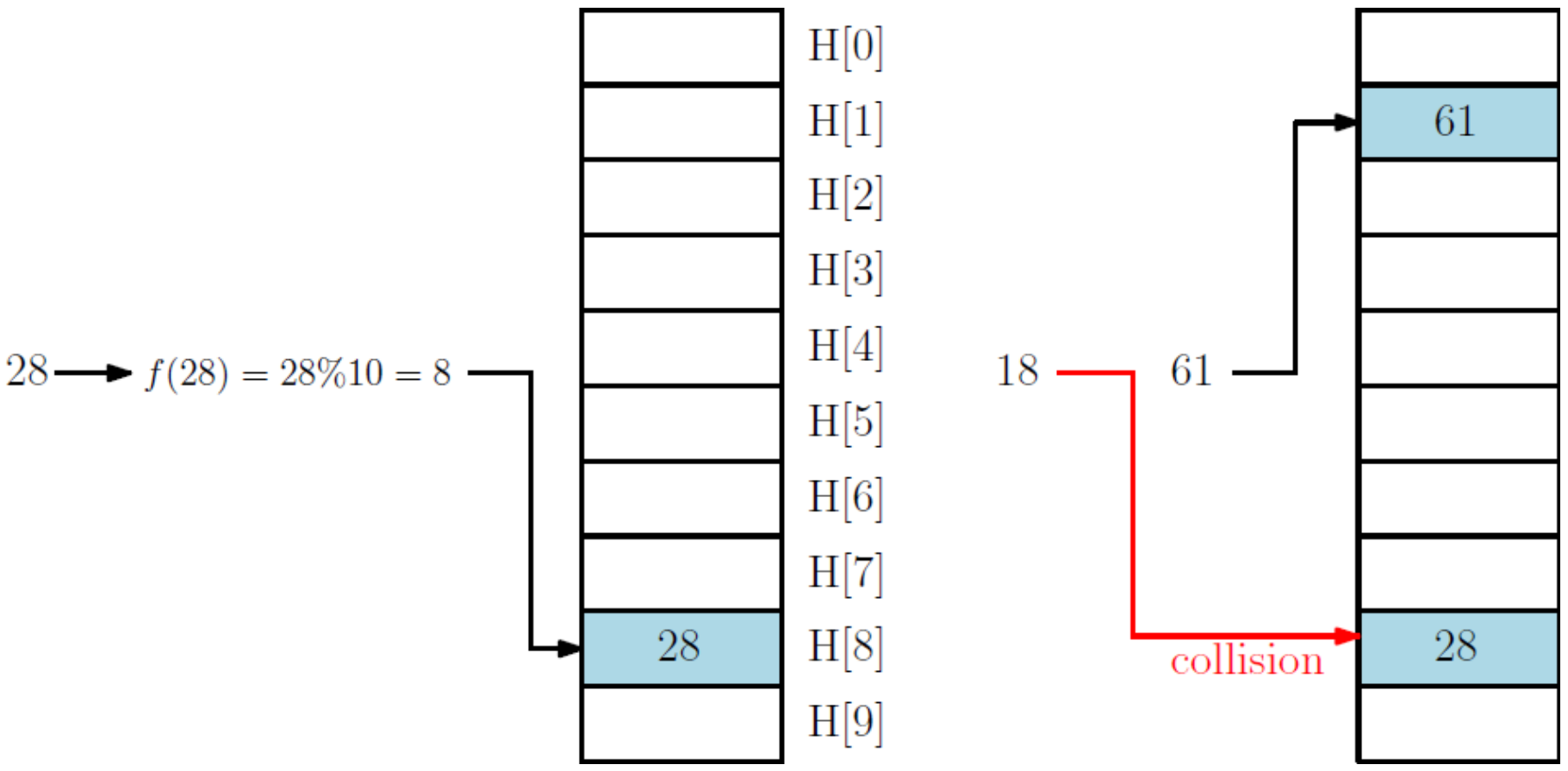
해시테이블(Hash Table)

매우 빠른 평균 삽입, 삭제, 탐색 연산 제공

1. 정보를 담아 저장하는 서랍장에 비유 가능
    a. ex) 정보 A는 3번째 서랍에 저장, B는 0번째, C는 다시 3번째, D는 4번째에 저장하는 식
    b. 정보 C를 찾고 싶다면, C가 저장된 서랍 번호를 알아내, 저장된 3번째 서랍에 들어 있는 정보들을 하나씩 비교해 C를 찾으면 됨
    c. 가장 핵심적인 과정은 각 정보를 몇번째 서랍에 넣을지를 결정하는 것
2. 정보 K(key 값)가 저장될 서랍장(slot) 번호를 계산하는 함수 f( )를 해시 함수라고 한다.
    a. ex) 해시 테이블 H를 일차원 배열 int H[10]으로 선언하여 사용한다고 하자
    b. 해시 함수는 f(K) = k % 10으로 정의된다고 하자
    c. K = 28을 저장하고 싶다면, H[f(28)] 슬롯에 저장된다. 즉 H[8]에 저장
    d. K = 61은 H[f(61)] = H[1]에 저장
    e. K = 18도 H[f(18)] = H[8]에 저장되어야 하나 이미 H[8]에는 28이 저장되어 있다. 이런 경우를 **충돌(collision)**이 발생했다고 한다.
    f. 충돌이 발생한 경우에 18을 저장할 공간이 더 있으면 저장하면 되지만, 이 예시처럼 값 하나만 저장할 수 있는 경우에는 18을 다른 곳에 저장해야 한다. 다른 곳에 저장하는 방법을 **충돌해결방법**이라 부른다.



해시 함수(Hash Function)

- Perfect h.f : ideal h.f = 비현실적
- c-universal h.f:
    - prob(f(x) == f(y)) = c/size(H)이 성립

현실에서 자주 쓰이는 해시 함수들 (1)

1. **Division** : f(k) = (k mod p) mode m (p : 소수)

- key 값들의 성질이 잘 알려져 있지 않은 경우 유용

2. **Folding** : key 값의 digit를 나눠 연산하는 형식
   - shift folding : 예 : 계좌번호 k = 1253-387-601 → 두 digit씩 나눠 모두 더한 후 mod m → (12 + 54 + 38 + 76 + 01) mod m
   - boundary folding : 여러 digit로 나눈 후, 더하는데 짝수 번 조각은 거꾸로 해서 더함. ex) 12 + 45 + 38 + 67 + 01) mod m
3. **Mid-Square** : key 값을 적당히 연산한 후, 그 결과의 중간 부분을 떼어나 주소로 이용
   - ex) m = 1000이라면, k = 3121이라면, 3121^2 = 9740641이 되고 중간에 3 digit를 떼어낸 406이 주소가 됨.
4. **Extraction** : key 값의 각 파트마다 임의의 digit를 떼어내 연결하는 식 계산
   - ex) 계좌번호가 1254-387-601이라면 1254에서 12, 601에서 1을 떼어낸 후 서로 붙여 121을 만듦. 121이 주소가 됨

현실에서 자주 쓰이는 해시 함수들(2) - key값이 string일 때 사용

1. key[i]은 i번째 문자(또는 숫자()의 값으로 ascii 코드 값 정도로 해석
2. Additive hash: key[i]의 단순 합
3. Rotating hash: <<, >> (비트 쉬프트) 연산과 ^(exclusive or) 연산을 반복
4. Universal hash:

좋은 해시 함수란?

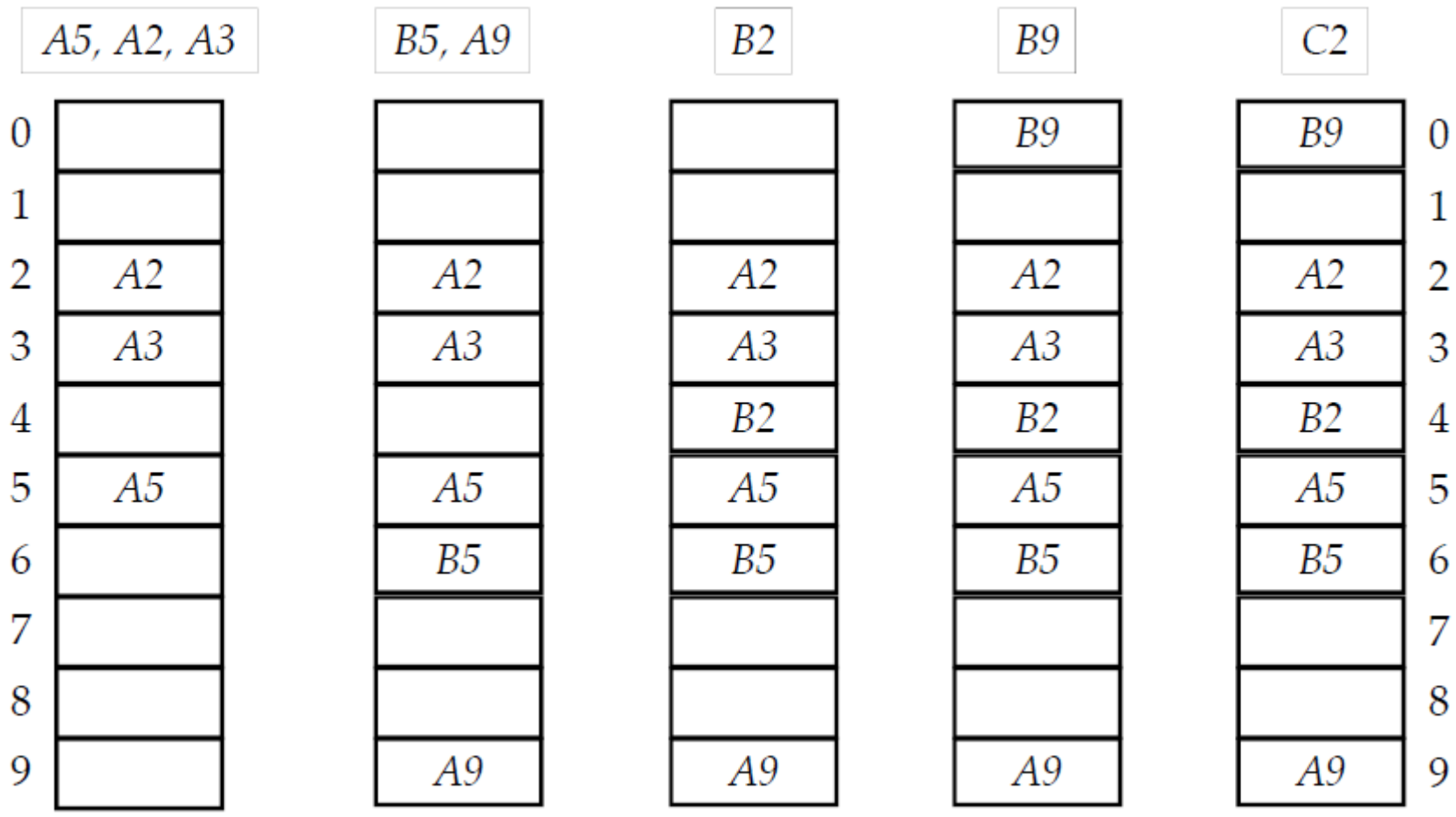
1. 충돌이 적어야 한다. (완전해시함수(perfect.h.f가 아닌 이상 충돌을 피할 수는 없지만 되도록 충돌이 적게 발생하는 해시 함수를 선택해야 함)
2. 빠르게 계산할 수 있어야 한다. (해시 함수 값을 자주 계산해야 하기 때문)

충돌해결: open addressing

충돌 해결 방법(collision resolution methods)

- 서로 다른 key 값 x, y에 대해, f(x) = f(y)가 된다면 두 key 값은 충돌이 발생했다고 정의
- 이 경우엔 두 값을 해쉬 테이블에 저장할 수 있는 방법 - 충돌해결방법이 필요
- Open addressing과 Chaining 두 가지 방법이 일반적이다.

Open addressing 방법: linear probing



1. H의 slot에 값 하나만 저장할 수 있다고 가정
2. A5, A2, A3에 key 값이 저장이 되어 있음
    a. 그 다음으로 B5, A9가 저장이 됨
3. 그 다음 B2, B9가 저장이 되어야 하는데 이미 H의 slot에 값이 들어가 있으므로 다른 곳에 저장
4. open addressing 방법은 아래쪽으로 slot을 차례로 탐색하면서 가장 먼저 발견된 빈 slot에 저장하는 것
5. B5는 그래서 H[6]에 저장
6. B2는 찾다가 H[4]가 비어있으니 거기에 저장
7. B9에 대해선 H[9]가 선점되어 있으니 다음 slot을 점검 한바퀴 돌아서 H[0]에 저장

- find_slot(key)
    - key 값을 갖는 아이템을 찾아 슬롯 번호(index)를 리턴. 만약, 그런 아이템이 없다면, 해당 아이템이 저장될 슬롯 번호를 리턴
    - 만약 key 값을 갖는 슬롯이 존재하지도 않고 빈 슬롯도 없다면 FULL을 리턴

- set(key, value)
    - key 값을 갖는 아이템이 이미 테이블에 있다면, 해당 아이템의 value를 매개변수 value 값으로 수정하고, 없다면 새 아이템 (key, value)를 삽입하는 연산
    - 정상적으로 수정 또는 삽입이 이루어졌다면, key 값을 그대로 리턴하고, 테이블에 빈 슬롯이 없어 삽입을 하지 못했다면 FULL 리턴
    - 이를 위해, open addressing 방법에 따라 key 값을 갖는 아이템을 찾거나 빈 슬롯을 찾아 H의 인덱스를 리턴하는 find_slot(key) 함수 필요

```
def find_slot(key):
  i = f(key)
  start = i
  while ( H[i] is occupied ) and ( H[i].key ≠ key )
    i = (i + 1) % m
    if(i == start) return FULL
  return i

def set(key, value):
  i = find_slot(key)
  if i == FULL return FULL
  if H[i] is occupied:  # 이미 key 값을 갖는 item이 H에 존재함 (수정)
    H[i].value = value  # value 값 update 후 리턴
    return key
```

```
# H[i]가 비어있는 경우, 즉 key 값을 갖는 item이 없다면 새로 저장함 (삽입)
if the table is almost full:  # if m < 2n (여기서 n은 테이블에 저장된 값 갯수)
  rebuild the table larger (usually m is doubled!) and copy items into new H
  i = find_slot(key)
H[i].key = key
H[i].value = value
return key
```

- remove(key)
  - key 값을 갖는 아이템을 find_slot을 이용해 찾는다. i = find_slot(key)라 하자.
  - H[i]가 비었다면 삭제할 아이템이 실제로 존재하지 않는 경우이므로 NOTFOUND 리턴
  - H[i] 가 존재한다면, 이 아이템 때문에 아래쪽으로 밀려서 저장된 아이템들을 연쇄저그로 위로 올려 이동해야 한 후, 성공적인 삭제가 수행되었다는의미에서 key값 자체를 리턴
    - H[1]는 현재 빈 슬롯이고 ,아래쪽 H[j]에 있는 아이템을 H[i]로 이동할지를 결정해야 한다.
    - H[j].key 값의 해시 함수 값을 k라 하자. 이 k 값이 [i, j]에 있다면 (즉, ..i..k..j.. 순서라면) H[j]를 H[i]로 옮기면 안된다. Why?
    - 또한 해시테이블이 원형 배열과 같기 때문에 i > j 일 수도 있으므로, ..j..i..k.. 순서라거나, ..k..j..i..인 경우에도 같은 이유로 옮기면 안된다.
    - 위의 경우가 아니라면 H[j]를 H[i]로 옮긴다. 그러면 이제 H[j]가 빈 슬롯이 되고, 같은 일을 반복한다.

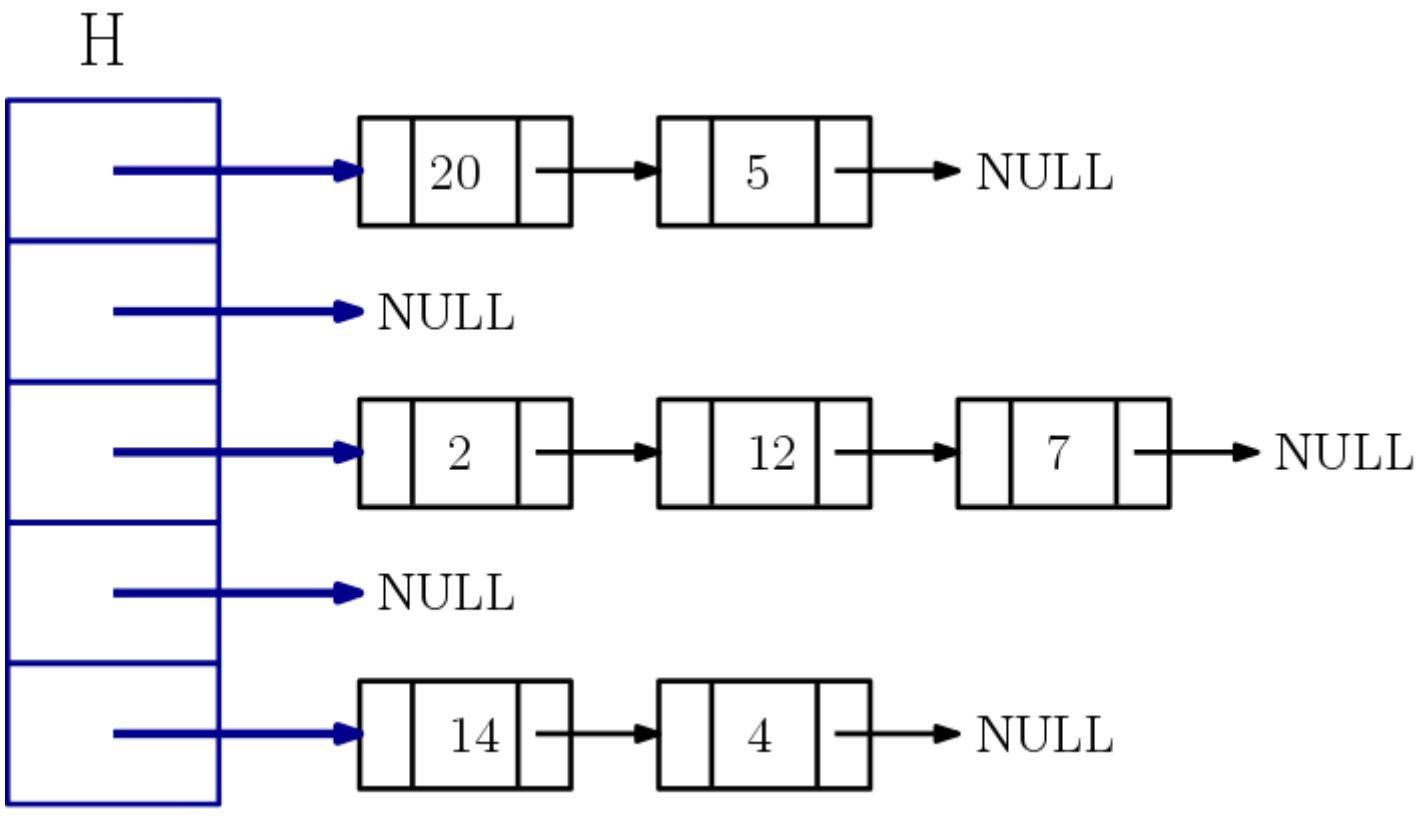
```
remove(key):
  i = find_slot( key )
  if H[i] is unoccupied  // 삭제할 아이템이 실제로 존재하지 않는 경우
    return NOTFOUND
  j = i
  while True:
    mark H[i] as unoccupied
    while True:
      j = (j+1) % m
      if H[j] is unoccupied  // 자리 이동 완료!
        return key
      k = f(H[j].key)
      # |    i..k..j |
      # |....j..i..k..| or |..k..j..i..|
      if not ( i < k <= j or j < i < k or k <= j < i): # H[j] --> H[i]
        break
    H[i] = H[j]
    i = j
```

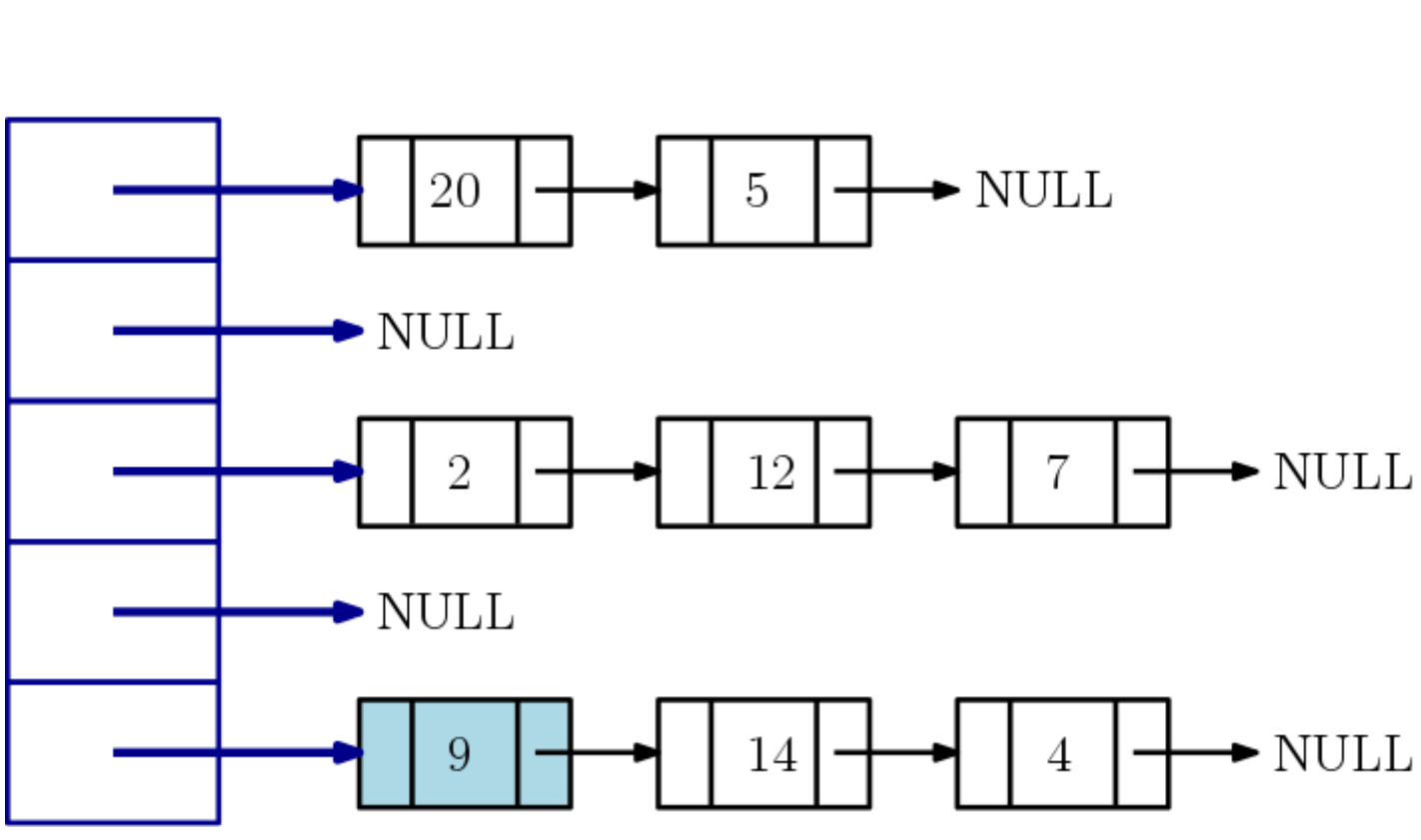
- search(key)
  - key 값을 찾는 아이템을 찾아 value 값을 리턴하고, 없다면 NOTFOUND를 리턴함

```
remove(key):
  i = find_slot( key )
  if H[i] is unoccupied  // 삭제할 아이템이 실제로 존재하지 않는 경우
    return NOTFOUND
  j = i
  while True:
    mark H[i] as unoccupied
    while True:
      j = (j+1) % m
      if H[j] is unoccupied  // 자리 이동 완료!
        return key
      k = f(H[j].key)
      # |    i..k..j |
      # |....j..i..k..| or |..k..j..i..|
      if not ( i < k <= j or j < i < k or k <= j < i): # H[j] --> H[i]
        break
    H[i] = H[j]
    i = j
```

충돌해결: chaning

1. H의 slot에 값 몇 개만 저장하도록 하는게 아니라, 각 slot마다 연결리스트를 가지도록 해, 이론적으론 무한히 많은 값들을 저장하도록 하는 방법
2. 간단한 구조의 한방향 연결리스트를 활용하는 것이 일반적
    a. H = [None] * m 처럼 해시 테이블을 m개의 슬롯을 갖는 리스트로 정의
    b. 모든 i에 대해, H[i] = None로 초기화한다. 즉 H[i]는 한방향 연결리스트의 head노드를 가르킨다.
        i. 여기서 노드에는 key 값과 value 값이 저장된다.
    c. set(key, value): H[f(key)] 리스트를 탐색하여 key 값을 갖는 노드가 있다면 value 값을 update하고, 없다면 pushFront한다
    d. remove(key): H[f(key)] 리스트를 탐색하여 key 값 노드를 deleteNode를 호출하여 삭제한다.
    e. search(key): H[f(key)] 리스트를 탐색하여 key 값이 있다면 value 값을 리턴하고, 없다면 None를 리턴한다.
3. 아래 그림에선 해시 테이블 H의 크기 SIZE = 5인 경우, f(key) = key % SIZE로 정의했을 때, set(9, value)를 한 경우의 변화이다.





```
class HashChain:
  def __init__(self, m):
    self.size = m   # 슬롯의 갯수 m
    self.H = [None] * self.size

  def hash_function(self, key):
    return f(key) # return hash value for key

  def find_slot(self, key):
    # chaining이므로 빈 슬롯을 찾을 필요없이 해시함수값을 리턴
    return self.hash_function(key)

  def set(self, key, value):
    i = self.find_slot(key)
    v = self.H[i].search(key)
    if v == None: # key 값을 갖는 노드가 없다면 삽입연산
      self.H[i].pushFront(key, value) # (key, value) 노드를 head 노드 위치에 삽입!
    else: # 기존의 key값을 갖는 노드가 있으므로 value값 수정
      v.value = value

  def remove(self, key):
    i = self.find_slot(key)
    v = self.H[i].search(key)
    if v == None return NOTFOUND
    else:
      self.H[i].deleteNode(v)
```