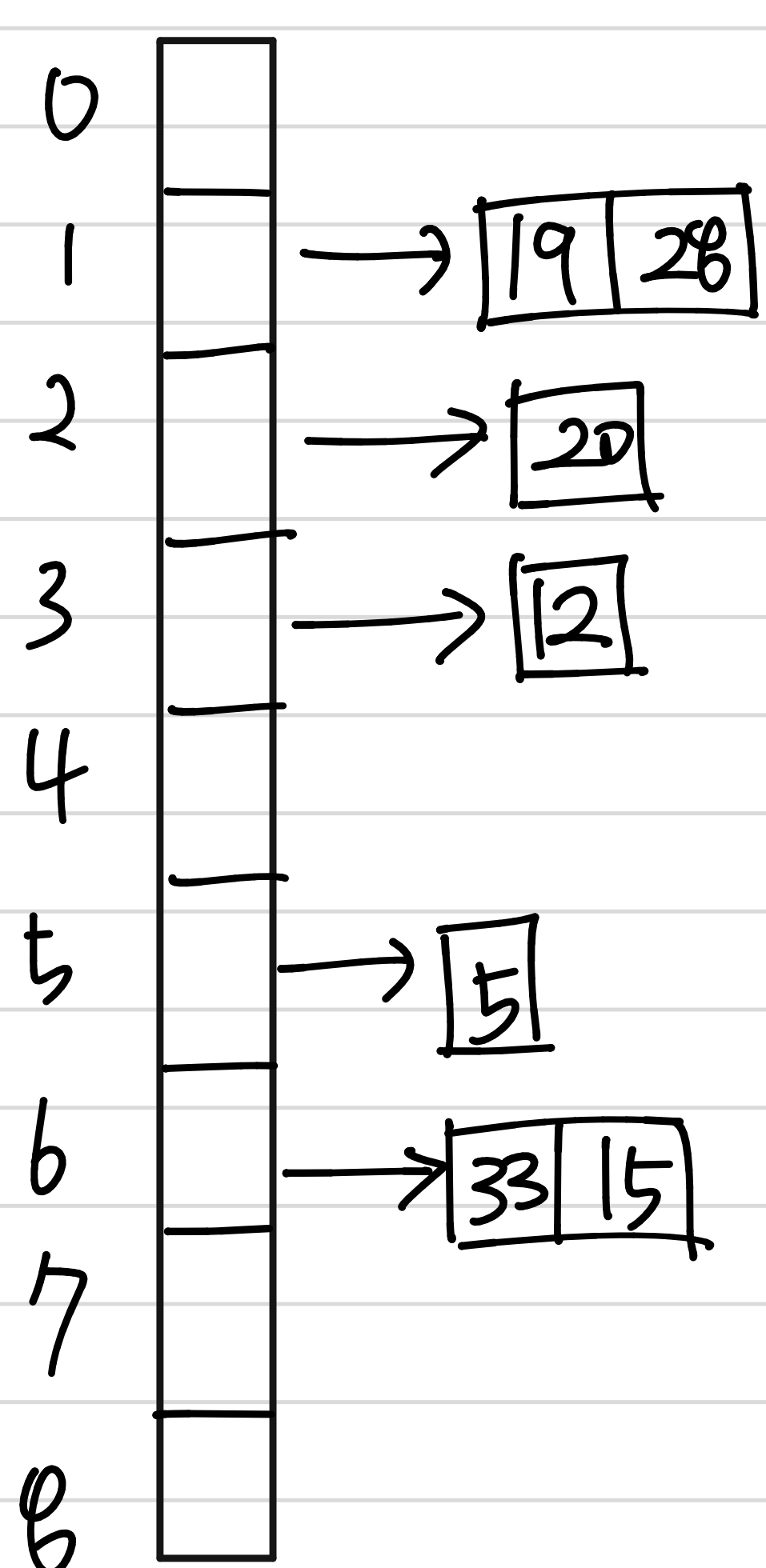
⑤ 20 삽입 ($20\%9=2$)



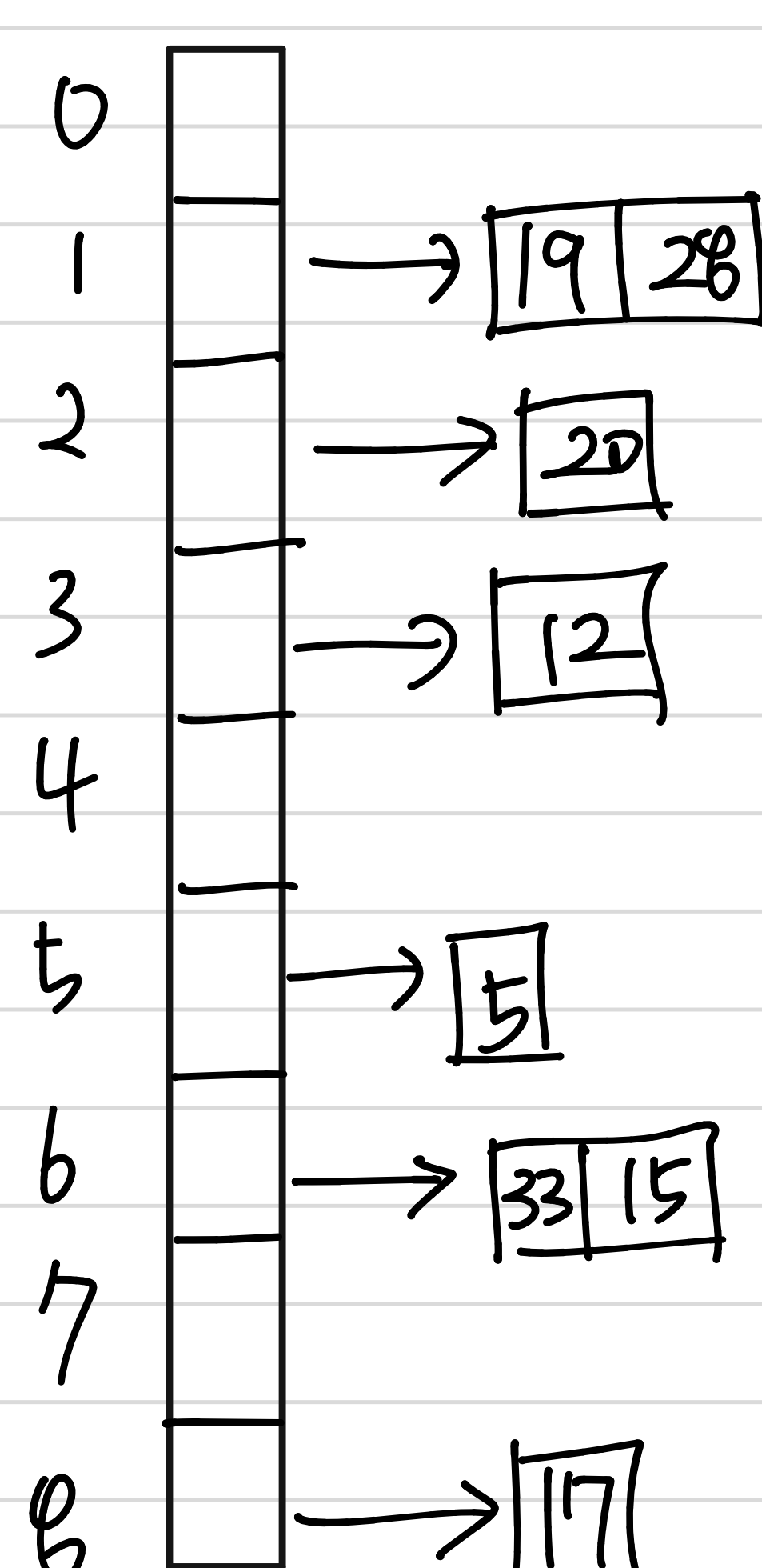
⑥ 33 삽입 ($33\%9=6$)



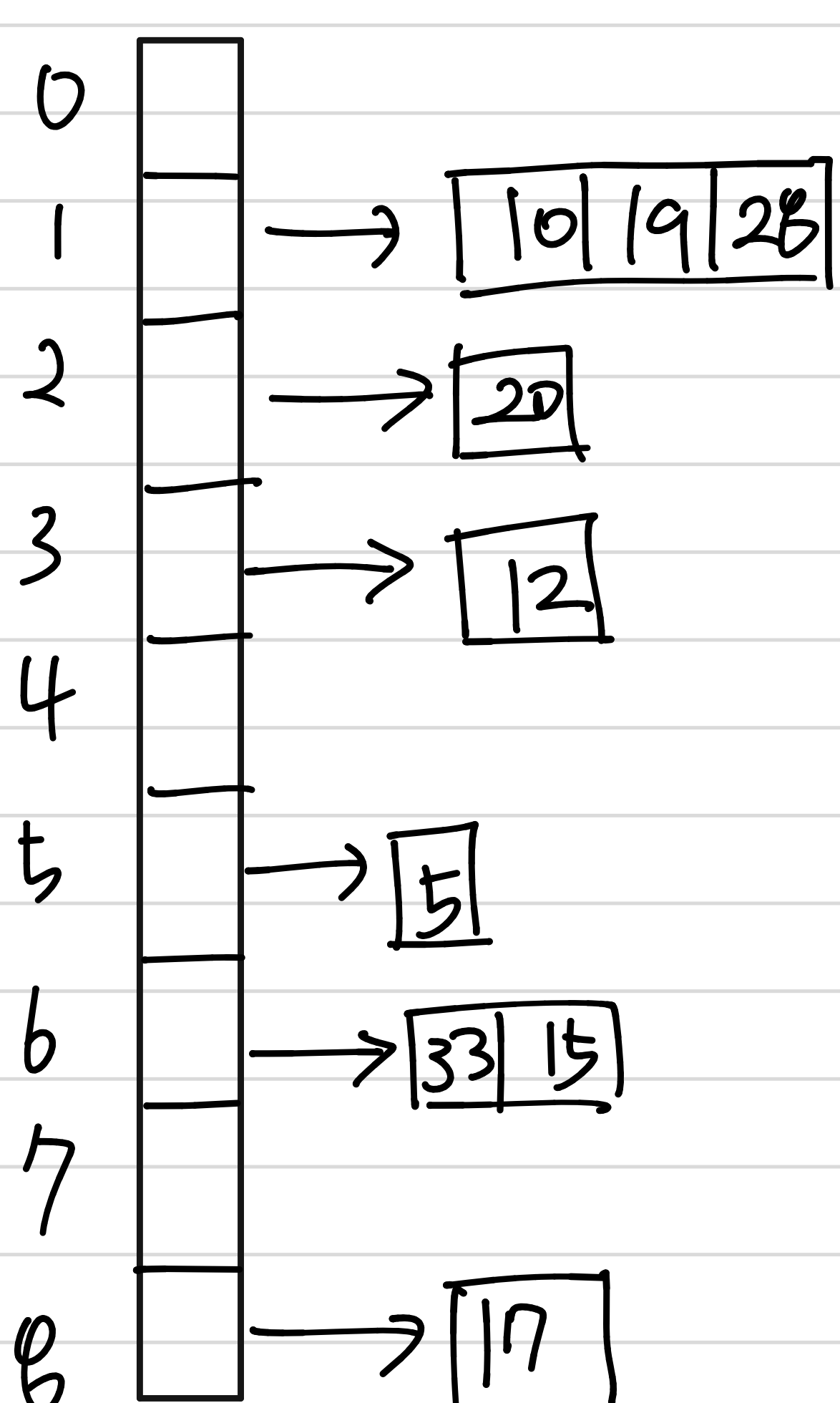
⑦ 12 삽입 ($12\%9=3$)



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_i 는 i번째 숫자라고 하자

Hash table의 모습을 리미한다.

(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	15
6						28	28	28	28
7							17	17	17
8									59
9			31	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 $T_2: 22\%11=0 \Rightarrow 22$ 를 slot 0에 Insert

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

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

⑤ $T_5: 15\%11=4 \Rightarrow$ (충돌 발생) 1번의 추가 탐색 후 slot 5에 삽입

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

⑦ $T_7: 17\%11=6 \Rightarrow$ (충돌 발생) 1번의 추가 탐색 후 slot 7에 삽입

⑧ $T_8: 88\%11=0 \Rightarrow$ (충돌 발생) 1번의 추가 탐색 후 slot 1에 삽입

⑨ $T_9: 59\%11=4 \Rightarrow$ (충돌 발생) 4번의 추가 탐색 후 slot 8에 삽입

(history: 5→6→7→8)

(2) Quadratic probing 사용

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

$h(59, 1) \uparrow h(59, 2) \uparrow h(59, 3) \uparrow h(59, 4)$

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									
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	15
9			31	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 $T_2: 22\%11=0 \Rightarrow 22$ 를 slot 0에 Insert

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

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

⑤ $T_5: 15\%11=4 \Rightarrow$ (충돌 발생) 1번의 추가 탐색 후 slot 8에 삽입

⑥ $T_6: 28\%11=6 \Rightarrow 28$ 를 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) 8번의 추가 탐색 후 slot 2에 삽입

⑨ $T_9: 59\%11=4 \Rightarrow$ (충돌 발생) 2번의 추가 탐색 후 slot 7에 삽입

- ① history 6 → 9 → ③ 순서대로 h(17.1), h(17.2), h(17.3)
 ② history 4 → 3 → 8 → 8 → 3 → 4 → 0 → ② 순서대로 h(88.1) ... h(88.8)
 ③ history 8 → ① 순서대로 h(59.1), h(59.2)

(2) Double hashing 사용

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

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									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 ⇒ 10을 slot 10에 Insert
 ② T₂ : 22%11 = 0 ⇒ 22를 slot 0에 Insert
 ③ T₃ : 31%11 = 9 ⇒ 31를 slot 9에 Insert
 ④ T₄ : 4%11 = 4 ⇒ 4를 slot 4에 Insert
 ⑤ T₅ : 15%11 = 4 ⇒ (충돌 발생) 2번의 추가 탐색 후 slot 5에 삽입
 ⑥ T₆ : 28%11 = 6 ⇒ 28를 slot 6에 Insert
 ⑦ T₇ : 17%11 = 6 ⇒ (충돌 발생) 1번의 추가 탐색 후 slot 3에 삽입
 ⑧ T₈ : 88%11 = 0 ⇒ (충돌 발생) 2번의 추가 탐색 후 slot 7에 삽입
 ⑨ T₉ : 59%11 = 4 ⇒ (충돌 발생) 2번의 추가 탐색 후 slot 2에 삽입

$$\begin{aligned} h(15, 1) &= (15 + 6) \% 11 = 10 \\ h(15, 2) &= (15 + 12) \% 11 = 5 \rightarrow \text{비어있는 slot} \end{aligned}$$

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

$$\begin{aligned} h(88, 1) &= (88 + 9) \% 11 = 9 \\ h(88, 2) &= (88 + 18) \% 11 = 7 \rightarrow \text{비어있는 slot} \end{aligned}$$

$$\begin{aligned} h(59, 1) &= (59 + 10) \% 11 = 3 \\ h(59, 2) &= (59 + 20) \% 11 = 2 \rightarrow \text{비어있는 slot} \end{aligned}$$

3. (교재 22.3 - 5)

Classification of edges

- Tree edges : DFS Tree E_π 내에 존재하는 edge
- Back edges : G_π 내에서 앞의 vertex u와 자손 v를 연결하는 Tree edge가 아닌 edge
- Forward edges : G_π 내에서 앞의 vertex u와 조상 v를 연결하는 edge로, self-loop도 역행 간선으로 간주한다.
- Cross edges : 모든 다른 edge를 의미

DFS Algorithm 내에서
 u의 간선 (u, v)를 살펴볼 때
 v의 색에 따른 간선 분류

- WHITE는 Tree 간선
- GRAY는 역행 간선
- BLACK은 순행 또는 교차간선

Theorem 22.7 (Parenthesis 정리)

graph G=(V, E)의 DFS에서 서로 다른 두 정점 u, v에 대해서
 다음의 세 개의 상황만이 존재할 수 있음

- [u.d, u.f], [v.d, v.f]가 겹치는 부분 x, G_π에서 서로의 자손이 아니다.
- [u.d, u.f]가 [v.d, v.f]를 완전히 포함, G_π에서 v가 u의 자손
- 2의 반대 상황.

(증명) u.d < v.d인 상황 가정

- if v.d < u.f : u가 gray일 때, v가 발견되었으므로, v가 u의 자손 또한 이 경우에 v가 u의 자손이므로, 재귀함 실행에 의해 자동적으로 v.f < u.f
- if u.f < v.d : u가 black이 된 후 v가 발견되므로, 서로가 서로의 자손이 아니다.
 and u.d > v.d 또한 대칭성을 가지므로, 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

(⇒) Parenthesis 정리에서 2에 해당한다. 따라서 G_π에서 v가 u의 자손에 해당한다. v가 u의 자손이므로, edge의 정리에 의해서 Back edge/cross edge 가 될 수 없고, tree edge거나 forward edge이다.

(⇐) (u, v)가 tree edge거나 forward edge이므로, G_π 내에서 v가 u의 자손이다. DFS 내에서 u는 v보다 먼저 발견되며, u가 GRAY 상태에서 v가 발견되므로, u.d < v.d < v.f < u.f가 성립한다.

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

(⇒) Parenthesis 정리에서 3에 해당한다. G_π에서 u와 v의 자손에 해당하며, u와 v의 자손이므로, edge의 정리에 따라서 (u, v)는 back edge이다.

(⇐) (u, v)가 back edge이므로, G_π 내에서 u와 v의 자손이다. DFS 내에서 v는 u보다 먼저 발견된다. v가 GRAY인 상태에서 u가 발견되며, u와 v의 자손이므로, u가 먼저 black이 된다. 즉, v.d ≤ u.d < u.f ≤ v.f가 성립한다.

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

(⇒) Parenthesis 정리에서 1에 해당한다. G_π에서 서로의 자손에 해당하지 않는다. tree edge, back edge, forward edge는 G_π 내에서 조상과 자손을 연결하는 edge이므로 (u, v)는 cross edge에 해당한다.

(⇐) (u, v)가 cross edge이므로, u와 v는 G_π 내에서 서로의 자손에 해당하지 않는다. v가 u보다 먼저 발견되므로, v.d < u.d이며, 또한 서로의 자손에 해당하지 않으므로, v가 gray인 상태에서 u를 발견할 수 없고, v가 black이 된 후에 u가 발견된다. 따라서 v.d < v.f < u.d < u.f가 성립함을 보일 수 있다.

Line 3에서 $u=2$ 인 상태에서 u 가 5로 수행된 이후 상태

(a) r s t x y z
0 5 3 10 7 5 relax 발생 x (π 조한 변화 x)

Line 3에서 $u=5$ 인 상태에서 u 가 5로 수행된 이후 상태

(π) r s t x y z
0 5 3 10 7 5 relax 발생 x (π 조한 변화 x)

Line 3에서 $u=2$ 인 상태로는 수행을 해도 상관없지만 위상정렬 상 마지막 순서이므로, relax가 발생하지 않는다.

Result
→

(a) 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 수행을 바꿔 전의 다익스트라 algorithm이 정확하게 shortest path를 계산한다고 가정하자.

바꿔전과 다른 점은 Q 에 남아 있는 임의의 node u 에 대해서 Line 5-8 과정을 수행하지 않는다는 것이다.

Q 의 마지막 node u 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 $v.d = \delta(S, v)$, 즉 최단거리가 보장되어 있다.

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

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

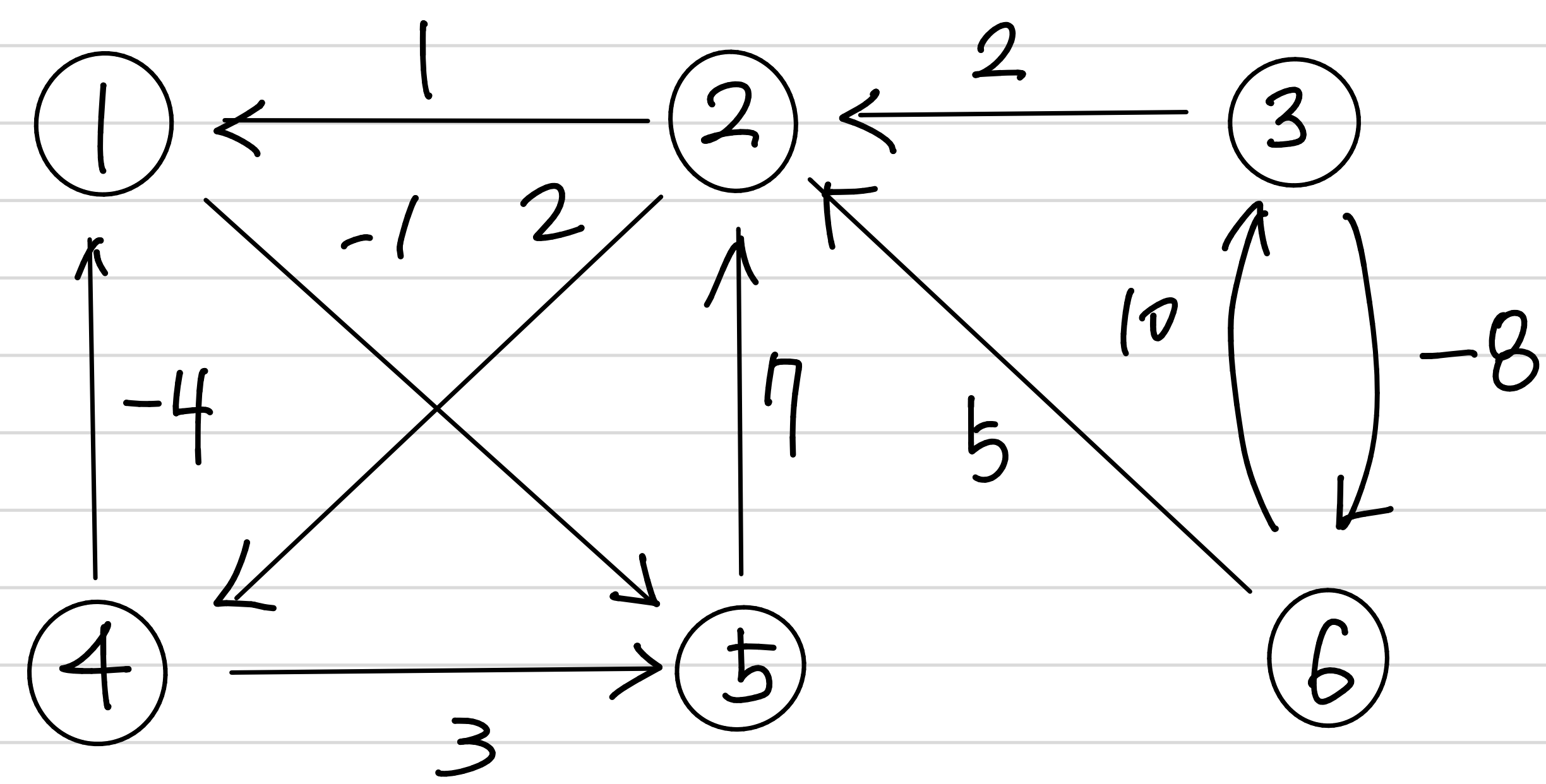
Line 8에서 어떠한 인접 vertex도 u 를 RELAX할 수 없기 때문에 $u.d$ 는 update되지 않은 상태로 계속 ∞ 의 값으로 남아 있다.

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

Line 5에서 Q 의 minimum d 를 갖는 vertex가 u 인 Loop를 생략할 때, 인접 vertex인 u 를 Line 8에서 RELAX하면, 이를 수행한 뒤에 $u.d = \delta(S, u)$ 를 보장할 수 있다.

\therefore 수정된 다익스트라 알고리즘 또한 모든 vertex들의 shortest path를 정확하게 계산하므로, correct 하다.

#8 (교재 25.2-1) python 사용



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

$h(u) = \delta(0, u)$ 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) $\hat{w}(1, 5) = 0$
 $\hat{w}(2, 1) = 3$.

inf가 아닌 값은 $\hat{w}[1][5] = 0, \hat{w}[2][1] = 3, \hat{w}[2][4] = 0$
 $\hat{w}[3][2] = 5, \hat{w}[3][6] = 0, \hat{w}[4][1] = 0, \hat{w}[4][5] = 8$
 $\hat{w}[5][2] = 4, \hat{w}[6][2] = 0, \hat{w}[6][3] = 2$ 이다. 1

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

복원 식: $\delta(u, v) = \hat{\delta}(u, v) + h(v) - h(u)$

```
----- 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 첨부함,


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