7. 정렬과 탐색

Sorting(정렬)
정렬은 데이터를 순서대로 재배열하는 작업-> 이를 위해선 사물들을 서로 비교할 줄 알아야 함.
asc-> 오름차순, desc-> 내림차순

-> 정렬은 자료의 탐색을 할 때도 매우 중요함-> 만약, 정렬되어 있지 않으면 탐색의 효율이 크게 저하

정렬시킬 대상을 레코드하고 하는데 레코드는 여러개의 필드로 이뤄진다
이때 정렬을 시킬려면 정렬의 시준이 되는 필드인 key가 필요하고, 이렇게 key를 선정하면 이를 기준으로 레코드들을 재배열함

Selection sort(선택 정렬)
선택 정렬-> 가장 작은 요소를 꺼내서 순서대로 출력하는 것
이를 위해, 최소값이 선택되면 이를 맨 앞의 요소와 교환하면 가장 작은 것부터 순서대로 오름차순으로 정렬된다

1. 만약 n개의 요소가 있다면, 이중 첫번째 요소를 제외하고 n-1개의 요소중에 최소값을 찾아서 첫번째 값과 바꿔준다
2. 1번 작업이 끝나면, 첫번째 요소는 최소값이 있고 두번째 요소부터는 정렬이 안된 것이 n-1개 있다.
   따라서 이때도 1번 작업과 동일하게 전체 리스트 중 2번째 요소를 제외하고 n-2개의 요소 중 최솟값을 찾아서 두번째 값과 바꿔준다
3. 해당 작업을 전체 요소의 개수가 n개이면 총 n-1번 반복한다

```
def printstep(arr,val):
    print(" Step %2d= " %val, end='')
    print(arr)

def selection_sort(A):
    n=len(A) #정렬할 리스트의 길이가 n
    for i in range(n-1): #정렬전 리스트의 첫번째 요소를 i로 지정하는 것은 총 n-1번(0,1,2,...,n-2 총 n-1개, 마지막 요소는 지정할 필요가 없으니 n-1번)
        least=i
        for j in range(i+1,n): #만약 i=0이면, 1번째 요소부터, n-1번째 요소까지 i=1이면 2번째 요소부터 n-1번째 요소까지...
            if(A[j]<A[least]):#우리가 설정한 least보다 더 작은 애가 있다면
                least=j #걔로 least를 바꾼다
        A[i], A[least]=A[least],A[i] #그 뒤에 맨 앞 요소와 least 요소를 바꾼다
        printstep(A,i+1)

data=[5,8,3]
print("Original Data: ",data)
selection_sort(data)
print("Selection: ",data)
```

```
Original Data:  [5, 8, 3]
 Step  1= [3, 8, 5]
 Step  2= [3, 5, 8]
Selection:  [3, 5, 8]
```

위 예시를 통해 코드를 한번 더 이해해보자

위 예시에서 n=3이고 i는 0부터 2까지 돈다

i가 0일때

→ least=0이 되고 j는 i+1부터니까 1부터 2까지 돌면서 만약 A[1]>A[0]이면 둘의 위치를 바꾼다

여기선 A[0]이 5고 A[1]이 8이기 때문에 위치를 바꿔주지 않는다.

하지만 j=2가 되면 A[2]=3<A[0]=5이므로 if문의 조건이 성립하기 때문에 둘의 위치를 바꿔준다

해당 결과가 Step 1이 된다: [3,8,5]

그후 i=1일때

→ least=1이 되고 j는 n=3보다 작아야 하기때문에 2가 된다

A[j]=5< A[least]=8 으로 if조건에 성립되기 때문에 둘의 위치를 다시 바꿔준다

해당 결과가 Step2가 된다: [3,5,8]

하지만 시간복잡도가 생각보다 크기 때문에 효율적인 알고리즘도 아니고, 안정성도 만족하지 못한다
하지만, 알고리즘이 비교적 단순하고, 입력 자료의 구성과 상관없이 자료 이동 횟수가 결정된다는 장점이 있다

Insertion sort(삽입 정렬)

삽입정렬은 카드를 정렬하는 것과 유사→ 만약 내 손에 정렬된 카드가 있고, 카드를 추가로 한 장씩 더 받을때마다 그 카드를 순서대로 끼워넣는다면 이 논리구조가 바로 삽입정렬

정렬에서도 이와 유사하게. 정렬이 안된 부분의 숫자를 순서대로 하나씩 정렬된 부분의 적절한 위치를 찾아서 끼워 넣는 과정→ but 배열에서 순서대로 끼워넣을려면 끼워넣을 자리 뒤 요소들을 모두 한 칸씩 뒤로 밀어야 한다는 문제가 있음

```
def printstep(arr,val):
    print(" Step %2d= " %val, end='')
    print(arr)

def insertion_sort(A):
    n=len(A)
    for i in range(1,n): #1부터 n-1까지
        key=A[i]
        j=i-1
        while j>=0 and A[j]>key:
            A[j+1]=A[j] #한칸씩 뒤로 밀기
            j -=1
        A[j+1]=key
        printstep(A,i)

data=[5,3,8,4]
print("Original Data: ",data)
insertion_sort(data)
print("insertion: ",data)
```

```
Original Data:  [5, 3, 8, 4]
 Step  1= [3, 5, 8, 4]
 Step  2= [3, 5, 8, 4]
 Step  3= [3, 4, 5, 8]
insertion:  [3, 4, 5, 8]
```

해당문제 역시 왜 이런 결과가 도출되었는지 해석해보자

해당 예시에선 n=4이고 따라서 i는 1부터 4까지이다

i=1일때

i=1이면 j=0이 된다→ key=A[1]=3이고 A[j=0]=5가 된다. A[j]>key를 만족하기 때문에

j들을 모두 한칸씩 뒤로 밀어주고 j를 -1한다→ 현재 j=-1

이후 A[j+1=0]=5를 A[1]인 key에 대입한다.

그 결과 원래 A[0]=5, A[1]=3인 것이 A[0]=3, A[1]=5로 변한다

Step 1: 3, 5, 8, 4

i=2일때

i=2이면 j=1이 되고 key=A[2]=8, A[j=1]=5가 된다. A[j]>key조건을 만족하지 않기 때문에 그대로 종료된다

Steo 2: 3,5,8,4

i=3일때

i=3이면 j=2가 되고 key=A[3]=4, A[j=2]=8이 된다. A[j]>key조건을 만족하기 때문에 한칸씩 뒤로 밀어주고 j -=1을 진행한다→ 그럼 j=1이 됨

이후 A[j+1=2]=8,

삽입 정렬은 많은 이동이 필요하므로, 양이 많은 경우엔 비효율적이다
하지만, 알고리즘이 간단하고 안정성도 충족하기 때문에 레코드가 대부분 이미 정렬되어 있는 경우엔 효율적으로 사용할 수 있다

Bubble sort 버블정렬

버블정렬은 인접한 2개의 레코드를 비교하여 크기가 순서대로가 아니면 서로 교환하는 방법을 사용-> 이때 인접한 2개 수 중에 큰 수가 오른쪽에 오도록 함
리스트를 한번 스캔하면 가장 큰 수가 가장 오른쪽 끝으로 이동한다.
해당 작업을 더 이상 교환이 일어나지 않을 때까지 반복해서 수행한다

```
def printstep(arr,val):
    print(" Step %2d= " %val, end='')
    print(arr)


def bubble_sort(A):
    n=len(A)
    for i in range(n-1,0,-1): #외부루프: n-1,n-2,...,2,1버블 정렬은 오른쪽에 최대값이 오므로 오른쪽부터 왼쪽 순
        bChanged= False
        for j in range(i):# 내부루프: 0,1,2,...,i-1
            if (A[j]>A[j+1]):#큰수가 오른쪽이 아니라 왼쪽에 있다면
                A[j],A[j+1]=A[j+1],A[j]#둘이 위치를 바꿔라
                bChanged=True #교환이 발생했음

        if not bChanged:break #교환이 발생하지 않으면 종료
        printstep(A,n-i)

data=[5,3,8,4,9,1,2]
print("Original Data: ",data)
bubble_sort(data)
print("bubble: ",data)
```

```
Original Data:  [5, 3, 8, 4, 9, 1, 2]
 Step  1= [3, 5, 4, 8, 1, 2, 9]
 Step  2= [3, 4, 5, 1, 2, 8, 9]
 Step  3= [3, 4, 1, 2, 5, 8, 9]
 Step  4= [3, 1, 2, 4, 5, 8, 9]
 Step  5= [1, 2, 3, 4, 5, 8, 9]
bubble:  [1, 2, 3, 4, 5, 8, 9]
```

정렬을 이용한 집합-시험
→ 정렬이 안된 집합 연산과 비교하기

Chapter3에서 만든 집합은 집합의 원소를 정렬없이 마음대로 설정했지만, 정렬 알고리즘을 배운 만큼 이번엔 삽입할 때 정렬된 형태가 되도록 삽입을 해보자
-> 정렬 개념이 들어갔으니 연산이 당연히 더 복잡해진다

1. Insert 연산
    a. 가장 먼저 해야 할 것은 삽입할 값을 집합이 이미 가지고 있는지 체크하는 것이다-집합은 중복을 포함하지 않기 때문이다
    b. 이젠 삽입을 해야 하는데, 과거 집합 구현처럼 정렬을 무시한 채로 삽입하는 것이 아니라 삽입을 한 후에도 정렬이 되어야 한다.
    따라서 우선적으로 삽입할 원소를 집합의 맨 뒤에 넣는다
    c. 그 뒤엔 맨 뒤에서부터 한칸씩 앞으로 오면서, 만약 앞 원소가 뒤 원소보다 크다면 둘의 위치를 바꿔주도록 한다

```
def insert(self,e):
    if self.contains(e) or self.isfull():
        return

    self.array[self.size]=e #맨뒤에 e라는 값을 대입
    self. size +=1

    for i in range(self.size-1,0,-1):
        if self.array[i-1]>=self.array[i]: #앞에 원소가 뒤에 원소가 더 크면 둘의 위치를 바꾼다
            break
        self.array[i-1],self.array[i]=self.array[i],self.array[i-1]
```

2. 합집합연산
    a. 집합의 원소가 크기순으로 되어있기 때문에 가장 작은 원소부터 공통으로 있는지 비교해서 없으면 더 작은 원소를 새로운 집합에 넣고 인덱스를 증가시킨다
    예를 들어 self와 setB중에 self가 더 작다면 self를 합집합에 추가하고 self의 인덱스만 증가시킨다
    b. 공통된 것이 있다면 이 역시 합집합에 추가하고 이땐 self, setB 인덱스 모두를 증가시킨다

```
def union(self,setB):
    setC=SortedArraySet()
    i=0 #self의 인덱스
    j=0 #setB의 인덱스
    while i<self.size and j<setB.size: #어느 한쪽 배열이 끝날 때 까지 반복
        a=self.array[i]
        b=setB.array[j]
        if a==b:#두 집합의 원소가 같으면 아무거나 C에 넣고 인덱스 둘 다 1개씩 올리기
            setC.append(a)
            i +=1
            j +=1
        elif a<b:#작은애가 있으면 걔를 C에 넣고 넣은 애의 인덱스 1 올리기
            setC.append(a)
            i +=1
        else:
            setC.append(b)
            j +=1
    while i<self.size:#self의 원소가 아직 남았으면, 남은 모든 원소를 C에 넣기
        setC.append(self.array[i])
        i +=1

    while j<setB.size:#setB의 원소가 아직 남았으면, 남은 모든 원소를 C에 넣기
        setC.append(setB.array[j])
        j +=1

    return setC
```

탐색-> 테이블에서 원하는 탐색키를 가진 레코드를 찾는 작업

용어정리

1. 테이블=레코드의 집합.
2. 탐색키- 레코드들을 서로 구별하여 인식할 수 있는 키
3. 맵,딕셔너리, 사전: 탐색을 위한 자료구조, (키, 값)=엔트리의 집합

순차탐색

1. 정렬되지 않은 테이블에서 원하는 레코드를 찾을 수 있는 가장 단순하고 직관적인 방법
2. 맨 처음부터 원하는 값이 나올때까지 계속 비교하는 것-> 그러다보니 당연히 제일 좋은거는 찾는 값이 맨 앞에 있는 경우이고, 최악은 맨 뒤 혹은 아예 리스트에 값이 없는 경우가 최악

```
def sequential_search(A,key,low,high):
    for i in range(low, high+1):
        if A[i]==key:
            return i
    return -1
```

이진탐색-> 순차탐색을 보완한 것

1. 이진탐색은 순차탐색보다 훨씬 더 효율적으로 탐색이 가능하지만 "정렬"이 되어 있어야 한다는 전제가 반드시 필요하다!
2. 전제가 되어 있다는 가정하에 우선 중앙에 리스트의 정중앙에 있는 값을 조사한다
3. 만약 중앙에 있는 값을 기준으로 찾는 값이 왼쪽에 있다면 오른쪽을 볼 필요가 없어진다
4. 왼쪽 값들을 기준으로 걔네들의 정중앙 값을 또 찾아서 왼쪽 오른쪽을 비교한다
5. 2,3,4과정을 반복해 원하는 값을 찾는다

```
#이진탐색 순환구조
def binary_search(A,key,low,high):
    if (low>high): #우리가 원하는 값이 없으면 low와 high가 역전된다
        return -1

    middle=(low+high)//2

    if(key==A[middle]):
        return middle
    elif (key<A[middle]): #우리가 찾는 값이 왼쪽에 있다는 말
        return binary_search(A,key,low,middle-1)
    else: #우리가 찾는 값이 중앙값 기준 오른쪽에 있다는 말
        return binary_search(A,key,middle+1,high)
```

```
#이진탐색 반복구조
def binary_search_iter(A,key,low,high):
    while(low<=high):
        middle=(low+high)//2

    if key==A[middle]:
        return middle
    elif key>A[middle]:
        low=middle+1
    else: #key<A[middle]
        high=middle-1
    return -1
```

보간탐색

보간탐색은 이진탐색과 유사→ 우리가 사전에서 단어를 찾을때를 생각해보면, 우리는 단어에서 알파벳순서대로 앞쪽인지 뒤쪽인지

를 대강 파악한다→ 이와 비슷하게 탐색에서도 탐색키가 존재할 위치를 예측해 탐색하는 방법이 보간탐색

그렇다면, 어떻게 탐색위치를 예측할까?

이진 탐색에서는 항상 middle=(low+high)/2였지만 보간탐색에서는 다음과 같이 탐색위치를 결정한다

$$\text{탐색위치} = \text{low} + (\text{high} - \text{low}) \times \left(\frac{k - A[\text{low}]}{A[\text{high}] - A[\text{low}]} \right)$$

이때 k= 찾고자 하는 킷값, low, high는 각각 탐색범위의 최소, 최대 인덱스를 나타낸다

```
#보간탐색
def binary_search_iter(A,key,low,high):
    while(low<=high):
        middle=int(low+(high-low)*(key-A[low].key)/(A[high].key-A[low].key))

    if key==A[middle]:
        return middle
    elif key>A[middle]:
        low=middle+1
    else: #key<A[middle]
        high=middle-1
    return -1
```

해당 코드에서 주의할 것은 위치의 비율을 계산하기 위해서 실수 계산이 적용되지만 인덱스는 정수이기 때문에 마지막에 int를 추가해줘야 한다는 것이다.

고급 탐색구조: 해싱

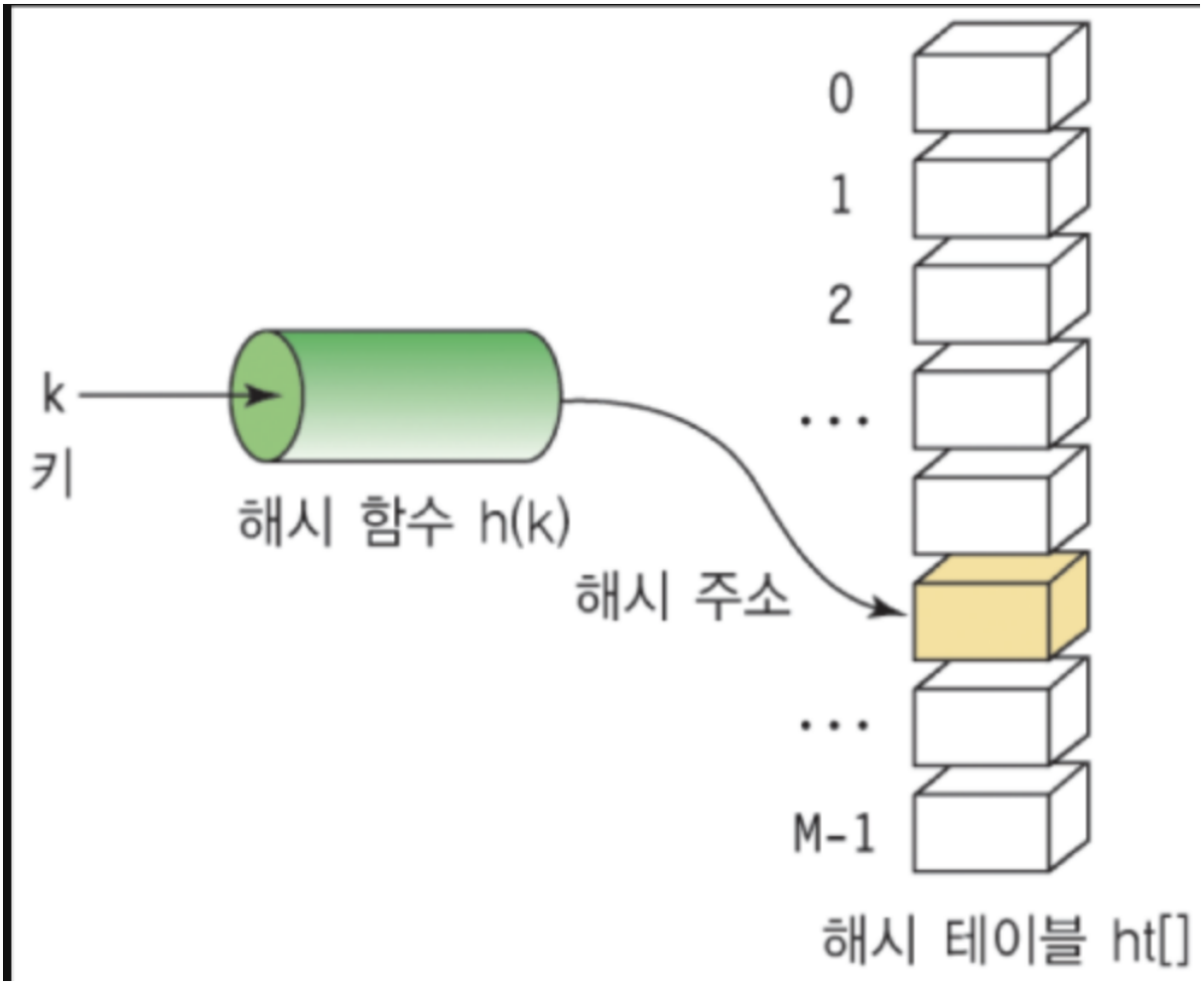
해싱은 쉽게말해 세대별 우편함과 동일→ 만약 한 아파트의 모든 우편물이 한 곳에 집중되어 있다면(like 우리학교 기숙사) 일일히 하나하나 다 뒤져봐야 한다→ 하지만, 세대별 우편함이 있기 때문에 쉽게 찾을 수 있다→ 물론 이를 위해선 세대별로 우편함을 만들어야 하기 때문에 공간과 비용이 필요함

기존의 탐색: 우리가 찾을려는 킷값과 탐색키(low, high, middle같은)들을 비교해서 원하는 항목을 찾았다

반면 해싱은 비교가 아니라, 킷값에 산술적인 연산을 적용하여 레코드가 저장되어야 할 위치를 직접 계산하는 것→ 이를 위해, 탐색키로부터 레코드가 저장될 위치를 함수를 이용해서 계산하고(using 해시함수) 그 위치에 레코드가 있는지 확인(우리집 우편함에 택배가 왔는지 확인하는 것과 동일)

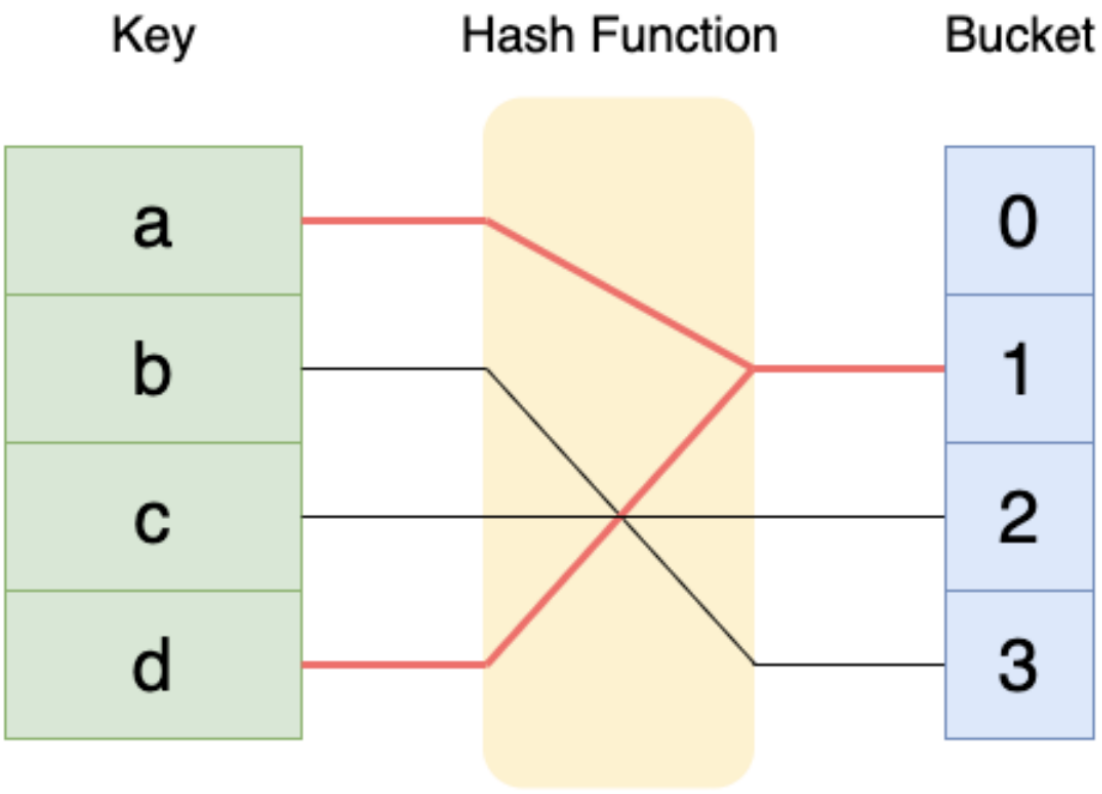
이때, 해시 함수에 의해 계산된 위치에 레코드를 저장할 테이블 is called 해시 테이블

충돌과 overflow



킷값(우리가 찾고 싶은 값)을 해시 함수에 대입→ 이때 만들어진 함숫값(h(키)=해시주소)을 인덱스로 사용해 해시 테이블에 접근→ 이때 만약 서로 다른 키가 해시 함수에 의해 서로 같은 해시 주소로 계산되면 어떻게 될까?

→ 이를 해싱에서는 collision(충돌)이라고 하고 충돌을 일으키는 키들을 동의어라고 한다



위 그림을 통해 이해해보자.

위 그림에서 key값이 a,d인 두 key는 분명 서로 다르지만 해시함수를 통해 1이라는 하나의 해시주소를 가지게 된다. 이러한 상황을 충돌이라 하고 이때 a,d는 동의어가 된다.

그렇다면 이런 충돌을 어떻게 해결할 수 있을까?

1. 만약 버킷에 여러개의 슬롯이 있다면, 1번이라는 버킷에 서로 다른 두 슬롯에 a,d를 저장하면 된다
2. 하지만 때에 따라선, 충돌이 슬롯수보다 더 많이 발생할 수도 있음(슬롯에 여유가 없을때)
   → 이처럼 충돌이 슬롯수보다 많이 발생하는 경우를 오버플로라고 한다
   → 오버플로가 발생하면 해당 버킷에 더 이상 항목을 저장할 수 없기 때문에 이를 반드시 해결해야 한다

좋은, 이상적인 해싱은 충돌이 절대 일어나지 않는 경우임
→ 하지만, 충돌이 아예 일어나지 않을려면 해시 테이블의 크기를 아주 넉넉히 크게 하면서 메모리도 지나치게 많이 필요
→ 따라서 실제 해싱은 테이블의 크기를 적절히 줄이고, 해시 함수를 이용해 주소를 계산 + 실제 해싱에서는 충돌과 오버플로가 빈번하게 발생

해시함수

→ 충돌이 일어나는 이유는 결국 해시 함수때문임→ 따라서 좋은 해시함수는 충돌이 적고, 주소가 테이블에서 고르게 분포되어야 하며, 계산이 빨라야 함

제산 함수 (가장 general하게 쓰임)
→ 나머지 연산을 이용한 함수
→ 해시 테이블의 크기가 M일때, 탐색키 K에 대해 해시 함수는 다음과 같다

$$h(k) = k * mod M$$

이때 가능하면, 테이블의 크기 M은 소수로 선택하자
→ 소수면 K%M이 0에서 M-1 사이에 골고루 분포→ 충돌이 일어날 확률 낮아짐

해싱의 오버플로 해결방법→ 해싱에서는 오버플로를 반드시 처리해야 함

1. 개방 주소법
   → 오버플로가 일어나는 항목을 해시 테이블의 다른 위치(주소)에 저장
   → ex. 선형 조사법, 이차 조사법, 이중 해싱법
2. 체이닝
   → 해시 테이블의 하나의 위치에 여러개의 항목을 저장할 수 있도록 테이블의 구조를 변경

```
#해시테이블 및 해시 함수 준비
M=13
table=[None]*M
def hashfn(key):
    return key % M
```

선형조사법

→ 개방주소 법의 대표적인 방법이 바로 선형조사법으로, 해시 함수로 계산된 버킷에 빈 슬롯이 없으면, 그다음 버킷에서 빈 슬롯이 있는지를 찾는 방법
→ 이때 빈 곳을 찾는 것을 조사(probing)라고 함
→ex. 해시테이블의 k th위치인 table[k]에서 충돌이 발생하면, 그 다음 위치인 table[k+1]부터 순서대로 비어있는지 조사하고, 빈 곳이 있으면 값을 저장
→ 빈 곳이 나올때까지 계속 조사를 반복하고 만약 테이블의 끝에 도달하면 다시 처음부터 k-1th까지 반복→ 이러다가 빈 곳을 발견하지 못하면(=조사과정에서 처음 충돌이 발생한 곳으로 다시 돌아왔다면) 테이블이 가득 찬 상태

1. 선형조사법의 삽입연산

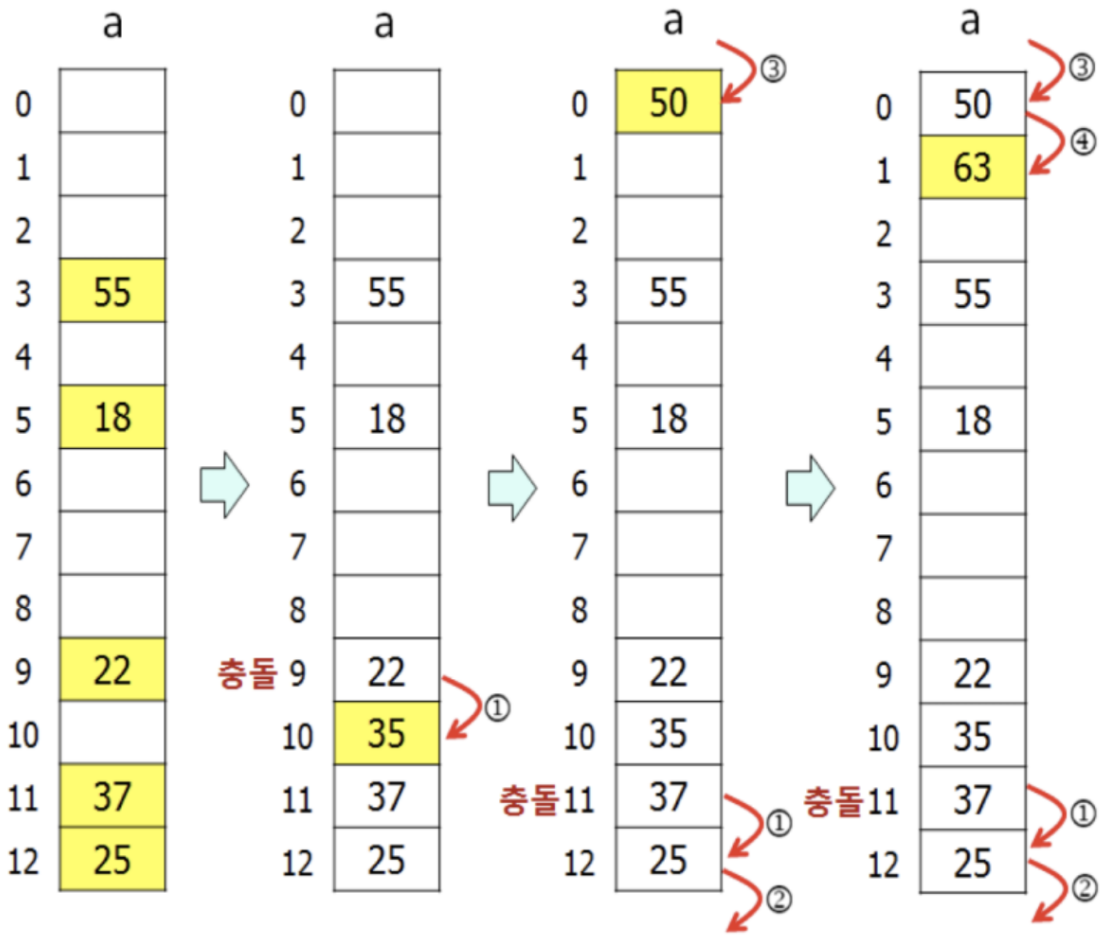
key	h(key) = key % 13
25	12
37	11
18	5
55	3
22	9
35	9
50	11
63	11

출처: https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=beaqon&logNo=221300352778

해당 자료를 예시로 삽입연산에 대해 생각해보자. 해시함수는 key%13이고 각 key값들은 해시함수에 따라 연산되고 해시 주소는 오른쪽과 같다. 이때 삽입을 진행해보면 25, 37, 18, 3, 9까지는 해당 주소가 비어있기 때문에 아무런 문제가 발생하지 않는다

문제는 35부터인데 35의 경우 해시주소가 22일때와 동일하기 때문에 충돌이 발생→ 원래는 35 역시 9번 주소로 가야 하지만 이미 9번 주소는 22가 들어갔기 때문에 그 다음 주소인 10번 주소에 35가 저장

50과 63 역시 35때와 동일하게 처리→ 해시주소가 둘 다 11인데 이는 37과 같고, 이미 37이 11번 주소에 들어갔기 때문에 12번으로 가야하지만 12번도 들어서 있다→ 따라서 빈 칸을 찾아 계속 탐색과정을 이어가고 이후 비어있던 0 번주소에 50, 1번 주소에 63이 저장



출처:https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=beaqon&logNo=221300352778

```
#1. 선형조사법의 삽입 연산
def insert(key):
  i=hashfn(key)
  count=M
  while count>0:
    if table[i]==None or table[i]== -1: #i번째 인덱스 값이 비어있거나, 삭제된 것이라면 멈춘다
      break
    i=(i+1)%M #
    count -=1

  if count>0:
    table[i]=key
```

2. 선형조사법의 탐색 연산

탐색은 삽입과 유사

→ 탐색키가 입력되면 이를 해시함수에 대입해 해시주소를 계산

→ 해당 주소에 같은 키의 레코드가 있으면 탐색 성공

→ 만약 탐색을 계속 진행하다가 레코드가 없는 버킷을 만나거나, 모든 버킷을 다 검사해도 없으면 실패

```
def search(key) :
    i = hashFn(key)
    count = M
    while count>0 :
        if table[i] == None :   # 탐색 실패
            return None

        if table[i] == key :    # 탐색 성공
            return table[i]
```

```
        i = (i + 1) % M
        count -= 1

    return None                # 탐색 실패
```

3. 선형조사법의 삭제연산
   삭제연산은 조금 생각해야 할 것이 있음
   탐색을 진행할 시에 항목이 삭제되면, 탐색이 불가능해짐
   → 따라서 이제는 비어있는 버킷을 2개(한번도 사용하지 않은 새삥, 한번 사용했다가 삭제되어 현재는 비어있는 중고)로 구분→
   탐색과정은 새삥 버킷을 만나야만 중단이 되도록 코드를 수정해야함

```
def search(key) :
    i = hashFn(key)
    count = M
    while count>0 :
        if table[i] == None :   # 탐색 실패
            return
        if table[i] == key :    # 탐색 성공
            return table[i]

        i = (i + 1) % M
        count -= 1 #count자리에 None이 아니라 -1을 넣는 것 유의

    return None                # 탐색 실패
```

```
#코드 7.15: 선형조사법의 테스트 프로그램
if __name__ == "__main__":
    data = [45, 27, 88, 9, 71, 60, 46, 38, 24]
    for d in data :
        print( "h(%2d)=%2d"%(d,hashfn(d)), end=' ')
        insert(d)
        print(table)

    print("46탐색-->", search(46))
    print("39탐색-->", search(39))

    print("60삭제-->", end='')
    delete(60)
    print(table)
    print("46삭제-->", end='')
    delete(46)
    print(table)
```

```
h(45)= 6 [None, None, None, None, None, None, 45, None, None, None, None, None, None]
h(27)= 1 [None, 27, None, None, None, None, 45, None, None, None, None, None, None]
h(88)=10 [None, 27, None, None, None, None, 45, None, None, None, 88, None, None]
h( 9)= 9 [None, 27, None, None, None, None, 45, None, None, 9, 88, None, None]
h(71)= 6 [None, 27, None, None, None, None, 45, 71, None, 9, 88, None, None]
h(60)= 8 [None, 27, None, None, None, None, 45, 71, 60, 9, 88, None, None]
h(46)= 7 [None, 27, None, None, None, None, 45, 71, 60, 9, 88, 46, None]
h(38)=12 [None, 27, None, None, None, None, 45, 71, 60, 9, 88, 46, 38]
h(24)=11 [24, 27, None, None, None, None, 45, 71, 60, 9, 88, 46, 38]
46탐색--> 46
39탐색--> None
60삭제-->[24, 27, None, None, None, None, 45, 71, -1, 9, 88, 46, 38]
46삭제-->[24, 27, None, None, None, None, 45, 71, -1, 9, 88, -1, 38]
```

결과에서 유의할 것은 삭제할 때 None대신 -1이 들어간다는 것

이번에는 체이닝을 통해 오버플로를 해결해보자

체이닝→ 하나의 버킷에 여러 개의 레코드를 저장할 수 있도록 하는 방법

여러개의 레코드를 저장하고자 연결 리스트를 사용

체이닝을 이용해 크기가 7인 해시 테이블에 h(k)= K%7의 해시함수를 이용해 8,1,9,6,13을 삽입하는 과정을 보자

위 그림에서 볼 수 있듯이, 만약 충돌이 일어나지 않는다면 시작 노드에 바로 저장하면 되고, 충돌이 발생한다면 시작 노드의 링크과 연결

```
M = 13                  # 해시 테이블의 크기
table = [None]*M        # 해시 테이블. 모든 항목은 None으로 초기화
def hashFn(key) :     # 해시 함수로는 제산함수 사용
    return key% M

class Node:
    def __init__( self, data, link=None ):
        self.data = data
        self.link = link

# 코드 7.16: 체이닝을 이용한 오버플로 처리
def insert(key) :
    k = hashFn(key)     # 해시 주소 계산
    n = Node(key)       # 새로운 노드 생성
    n.link = table[k]   # 노드의 다음 노드로 시작 노드 연결
    table[k] = n        # 새로운 노드가 시작 노드가 됨

def search(key) :
    k = hashFn(key)
    n = table[k]                # 시작 노드
    while n is not None:        # 연결된 모든 노드를 탐색
        if n.data == key :      # 탐색 성공. 노드의 데이터(레코드) 반환
            return n.data
        n = n.link              # 링크를 따라 다음 노드로 진행
    return None                 # 탐색 실패. None 반환

def delete(key) :
    k = hashFn(key)
    n = table[k]                # 시작 노드
    before = None               # 이전 노드
    while n is not None:        # n이 None이 아닐때 까지
        if n.data == key :      # 삭제할 레코드 찾음
            if before == None : # 삭제할 항목이 시작 노드이면, 다음 노드가 시작노드
                table[k] = n.link
            else :              # 두 번째 이후 항목 삭제인 경우
                before.link = n.link
            return
        before = n
        n = n.link

#=========================================================================
def printTable() :
    for i in range(M):
        n = table[i]
        if n != None :
            print("[%2d] "% i, end='')
            while n is not None :
                print( n.data, end=' ')
                n = n.link
            print()

if __name__ == "__main__":
    data = [45, 27, 88, 9, 71, 60, 46, 38, 24]
    for d in data :
        insert(d)
    printTable()

    print("46탐색-->", search(46))
    print("39탐색-->", search(39))

    print("60삭제-->")
    delete(60)
```

```
    print("46삭제-->")
    delete(46)
    printTable()
```