자료구조

3. 스택과 큐

6. 해시테이블

3. 스택과 큐

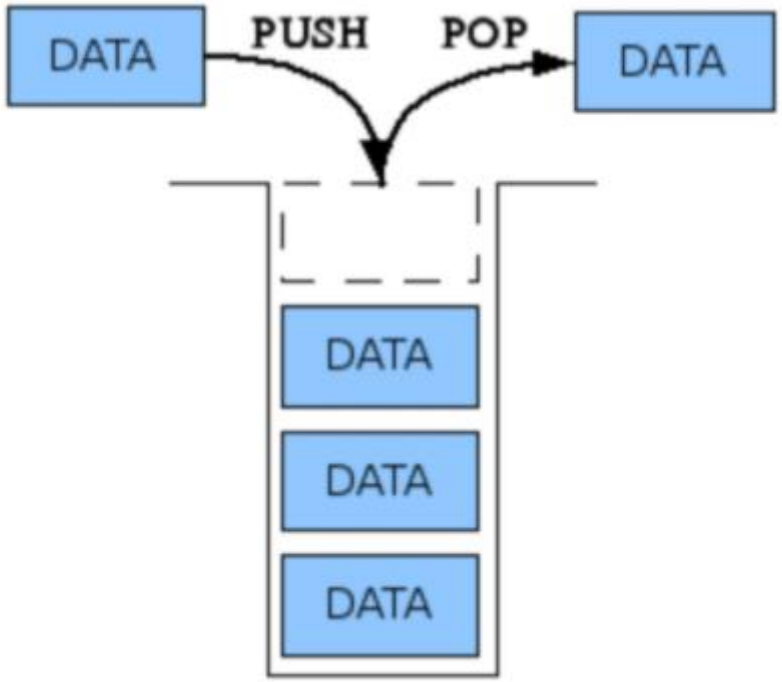
스택

- 스택이란 한 쪽 끝에서만 항목을 삭제하거나 새로운 항목을 저장하는 구조이다.
- 한 쪽 끝에서만 자료를 넣고 뺄 수 있는 LIFO(Last In First Out) 형식의 자료 구조
- 가장 최근에 스택에 추가한 항목이 가장 먼저 제거될 항목이다.

- pop(): 스택에서 가장 위에 있는 항목을 제거한다.
- push(item): item 하나를 스택의 가장 윗 부분에 추가한다.
- peek(): 스택의 가장 위에 있는 항목을 반환한다.
- isEmpty(): 스택이 비어 있을 때에 true를 반환한다.

- 스택의 item 수가 배열의 ¼만 차지하게 될 때, 메모리 낭비를 줄이기 위해 배열의 크기를 ½로 축소시킨다.

스택



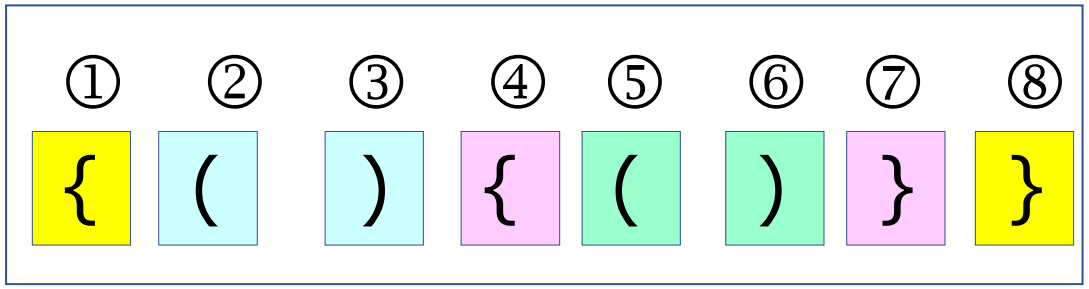
- 스택은 특히 자료의 출력 순서가 입력 순서의 역순으로 이루어져야 할 경우에 사용하기 좋다
- 예를 들어 (A,B,C,D) 순서로 스택에 입력을 했다가 스택에서 꺼내게 된다면 (D,C,B,A) 처럼 역순 으로 만드는 것이 가능
- 역순으로 만들 수 있다는 것은 어떤 것을 되돌리려는 기능을 구현할 때 유용하다.
- 컴퓨터 안에서 수많은 함수 호출을 하게 되는데 함수는 실행이 끝나면 자신을 호출한 함수로 돌아가야 한다. 이때 스택은 함수가 복귀할 주소 를 기억하는 데 사용된다.

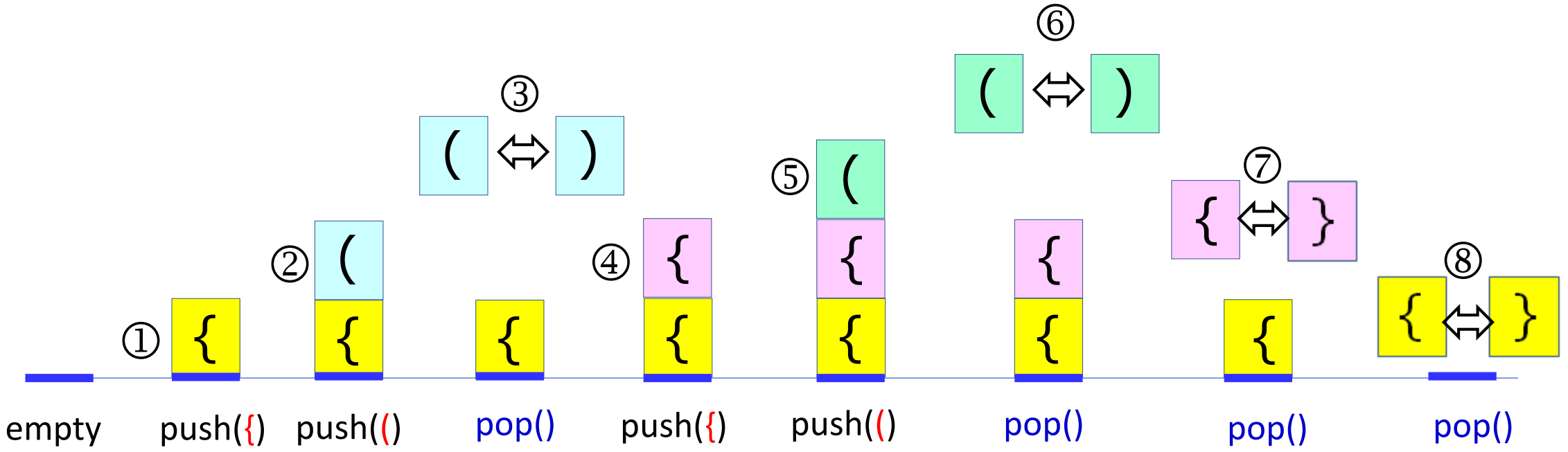
- 배열로 구현한 스택의 push와 pop 연산은 각각 O(1) 시간이 소요
- 배열 크기를 확대 또는 축소시키는 경우에 스택의 모든 item들을 새 배열로 복사해야 하므로 O(N) 시간이 소요
- 상각분석: 각 연산의 평균 수행시간은 O(1) 시간
- 단순연결리스트로 구현한 스택의 push와 pop 연산은 각각 O(1) 시간이 걸리는데, 연결리스트의 앞 부분에서 노드를 삽입하거나 삭제하기 때문

스택

괄호 짝맞추기

왼쪽 괄호는 스택에 push
오른쪽 괄호를 읽으면 pop 수행

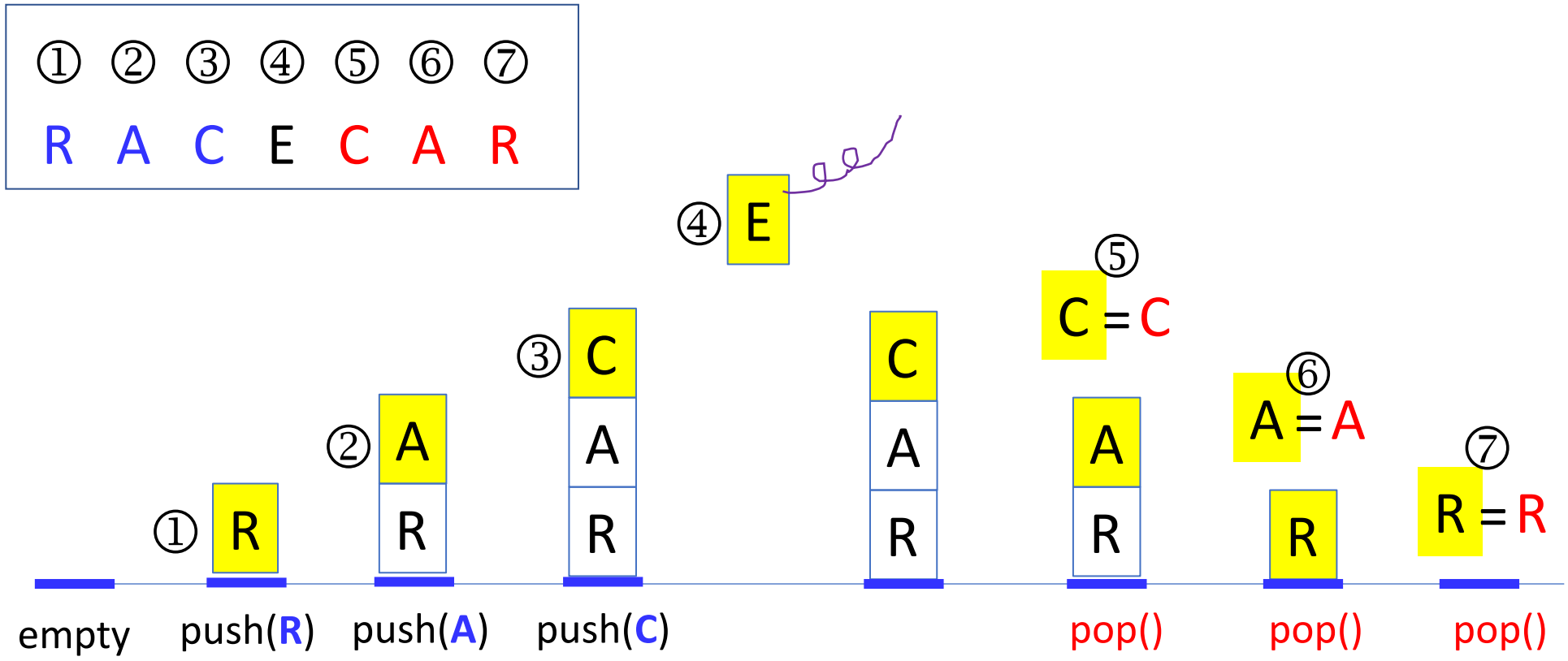




스택

회문 검사하기

전반부의 문자들을 스택에 push한 후,
후반부의 각 문자를 차례로 pop한 문자와 비교

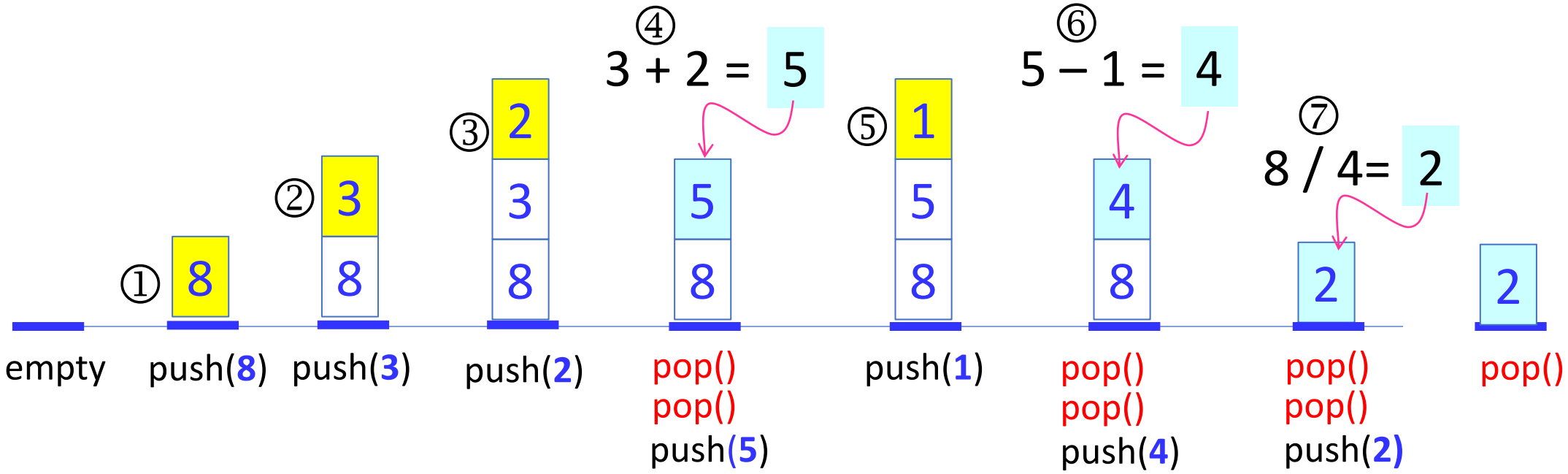


스택

후위표기법 계산

피연산자는 스택에 push하고,
연산자는 2회 pop하여 계산한 후 push

①	②	③	④	⑤	⑥	⑦
8	3	2	+	1	−	/



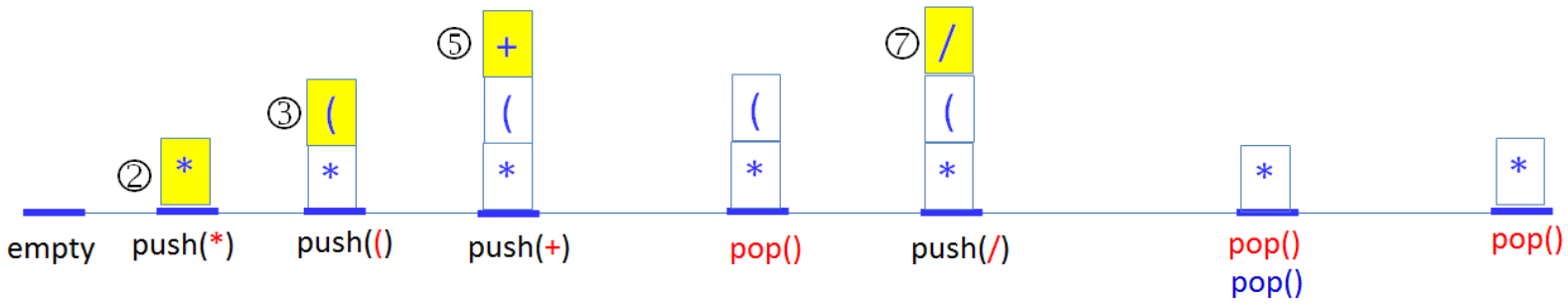
스택

중위표기법 수식을 후위표기법으로 변환

왼쪽 괄호나 연산자는 스택에 push하고,
피연산자는 출력

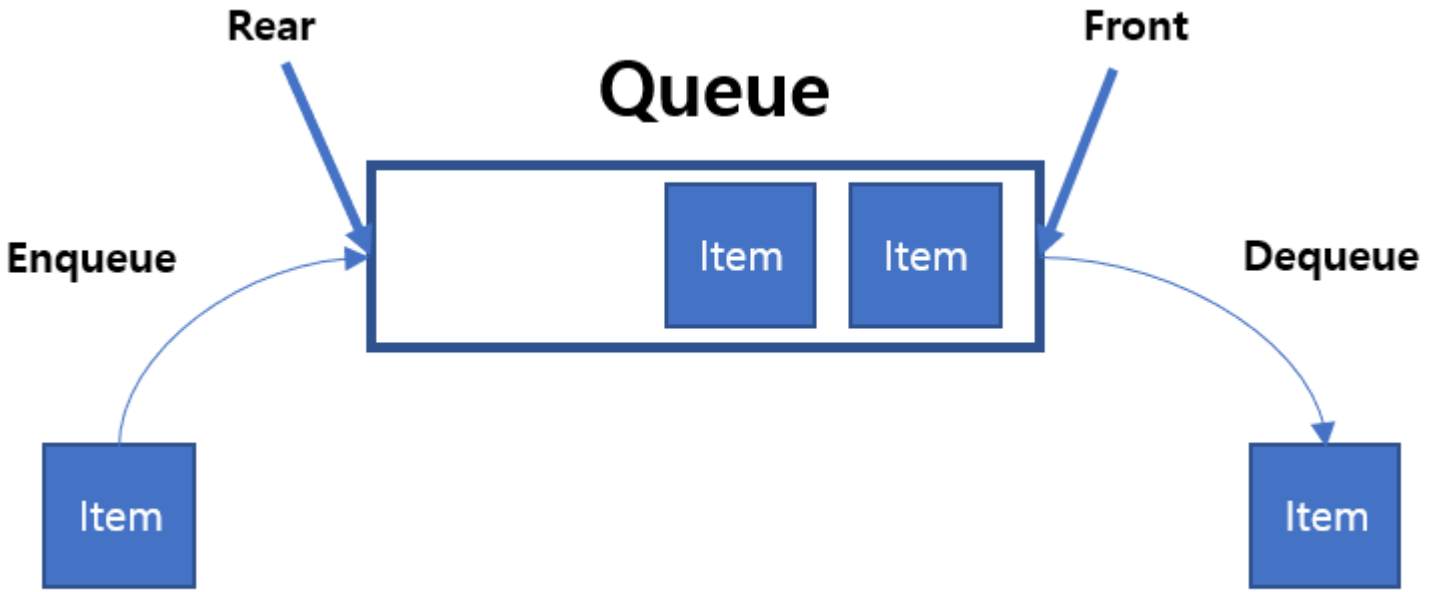
①	②	③	④	⑤	⑥	⑦	⑧	⑨
A	*	(	B	+	C	/	D	)

출력: A (①) A B (④) A B C (⑥) A B C + (⑦) A B C + D (⑧) A B C + D / (⑨) A B C + D / *



큐

- 큐는 한쪽 끝(rear)에서는 삽입연산만 이루어지며 다른 한쪽 끝(front)에서는 삭제연산만 이루어지는 유한 순서 리스트이다.
- 구조상 먼저 삽입된 item이 먼저 삭제가 이루어진다. (FIFO)
- 일상생활의 관공서, 은행, 우체국, 병원 등에서 번호표를 이용한 줄서기가 대표적인 큐

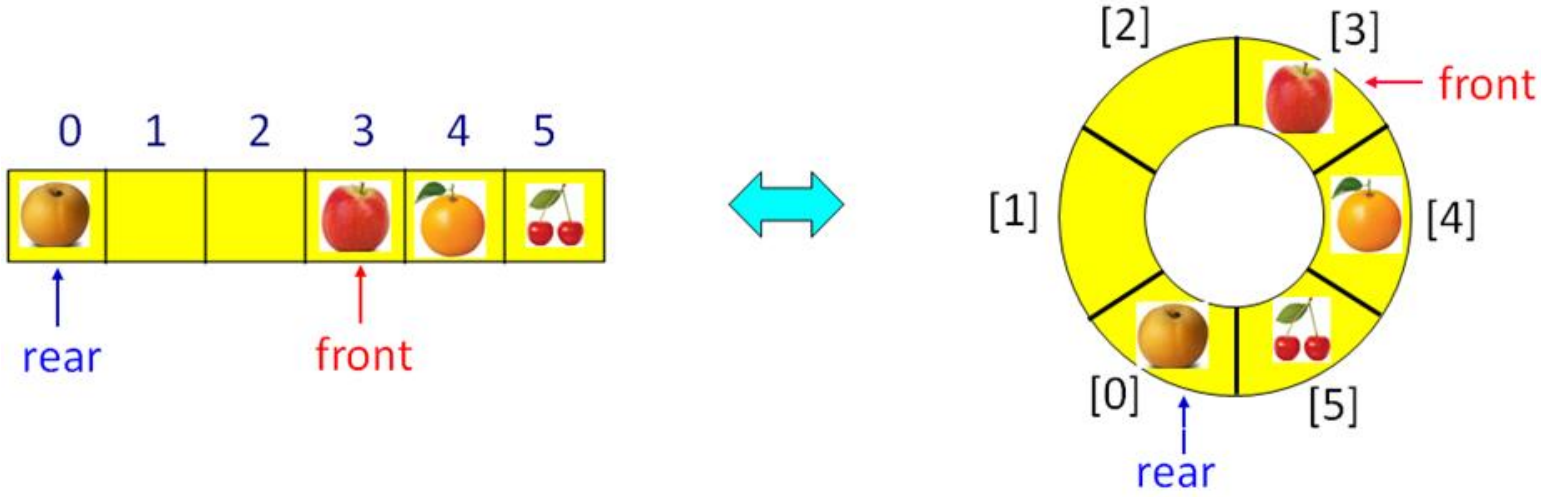


큐 문제점

- 큐에 데이터가 삽입이 되며 점차 rear가 증가하게 되면 결국 full상태가 된다. 하지만 이때 큐에 원소가 꽉 차있지 않을 수 있다. front에서 삭제가 일어났다면 그만큼 공간이 비었기 때문이다. 따라서 만원(Full) 상태가 되었을 때에는 첫번째 원소의 위치를 큐의 [0]번 인덱스부터 위치되게 한뒤 이것을 기준으로 rear이 위치도 다시 정해주어야 한다.
- 위의 문제로 순차 표현 큐는 많은 비용(큐 원소 이동에 따른 지연시간)을 발생하게 된다.

큐 문제점 해결방안

- 선형 배열을 원형 배열로 변환한다.



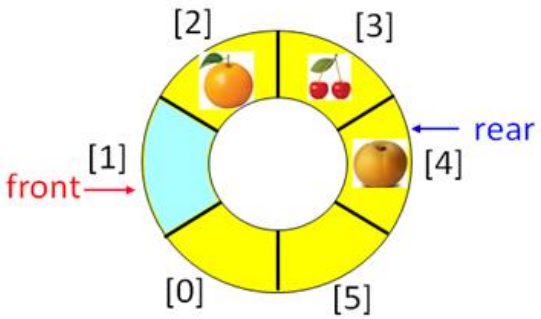
  - 배열의 앞뒤가 맞닿아 있다고 생각하기 위해 배열의 rear 다음의 비어있는 원소의 인덱스

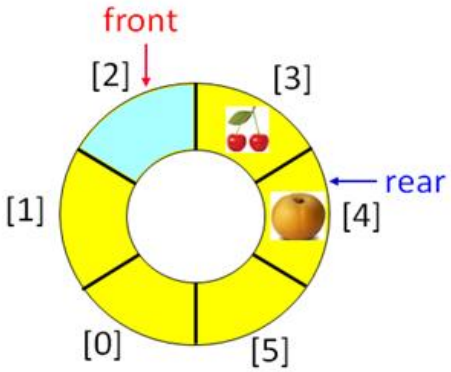
$$rear = (rear + 1) \% N$$

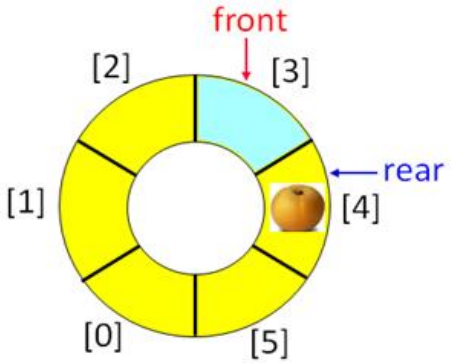
  - 여기서 N은 배열의 크기이다.

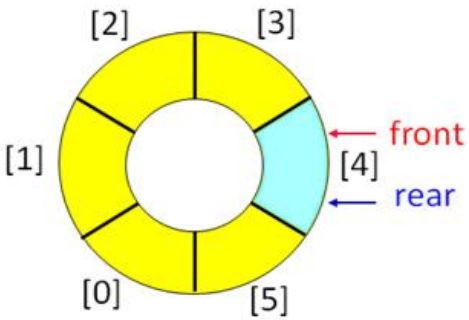
큐 문제점 해결방안

- 선형 배열을 원형 배열로 변환한다.









- 배열의 크기가 N이라면 실제로는 N-1 개의 공간만 item들을 저장하는데 사용
- Empty가 되면 front==rear가 된다.

데크

- 데크(Double-ended Queue, Deque): 양쪽 끝에서 삽입과 삭제를 허용하는 자료구조
- 데크는 스택과 큐 자료구조를 혼합한 자료구조
- 스택과 큐를 동시에 구현하는데 사용

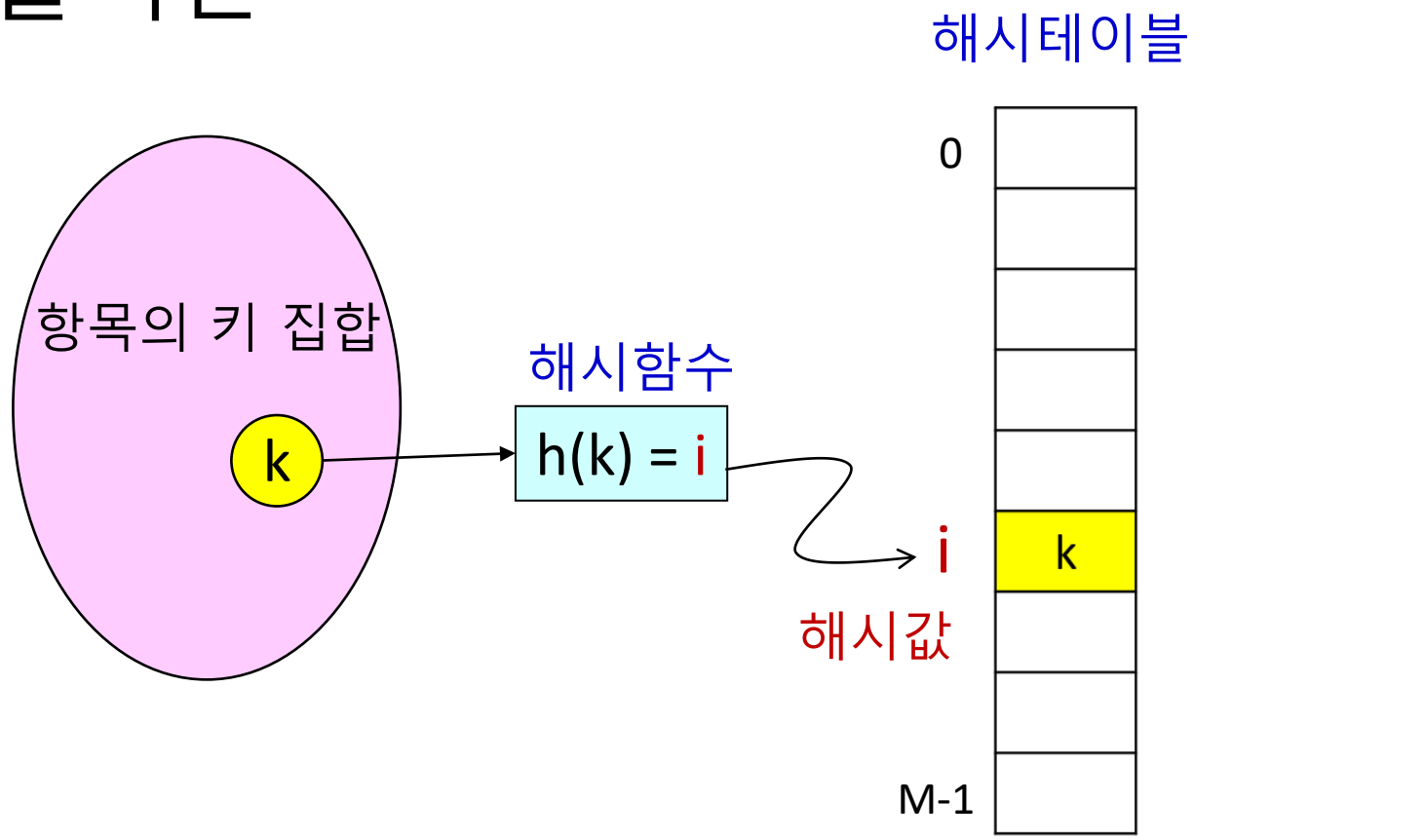
- 웹 브라우저의 방문 기록 등에 쓸 수 있음
- 수행시간은 데크를 배열이나 이중연결리스트로 구현한 경우, 스택과 큐의 수행시간과 동일

6. 해시테이블

해시테이블이란

- O(logN) 시간보다 빠른 연산을 위해, 키와 1차원 배열의 인덱스의 관계를 이용하여 키(항목)를 저장한다.
- 키를 배열의 인덱스로 그대로 사용하면 메모리 낭비가 심해질 수 있어서 키를 변환하여 배열의 인덱스로 사용한다.
- 키를 간단한 함수를 사용해 변환한 값을 배열의 인덱스로 이용하여 항목을 저장하는 것을 해싱(Hashing)이라고 한다.
- 항목이 해시값에 따라 저장되는 배열을 해시테이블(Hash Table)이라고 함

해시테이블이란



M = 해시테이블 크기

해시함수

- 가장 이상적인 해시함수는 키들을 균등하게(Uniformly) 해시테이블의 인덱스로 변환하는 함수
- 균등하게 변환한다는 것은 키들을 해시테이블에 랜덤하게 흩어지도록 저장하는 것을 뜻함
- 해시함수는 키들을 균등하게 해시테이블의 인덱스로 변환하기 위해 의미가 부여되어 있는 키를 간단한 계산을 통해 '뒤죽박죽' 만든 후 해시테이블의 크기에 맞도록 해시값을 계산
- 함수 계산 자체에 긴 시간이 소요된다면 해싱의 장점인 연산의 신속성을 상실하므로 그 가치를 잃음

해시함수의 종류

1.가장 널리 사용되는 해시함수로는 나눗셈 함수가 있음.

- 나눗셈 함수는 키를 소수(Prime) M으로 나눈 뒤, 그 나머지를 해시값으로 사용
- h(key) = key % M이고, 따라서 해시테이블의 인덱스는 0에서 M-1이 됨
- 여기서 제수로 소수를 사용하는 이유는 나눗셈 연산을 했을 때, 소수가 키들을 균등하게 인덱스로 변환시키는 성질을 갖기 때문

2. 중간제곱(Mid-square) 함수: 키를 제곱한 후, 적절한 크기의 중간부분을 해시값으로 사용

3. 접기(Folding) 함수: 큰 자릿수를 갖는 십진수를 키로 사용하는 경우, 몇 자리씩 일정하게 끊어서 만든 숫자들의 합을 이용해 해시값을 만든다.

예를 들어, 123456789012에 대해서 1234 + 5678 + 9012 = 15924를 계산한 후에 해시테이블의 크기가 3자리 수라면 15924에서 3자리 수만을 해시값으로 사용

해시함수의 종류

4. 곱셈(Multiplicative) 함수: 1보다 작은 실수 를 키에 곱하여 얻은 숫자의 소수부분을 테이블 크기 M과 곱한다. 이렇게 나온 값의 정수부분을 해시값으로 사용

h(key) = (int)(((key * δ) % 1) * M)이다. Knuth에 의하면 δ = (√5−1)/2 0.61803이 좋은 성능을 보인다.

예를 들면, 테이블 크기 M = 127이고 키가 123456789인 경우, 123456789 x 0.61803 = 76299999.30567, 0.30567 x 127 = 38.82009이므로 38을 해시값으로 사용

자바의 hashCode()

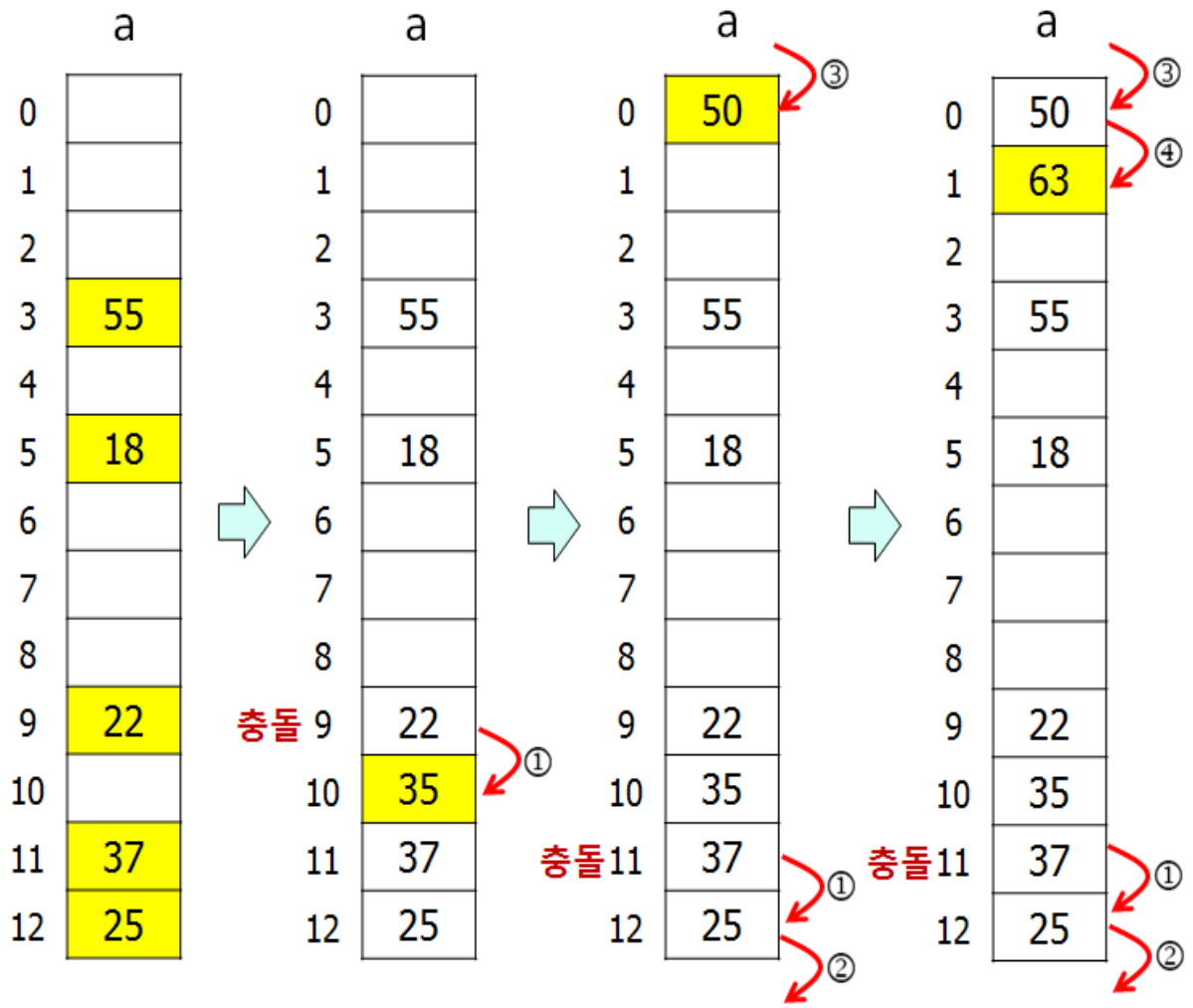
- 자바의 모든 클래스는 32비트 int를 리턴하는 hashCode()를 포함하고 있고, hashCode()는 객체를 int로 변환하는 메소드
- 2 개의 키가 동일하면 각각의 hashCode 값도 같아야 함
- Integer 객체의 경우 hashCode()는 아무런 계산 없이 key를 그대로 리턴
- Boolean 객체의 hashCode()는 key가 true이면 1231, false이면 1237을 각각 리턴
- Double 객체의 hashCode()는key를IEEE 64-bit 포맷으로 변환시킨 후, 모든 bit를 계산에 참여시키기 위해 최상위 32 bit와 최하위 32 bit를 XOR한 결과를 리턴
- String 객체는 key의 문자(char)를 31진수의 숫자로 간주하여 해시값을 계산

개방주소방식

- 개방주소방식(Open Addressing)은 해시테이블 전체를 열린 공간으로 가정하고 충돌된 키를 일정한 방식에 따라서 찾아낸 empty 원소에 저장
- 대표적인 개방주소방식:
- 선형조사(Linear Probing)
- 이차조사(Quadratic Probing)
- 랜덤조사(Random Probing)
- 이중해싱(Double Hashing)

선형조사

key	h(key) = key % 13
25	12
37	11
18	5
55	3
22	9
35	9
50	11
63	11



선형조사의 단점

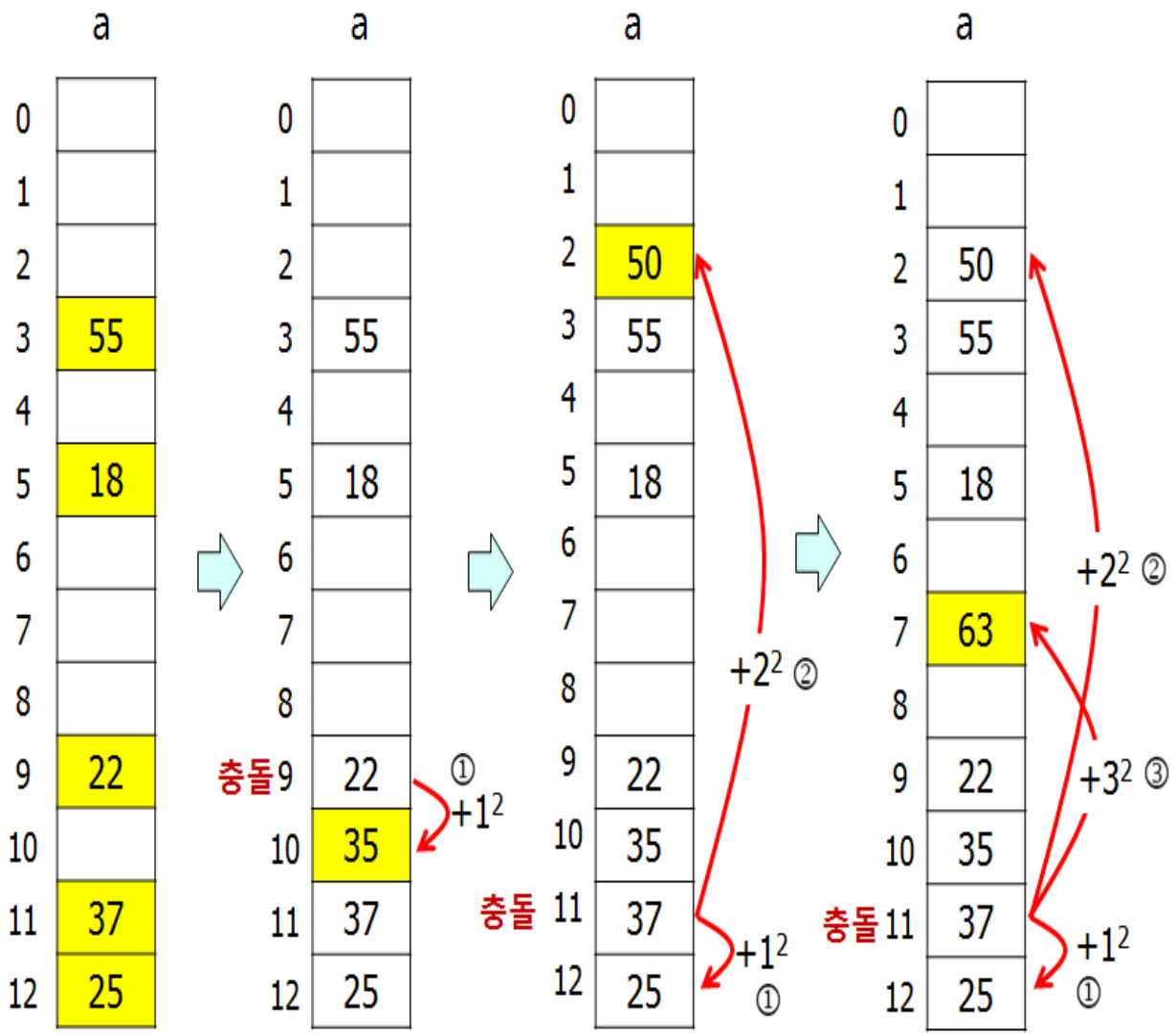
선형조사는 순차탐색으로 empty 원소를 찾아 충돌된 키를 저장하므로 해시테이블의 키들이 빈틈없이 뭉쳐지는 현상이 발생[1차 군집화(Primary Clustering)]

이러한 군집화는 탐색, 삽입, 삭제 연산 시 군집된 키들을 순차적으로 방문해야 하는 문제점을 야기

군집화는 해시테이블에 empty 원소 수가 적을수록 더 심화되며 해시성능을 극단적으로 저하시킴

이차조사

key	h(key) = key % 13
25	12
37	11
18	5
55	3
22	9
35	9
50	11
63	11



이차조사의 단점

이차조사는 이웃하는 빈 곳이 채워져 만들어지는 1차 군집화 문제를 해결하지만,

같은 해시값을 갖는 서로 다른 키들인 동의어(Synonym)들이 똑같은 점프 시퀀스(Jump Sequence)를 따라 empty 원소를 찾아 저장하므로 결국 또 다른 형태의 군집화인 2차 군집화(Secondary Clustering)를 야기

점프 크기가 제곱만큼씩 커지므로 배열에 empty 원소가 있는데도 empty 원소를 건너뛰어 탐색에 실패하는 경우도 피할 수 없음

랜덤조사

랜덤조사(Random Probing)는 선형조사와 이차조사의 규칙적인 점프 시퀀스와는 달리 점프 시퀀스를 무작위화 하여 empty 원소를 찾는 충돌해결방법

랜덤조사는 의사 난수 생성기를 사용하여 다음 위치를 찾음

랜덤조사 방식도 동의어들이 똑같은 점프 시퀀스에 따라 empty 원소를 찾아 키를 저장하게 되고, 이 때문 3차 군집화(Tertiary Clustering)가 발생

이중해싱

이중해싱(Double Hashing)은 2 개의 해시함수를 사용

하나는 기본적인 해시함수h(key)로 키를 해시테이블의 인덱스로 변환하고, 제2의 함수 d(key)는 충돌 발생 시 다음 위치를 위한 점프 크기를 다음의 규칙에 따라 정함

이중해싱은 동의어들이 저마다 제2 해시함수를 갖기 때문에 점프 시퀀스가 일정하지 않음

따라서 이중해싱은 모든 군집화 문제를 해결

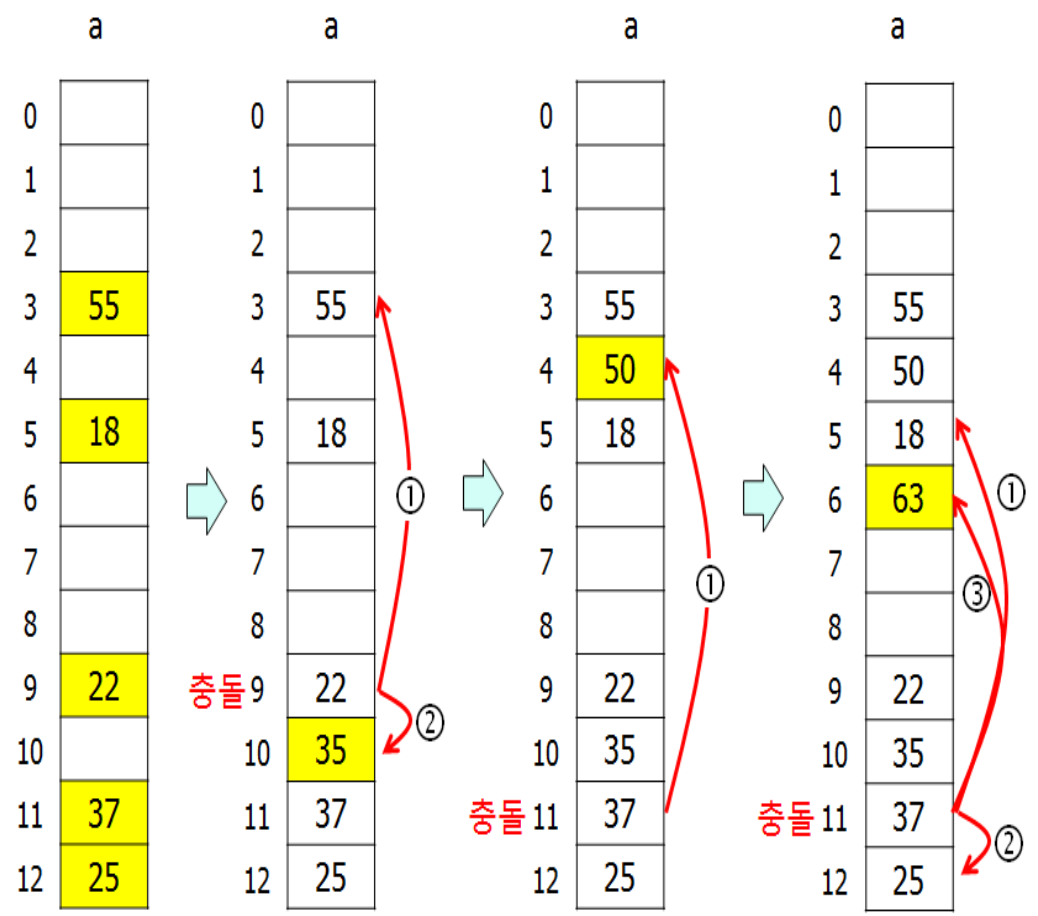
- h(key) = key % 13과 d(key) = 7-(key % 7) 에 따라, 25, 37, 18, 55, 22, 35, 50, 63을 해시테이블에 차례로 저장하는 과정

key	h(key)	d(key)	(h(key) + j*d(key)) % 13		
			j=1	j=2	j=3
25	12				
37	11				
18	5				
55	3				
22	9		①	②	
35	9	7	3	10	
50	11	6	4		③
63	11	7	5	12	6

h(key) = key % 13

d(key) = 7 – (key % 7)

(h(key) + j*d(key)) % 13, j = 0, 1, …

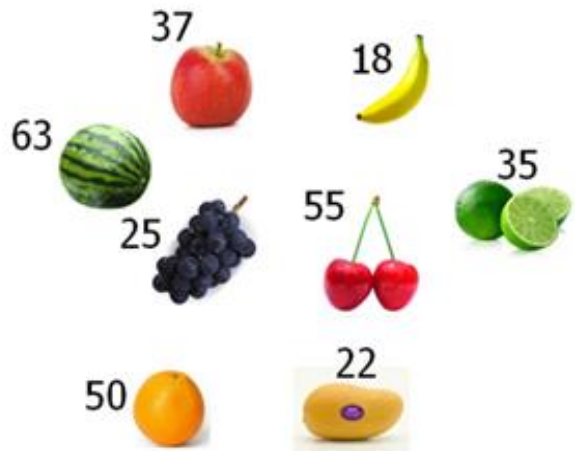


폐쇄주소방식

폐쇄주소방식(Closed Addressing)의 충돌해결 방법은 키에 대한 해시값에 대응되는 곳에만 키를 저장

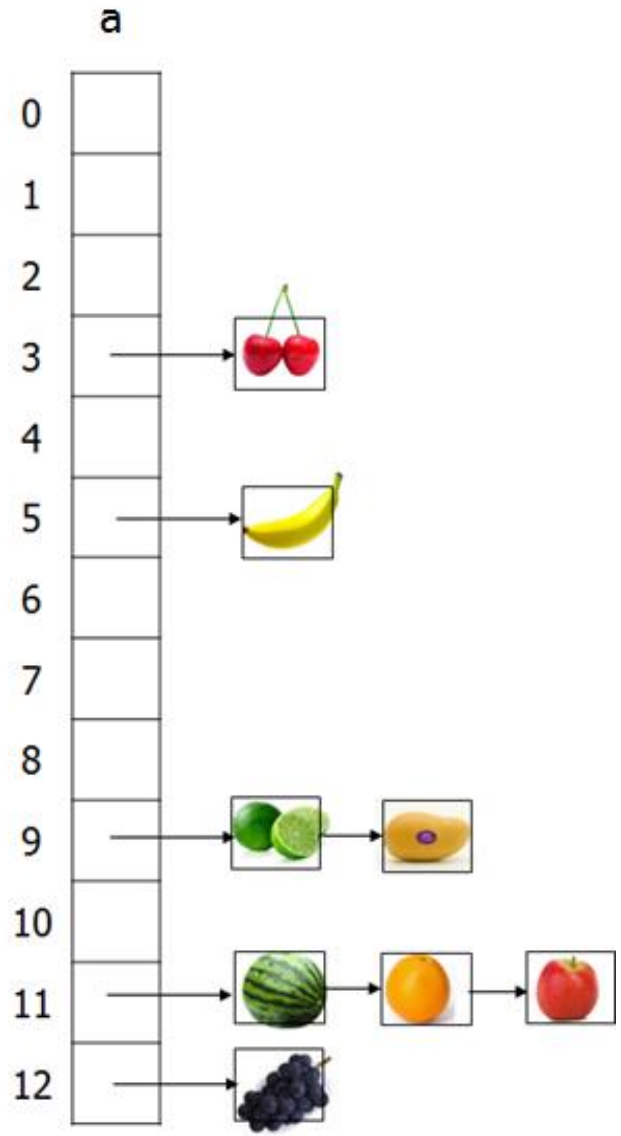
충돌이 발생한 키들은 한 위치에 모여 저장

이를 구현하는 가장 대표적인 방법: 체이닝(Chaining)



h(key) = key % 13

key	h(key)
25	12
37	11
18	5
55	3
22	9
35	9
50	11
63	11



참조

- 자바와 함께하는 자료구조의 이해 – 양성봉