알고리즘분석 실습 자료

14주차

Photo by Piotr Guzik on Unsplash

실습 소요 시간 100분

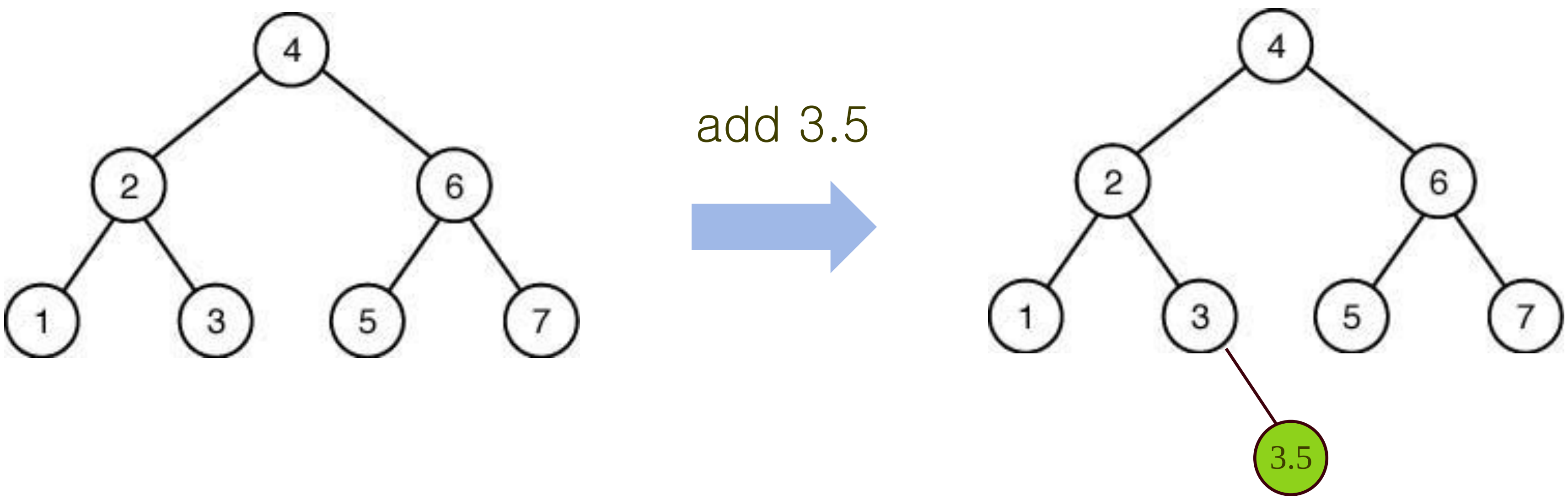
8장 계산복잡도: 검색 문제(1)

실습프로그램

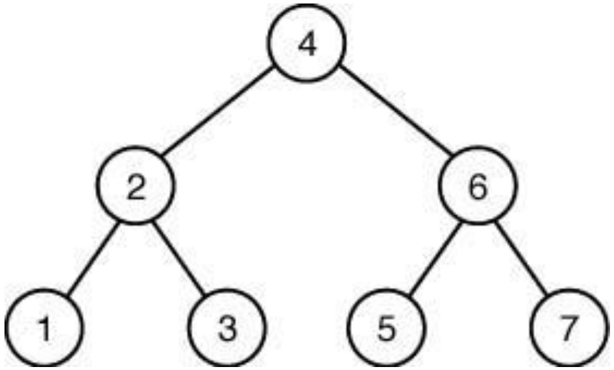
- ✓ 이진검색트리
- ✓ 보간검색
- ✓ closed hashing

Reason to Use Binary Search Tree

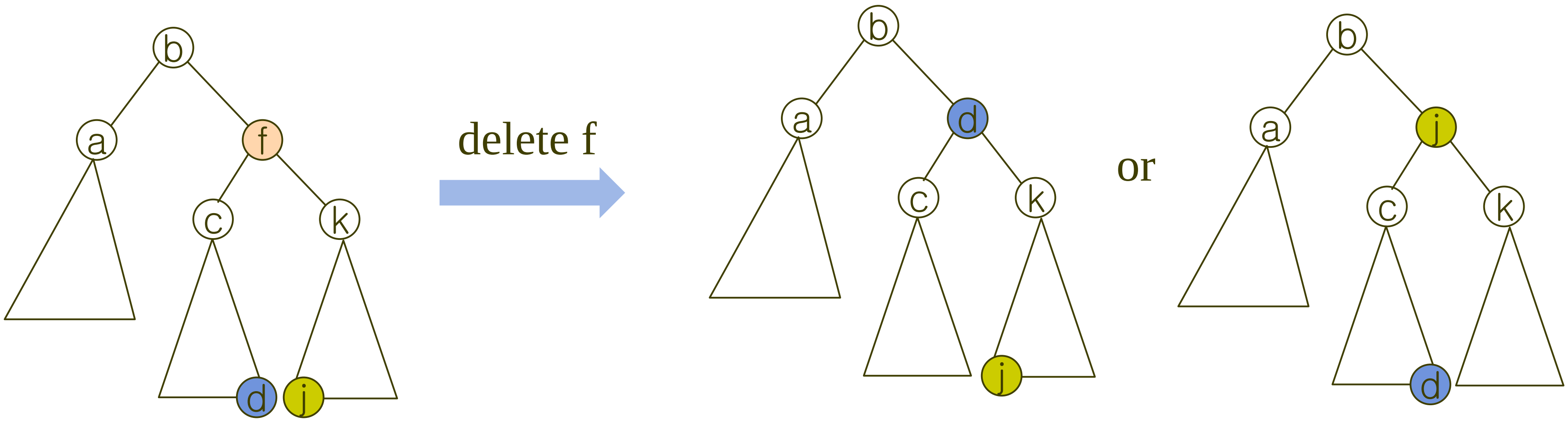
- We can add keys to and delete keys efficiently from the tree.
  - ✓ addition: trivial
  - ✓ deletion: use a simple operation



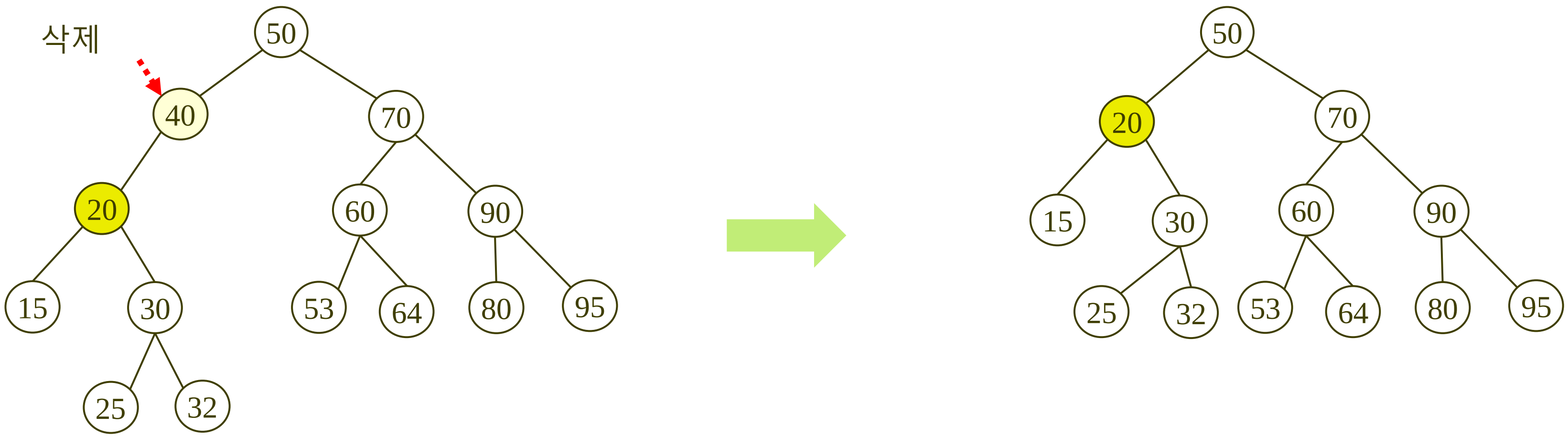
- 두 개의 자식노드가 있는 노드가 삭제될 경우



- ✓ delete 6.



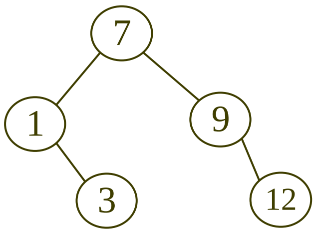
- 한 개의 자식노드가 있는 노드가 삭제될 경우



[실습프로그램] 이진검색트리 구축

```python
import utility
class Node:
    def __init__(self,data):
        self.l_child=None
        self.r_child=None
        self.data = data
def bin_search_tree_insert(root,node):
    if root is None:
        root=node
    else:


              구현









r = Node(7)
bin_search_tree_insert(r,Node(9))
bin_search_tree_insert(r,Node(1))
bin_search_tree_insert(r,Node(3))
bin_search_tree_insert(r,Node(12))

utility.print_inOrder(r)
print()
utility.print_preOrder(r)
```

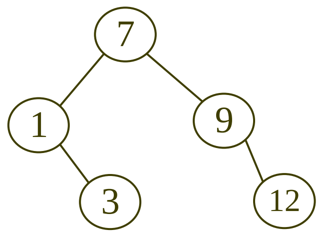
insert 7, 9, 1, 3, 12




```
1
3
7
9
12

7
1
3
9
12
>>>
```

[실습프로그램] 이진검색트리에서 하나의 원소를 삭제하는 프로그램

insert 7, 9, 1, 3, 12




```
def bin_search_tree_delete(root,node):


  프로그래밍 연습
```

(1) 두 개의 자식노드를 가진 노드가 삭제

(2) 하나의 자식노드를 가진 노드가 삭제

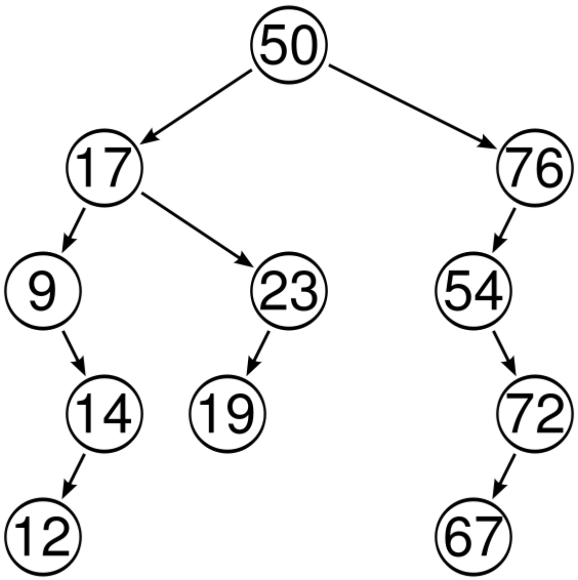
(3) 말단 노드의 삭제

검색시간 향상을 위한 트리구조

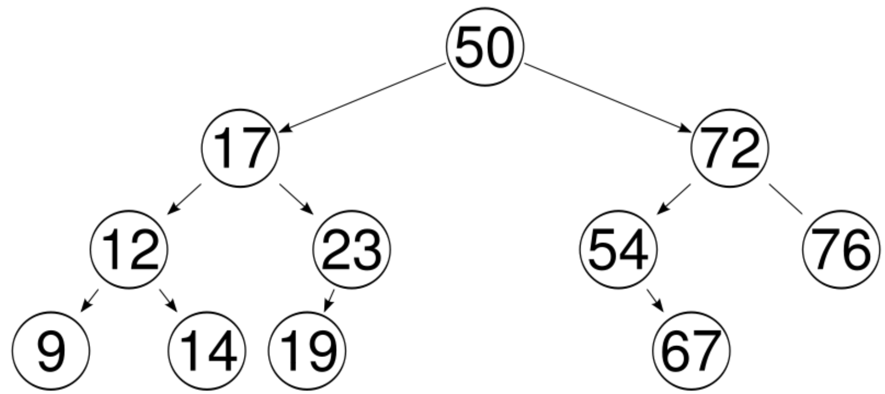
- 항상 균형을 유지하는 이진트리 활용

- **균형트리**
  - ✓ **AVL 트리**: 아이템의 추가, 삭제, 검색: 모두 $\Theta(\lg n)$
  - ✓ B-트리 / 2-3 트리: 잎마디들의 깊이(수준)를 항상 같게 유지. 아이템의 추가, 삭제, 검색: 모두 $\Theta(\lg n)$
  - ✓ red-black tree: 아이템의 추가, 삭제, 검색: 모두 $\Theta(\lg n)$

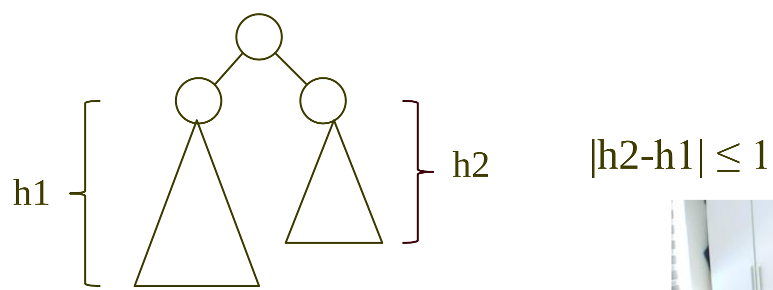
- **AVL 트리**: 좌,우subtree의 높이의 차가 최대 1인 이진검색트리
- 러시아의 두 수학자인 G.M. Adelson-Velsky 와 d E.M. Landi, 1962년



일반적인 이진검색트리

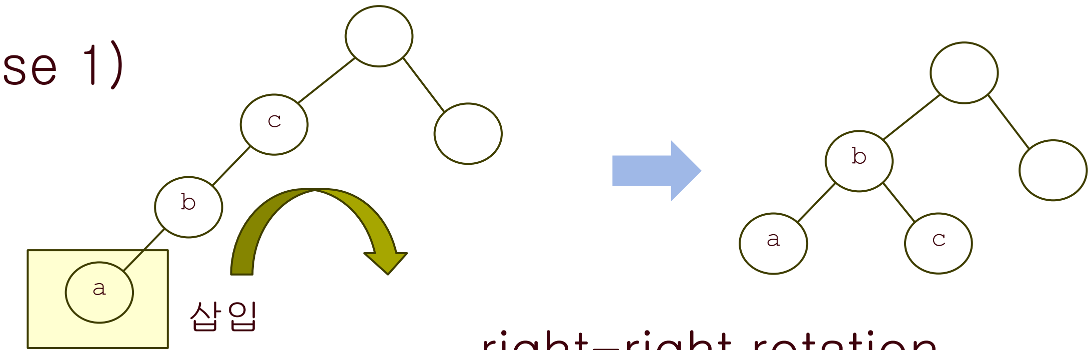


AVL 트리



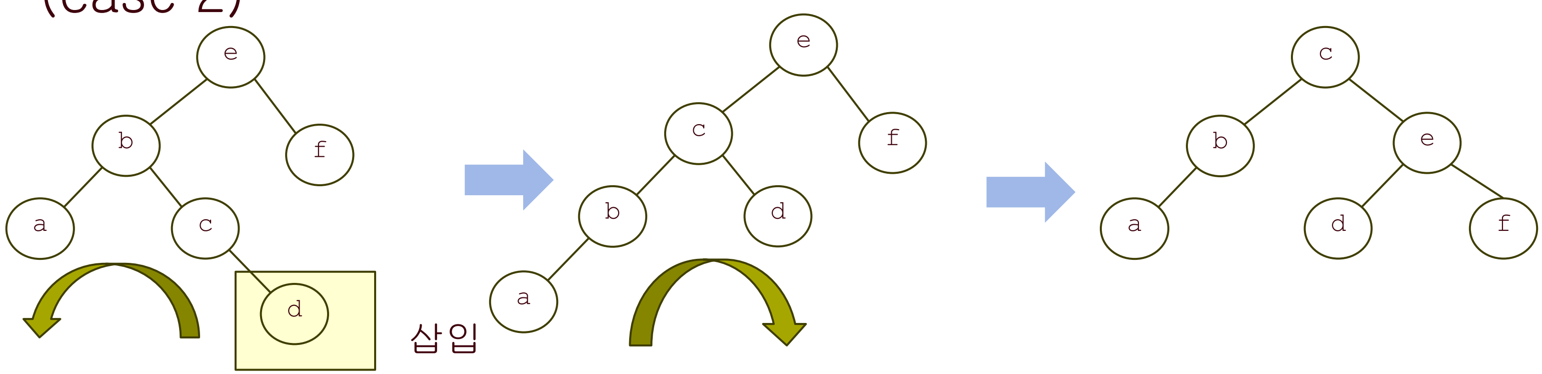
AVL 트리에서 데이터 추가 시 균형을 유지하는 방법

(case 1)



right-right rotation

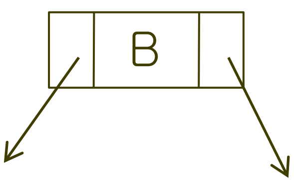
(case 2)

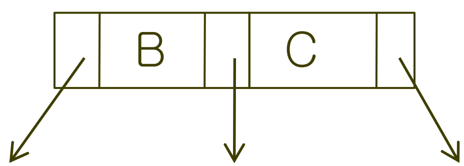


left-right rotation

(case 3) (case 4) : case 1, 2의 대칭 → 총 4 cases

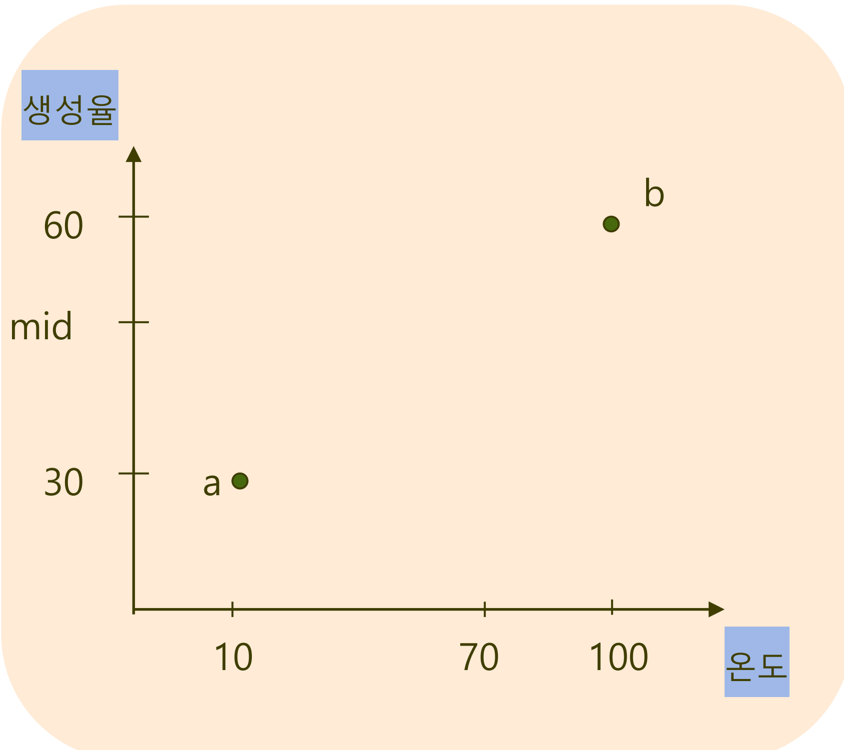
- B-tree 중 가장 간단한 형태인 2-3 트리
- 성질
  - ✓ 각 마디에는 키가 하나 또는 둘 존재
  - ✓ 각 내부마디의 자식 수는 키의 수+1
  - ✓ 어떤 주어진 마디의 왼쪽(오른쪽) 부분트리의 모든 키들은 그 마디에 저장되어 있는 마디보다 작거나(크거나) 같다.
  - ✓ 모든 잎마디는 수준이 같다
  - ✓ 데이터는 트리 내의 모든 노드에 저장

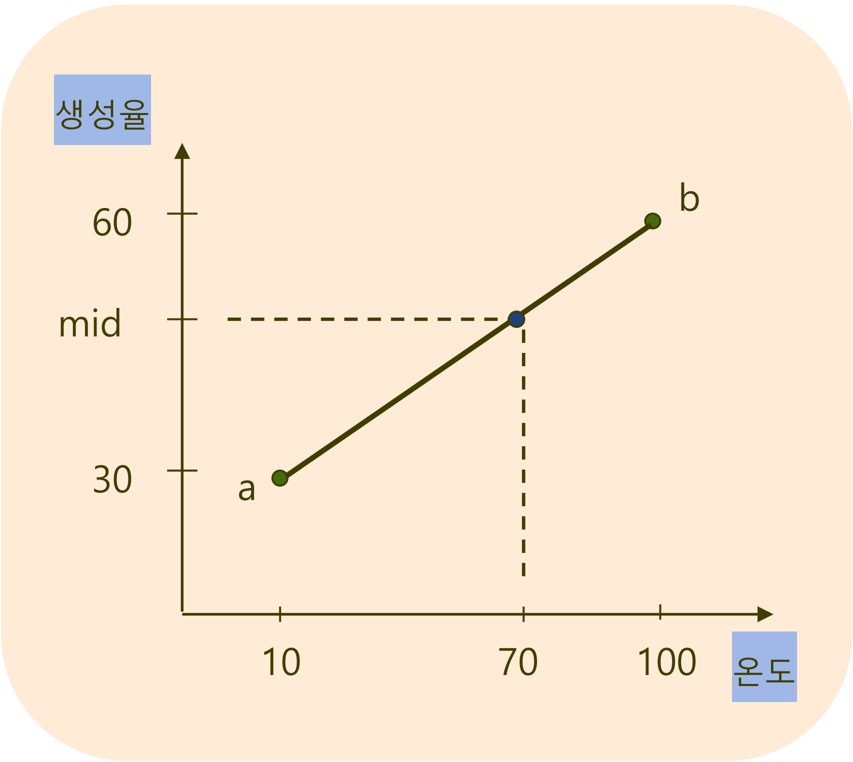




선형 보간법(linear interpolation)

- 두 개의 관찰점 a=(10,30), b=(100,60)을 갖고 특정 점의 값을 추정
- 70도 일 때 생성율은 얼마라고 추정할 수 있나?





$$mid = 30 + \left[\frac{70-10}{100-10} \times (60-30) \right]$$

보간 검색

- 보간검색(interpolation search)
  - ✓ 일반적으로 데이터가 가장 큰 값과 가장 작은 값이 균등하게 분포되어 있다고 가정하여, 키가 있을 만한 곳을 바로 가서 검사해 보는 방법

    → 키의 값 자체를 이용하는 방법
  - ✓ Find x,

$S[low]$		x				$S[high]$
low		mid				$high$

  - ✓ Estimate mid such that $S[mid]=x$.

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$$

선형보간법(Linear Interpolation)

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$$

- **보기**: S[1] = 4이고 S[10] = 97일 때 검색 키가 25이면, *mid* = 3.

$$mid = 1 + \left\lfloor \frac{25-4}{97-4} \times (10-1) \right\rfloor = 1 + \left\lfloor \frac{21}{93} \times 9 \right\rfloor = 3$$

- **분석**:
  - ✓ 아이템이 균등하게 분포되어 있고, 검색 키가 각 슬롯에 있을 확률이 같다고 가정하면, 선형보간검색의 <u>평균적인 시간복잡도</u>는 $A(n) \approx \lg(\lg n)$.
    (예) n=10억, $\lg(\lg n)$은 약 5.
  - ✓ 최악의 경우의 시간복잡도가 나쁨.

    (예) 10개의 아이템이 1, 2, 3, 4, 5, 6, 7, 8, 9, 100 이고, 여기서 10을 찾으려고 한다면, *mid*값은 항상 *low*값이 되어서 모든 아이템과 비교를 해야 한다. 따라서 <u>최악의 경우 시간복잡도</u>는 순차검색과 같다.

$$mid = 1 + \left\lfloor \frac{10-1}{100-1} \times (10-1) \right\rfloor = 1 + \left\lfloor \frac{9}{99} \times 9 \right\rfloor = 1$$

```
void interpsrch(int n, const number S[], number x, index& i){

    index low, high, mid;
    number denominator;

    low =1; high=n; i=0;
    if(S[low] <= x <= S[high])
      while (low <= high && i==0){
           denominator = S[high] - S[low];
           if(denominator == 0)
                  mid = low;
           else
                  mid = low + ⌊((x - S[low])*(high - low))/denominator⌋;
           if (x==S[mid])
                  i = mid;
           else if ( x < S[mid])
                  high = mid -1;
           else
                  low = mid + 1;
           }
}
```

◆ 배열 내에 찾는 데이터가 존재하는 경우의 의사코드

```
void interpsrch(int n, const number S[], number x, index& i){

    index low, high, mid;
    number denominator;

    low =1; high=n; i=0;
    if(S[low] <= x <= S[high])
      while (low <= high && i==0){
           if(x < S[low] or x > S[high])
                break
           denominator = S[high] - S[low];
           if(denominator == 0)
                mid = low;
           else
                mid = low + ⌊((x - S[low])*(high - low))/denominator⌋;
           if (x==S[mid])
                i = mid;
           else if ( x < S[mid])
                high = mid -1;
           else
                low = mid + 1;
           }
}
```

◆ 배열 내에 찾는 데이터가 존재하지 않을 수 있는 경우의 의사코드

[실습프로그램] 보간검색

```
import math
def interpolationSearch(S,x):
   low = 0
   high= len(S)-1
   location= -1


      구현


S = [1,3,4,7,8,11,13,15,16,20,22,25,29,30,33,36,37,39,41,43,45,48]
x = 11
print(S)
print(f’Location of {x:d} is {interpolationSearch(S,x)):d}th’)
```

```
[1, 3, 4, 7, 8, 11, 13, 15, 16, 20, 22, 25, 29, 30, 33, 36, 37, 39, 41, 43, 45, 48]
Location of 11 is 5th
>>>
```

보강된 보간검색법 (robust interpolation search)

- gap 변수 사용

  1. gap의 초기값 $= \lfloor (high - low + 1)^{1/2} \rfloor$

  2. *mid*값을 위와 같이 선형보간법으로 구한다.

  3. 다음 식으로 새로운 *mid*값을 구한다.

  $$mid = \text{MIN}\ (high - gap,\ \text{MAX}(mid,\ low + gap))$$

- **보기(이전 예)**: 10개의 아이템이 1, 2, 3, 4, 5, 6, 7, 8, 9, 100 이고, x=10.

  $gap = \lfloor (10 - 1 + 1)^{1/2} \rfloor = 3$, 초기 *mid*=1,

  $mid = MIN(10 - 3,\ MAX(1,\ 1 + 3)) = 4$

- **분석**: 아이템이 균등하게 분포되어 있고, 검색 키가 각 슬롯에 있을 확률이 같다고 가정하면, 보강된 보간검색의 <u>평균 시간복잡도</u>는 $A(n) \approx \Theta(\lg(\lg n))$ 이 되고, 최악의 경우는 $W(n) \approx \Theta((\lg n)^2)$

[실습프로그램] 보강된 보간검색

```
import math
def robustInterpolationSearch(S,x):
  low = 0
  high= len(S)-1
  location= -1


     구현









S = [1,3,4,7,8,11,13,15,16,20,22,25,29,30,33,36,37,39,41,43,45,48]
x = 11
print(S)
print(f’Location of {x:d} is {robustInterpolationSearch(S,x))):d}th’)
```

```
[1, 3, 4, 7, 8, 11, 13, 15, 16, 20, 22, 25, 29, 30, 33, 36, 37, 39, 41, 43, 45, 48]
Location of 11 is 5th
>>>
```

[실습프로그램] closed hashing

```
# closed hashing data into 0 ... M-1
M=10

#conver d to integer.
# each char is convered to ascii code number
def hashing(data):
  hash_data =[-1 for i in range(0,M-1)]
  for d in data:
      s = 0
      for x in d:
         s+=ord(x)
      s = s%M
      hash_data[s]=d
  return hash_data

data =["abc", "name", "school", "KHU"]
print(hashing(data))
```

ord('a')=97 ascii code값

```
[-1, -1, 'KHU', -1, 'abc', -1, -1, 'name', 'school']
>>>
```

[연습] collision이 발생할 경우 저장할 수 있는 방법을 구현