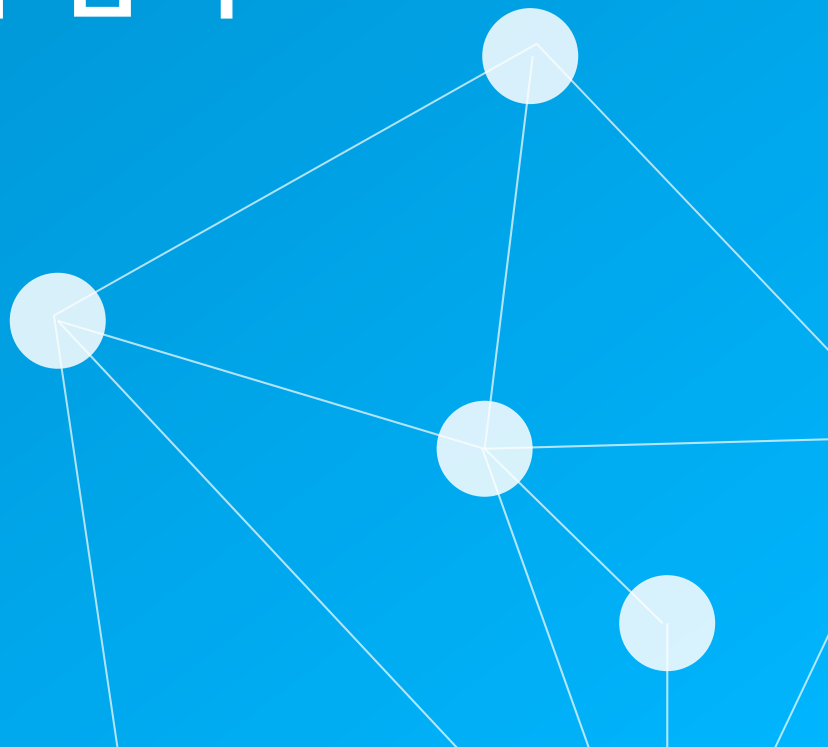
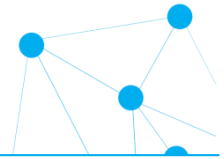
파이썬
자료구조

07 CHAPTER

정렬과 탐색

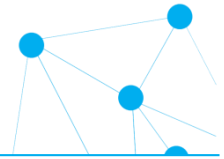


7장. 학습 목표



- 정렬의 개념과 간단한 정렬 알고리즘의 동작 원리를 이해한다.
- 정렬을 이용해 집합 관련 연산의 효율을 향상시키는 방법을 이해한다.
- 탐색의 개념과 간단한 탐색 알고리즘의 동작 원리를 이해한다.
- 해싱의 개념과 해시함수, 오버플로의 개념을 이해한다.
- 오버플로 해결방법을 이해하고, 다양한 방법으로 맵을 구현할 수 있다.

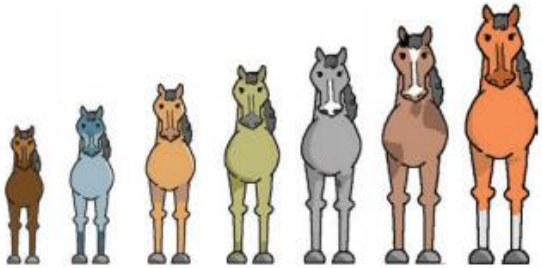
7.1 정렬이란?



- 정렬이란?
- 용어들
- 정렬 알고리즘 종류

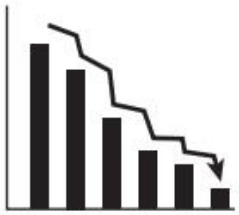
정렬이란?

- 데이터를 순서대로 재배열하는 것
  - 가장 기본적이고 중요한 알고리즘
  - 비교할 수 있는 모든 속성들은 정렬의 기준이 될 수 있다
  - 오름차순(ascending order)과 내림차순(descending order)



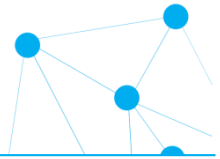
경주마의 정렬(키 순)

오름차순정렬

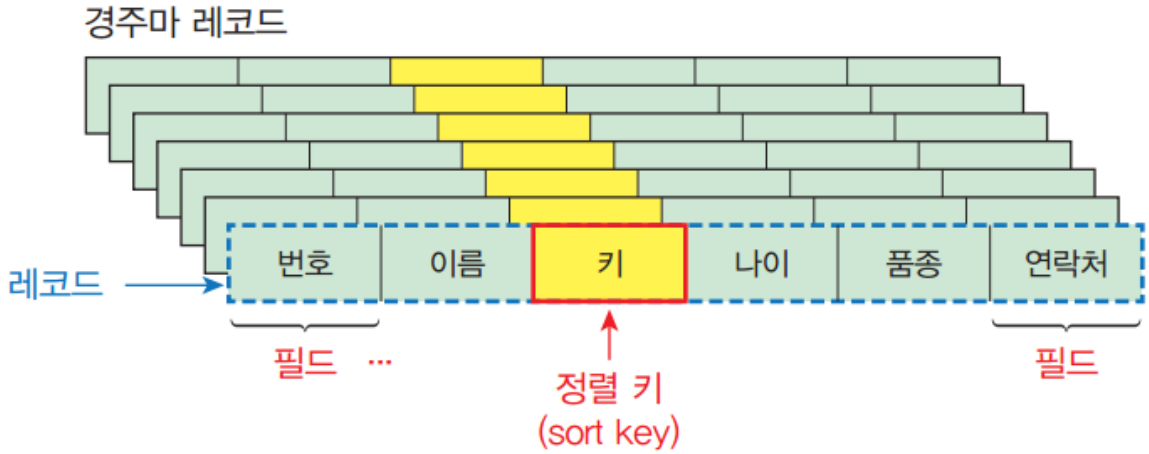


내림차순정렬

용어들

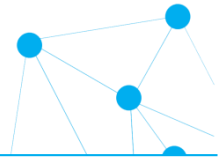


- 레코드: 정렬시켜야 될 대상
  - 여러 개의 필드(field)로 이루어짐
  - 정렬 키(sort key): 정렬의 기준이 되는 필드

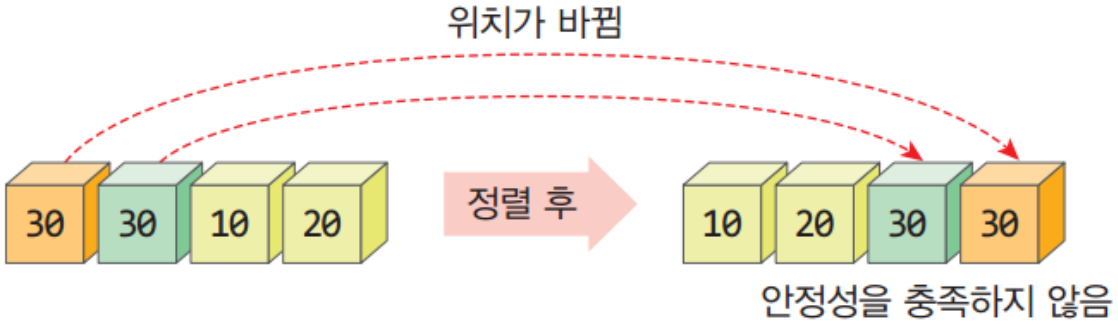


- 정렬이란 레코드들을 키(key)의 순서로 재배열하는 것

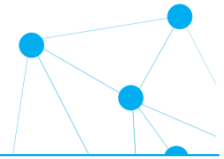
정렬 알고리즘 종류



- 정렬 장소에 따른 분류
  - 내부(internal) 정렬: 모든 데이터가 메인 메모리
  - 외부(external) 정렬: 외부 기억 장치에 대부분의 레코드
- 단순하지만 비효율적인 방법
  - 삽입, 선택, 버블정렬 등
- 복잡하지만 효율적인 방법
  - 퀵, 힙, 병합, 기수정렬, 팀 등
- 정렬 알고리즘의 안정성(stability)

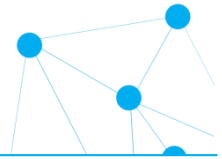


7.2 간단한 정렬 알고리즘



- 선택 정렬(selection sort)
- 삽입 정렬(insertion sort)
- 버블 정렬(bubble sort)

선택 정렬(selection sort)



- 오른쪽 리스트에 서 가장 작은 숫자를 선택하여 왼쪽 리스트의 맨 뒤로 이동하는 작업을 반복

정렬 된(왼쪽) 리스트	정렬 안 된(오른쪽) 리스트	설명
[ ]	[5,3,8,4,9,1,6,2,7]	초기상태
[1]	[5,3,8,4,9,6,2,7]	1선택 및 이동
[1,2]	[5,3,8,4,9,6,7]	2선택 및 이동
[1,2,3]	[5,8,4,9,6,7]	3선택 및 이동
…	…	4~8 선택 및 이동
[1,2,3,4,5,6,7,8,9]	[ ]	9선택 및 이동

선택 정렬 알고리즘

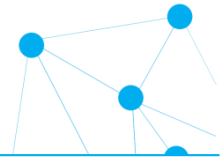

```
def selection_sort(A) :
    n = len(A)
    for i in range(n-1) :
        least = i;
        for j in range(i+1, n) :
            if (A[j]<A[least]) :
                least = j
        A[i], A[least] = A[least], A[i]
        printStep(A, i + 1);
```

- 시간 복잡도

$$(n-1)+(n-2)+\cdots+1=n(n-1)/2=O(n^2)$$

  - 알고리즘이 간단, 자료 이동 횟수가 미리 결정됨

테스트 프로그램



```
data = [ 5, 3, 8, 4, 9, 1, 6, 2, 7 ]
print("Original  : ", data)
selection_sort(data)
print("Selection : ", data)
```

```
C:₩WINDOWS₩system32₩cmd.exe
Original  : [5, 3, 8, 4, 9, 1, 6, 2, 7]
  Step  1 = [1, 3, 8, 4, 9, 5, 6, 2, 7]
  Step  2 = [1, 2, 8, 4, 9, 5, 6, 3, 7]
  Step  3 = [1, 2, 3, 4, 9, 5, 6, 8, 7]
  Step  4 = [1, 2, 3, 4, 9, 5, 6, 8, 7]
  Step  5 = [1, 2, 3, 4, 5, 9, 6, 8, 7]
  Step  6 = [1, 2, 3, 4, 5, 6, 9, 8, 7]
  Step  7 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
  Step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Selection : [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

정렬 안 된 리스트

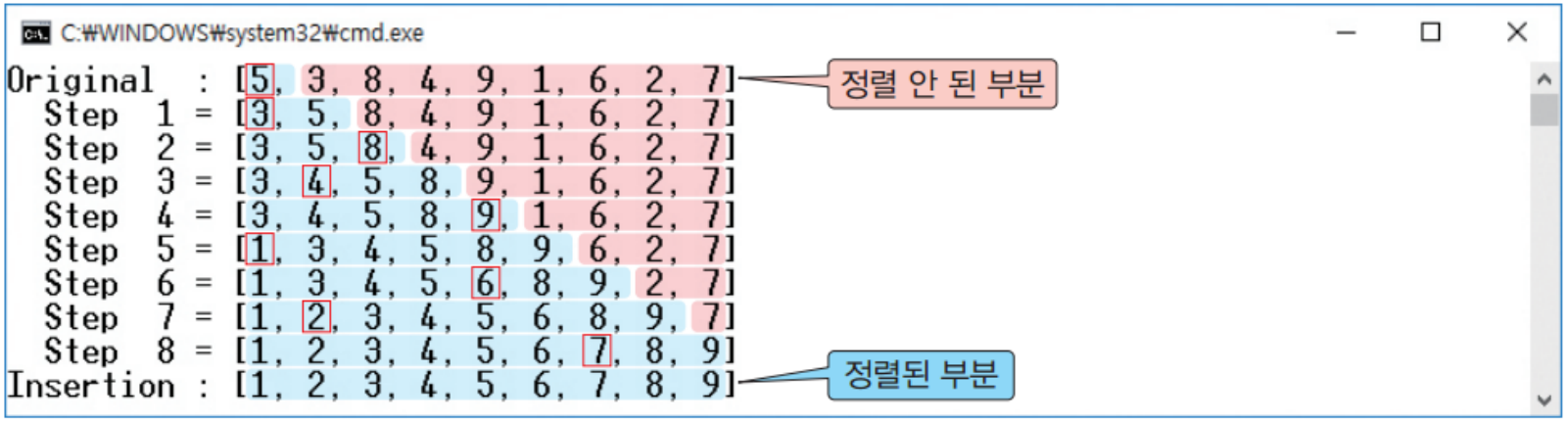
정렬된 리스트

삽입 정렬(insertion sort)

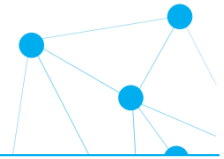
- 정렬되어 있는 부분에 새로운 레코드를 올바른 위치에 삽입하는 과정 반복

삽입 정렬 알고리즘


```
def insertion_sort(A) :
    n = len(A)
    for i in range(1, n) :
        key = A[i]
        j = i-1
        while j>=0 and A[j] > key :
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key
        printStep(A, i)
```

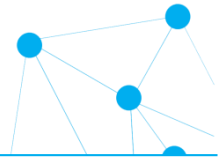


삽입 정렬 분석



- 복잡도 분석
  - 최선의 경우 O(n) : 이미 정렬되어 있는 경우: 비교: *n-1* 번
  - 최악의 경우 O(n^2) : 역순으로 정렬되어 있는 경우
    - 모든 단계에서 앞에 놓인 자료 전부 이동
    - 비교: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
    - 이동: $\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$
  - 평균의 경우 O(n^2)
- 특징
  - 많은 이동 필요 → 레코드가 큰 경우 불리
  - 안정된 정렬방법
  - 대부분 정렬되어 있으면 매우 효율적

버블 정렬 (bubble sort)

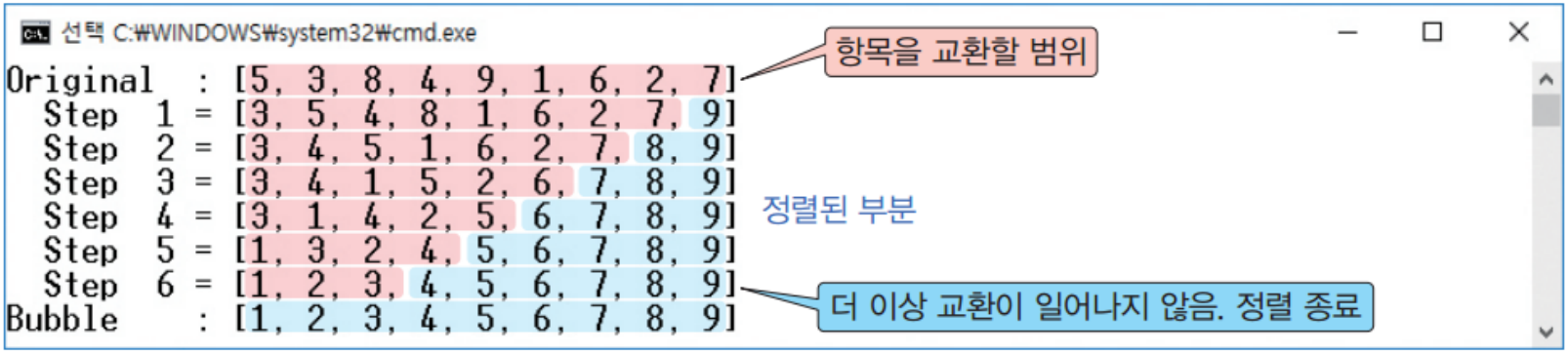


- 기본 전략
  - 인접한 2개의 레코드를 비교하여 순서대로 서로 교환
  - 비교-교환 과정을 리스트의 전체에 수행(스캔)
    - 한번의 스캔이 완료되면 리스트의 오른쪽 끝에 가장 큰 레코드
  - 끝으로 이동한 레코드를 제외하고 다시 스캔 반복

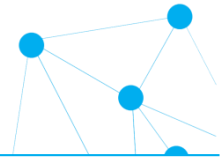
버블 정렬 알고리즘


```
def bubble_sort(A) :
    n = len(A)
    for i in range(n-1, 0, -1) :
        bChanged = False
        for j in range (i) :
            if (A[j]>A[j+1]) :
                A[j], A[j+1] = A[j+1], A[j]
                bChanged = True

        if not bChanged: break;
        printStep(A, n - i);
```



버블정렬 분석



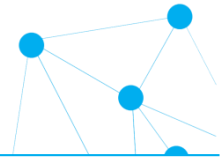
- 비교 횟수(최상, 평균, 최악의 경우 모두 일정)

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- 이동 횟수
  - 역순으로 정렬된 경우(최악): 이동 횟수 = 3 * 비교 횟수
  - 이미 정렬된 경우(최선의 경우) : 이동 횟수 = 0
  - 평균의 경우 : $O(n^2)$

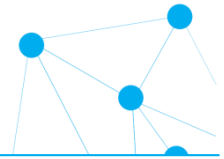
- 레코드의 이동 과다
  - 이동연산은 비교연산 보다 더 많은 시간이 소요됨

7.3 정렬 응용: 집합 다시 보기



- 정렬된 리스트를 이용한 집합
- 비교 연산: __eq__
- 합집합/교집합/차집합
- 복잡도 비교

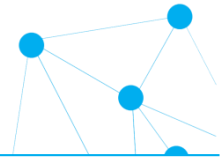
정렬 응용: 집합 다시 보기



- 3장에서 구현한 집합 자료구조 수정하기
  - 집합의 원소들을 항상 정렬된 순으로 저장
  - 삽입 연산은 더 복잡해 짐
  - 집합의 비교나 합집합, 차집합, 교집합 → 효율적 구현 가능
- 삽입 연산
  - 삽입할 위치를 먼저 찾아야 함.

```
def insert(self, elem) :                    # 정렬된 상태를 유지하면서 elem을 삽입
    if elem in self.items : return          # 이미 있음
    for idx in range(len(self.items)) :     # loop: n번
        if elem < self.items[idx] :         # 삽입할 위치 idx를 찾음
            self.items.insert(idx, elem)    # 그 위치에 삽입
            return
    self.items.append(elem)                 # 맨 뒤에 삽입
```

비교 연산: __eq__

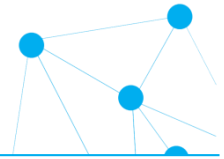


- 두 집합의 비교 방법
  - 두 집합의 원소의 개수가 같아야 같은 집합이 됨
  - 집합이 정렬되어 있으므로 순서대로 같은 원소를 가져야 함

```
def __eq__( self, setB ):                    # 두 집합 self, setB가 같은 집합인가?
    if self.size() != setB.size() :          # 원소의 개수가 같아야 함
        return False
    for idx in range(len(self.items)):       # loop: n번
        if self.items[idx] != setB.items[idx] :   # 원소별로 같은지 검사
            return False
    return True
```

- 시간 복잡도:$O(n^2)$ ➔ $O(n)$으로 개선

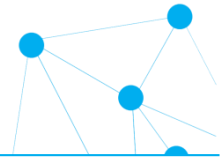
합집합/교집합/차집합



- 합집합 연산 방법
  - 가장 작은 원소들부터 비교하여 더 작은 원소를 새로운 집합에 넣고 그 집합의 인덱스를 증가시킴.
  - 만약 두 집합의 현재 원소가 같으면 하나만을 넣음. 인덱스는 모두 증가시킴.
  - 한쪽 집합이 모두 처리되면 나머지 집합의 남은 모든 원소를 순서대로 새 집합에 넣음

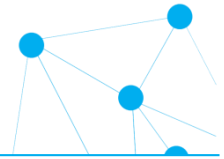
- 시간 복잡도:$O(n^2)$ ➔ $O(n)$으로 개선
- 교집합과 차집합도 동일한 방법 적용 가능

개선된 합집합 알고리즘



```
def union( self, setB ):                        # 집합 self와 집합 setB의 합집합
    newSet = Set()                              # 반환할 합집합
    a = 0                                       # 집합 self의 원소에 대한 인덱스
    b = 0                                       # 집합 setB의 원소에 대한 인덱스
    while a < len( self.items ) and b < len( setB.items ) :
        valueA = self.items[a]                  # 집합 self의 현재 원소
        valueB = setB.items[b]                  # 집합 setB의 현재 원소
        if valueA < valueB :                    # self의 원소가 더 작으면
            newSet.items.append( valueA )       # 이 원소를 합집합에 추가
            a += 1                              # self의 현재원소 인덱스 증가.
        elif valueA > valueB :                  # setB의 원소가 더 작으면
            newSet.items.append( valueB )       # 이 원소를 합집합에 추가
            b += 1                              # setB의 현재원소 인덱스 증가.
        else :                                  # 중복되는 원소
            newSet.items.append( valueA )       # 하나만 추가
            a += 1                              # self와 setB의 인덱스 모두 증가
            b += 1
    while a < len( self.items ):                # self에 남은 원소를 모두 추가
        newSet.items.append( self.items[a] )
        a += 1
    while b < len( setB.items) :                # setB에 남은 원소를 모두 추가
        newSet.items.append( setB.items[b] )
        b += 1
    return newSet                               # 합집합 반환
```

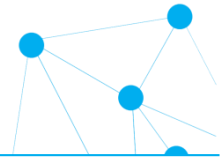
복잡도 비교



[표 7.1] 정렬되지 않은 리스트와 정렬된 리스트로 구현한 집합에서의 복잡도 비교

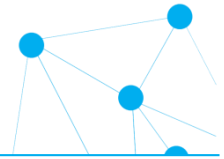
집합의 연산	정렬되지 않은 리스트	정렬된 리스트
`insert(e)`	$O(n)$	$O(n)$
`__eq__(setB)`	$O(n^2)$	$O(n)$
`union(setB)`	$O(n^2)$	$O(n)$
`intersect(setB)`	$O(n^2)$	$O(n)$
`difference(setB)`	$O(n^2)$	$O(n)$

7.4 탐색과 맵 구조

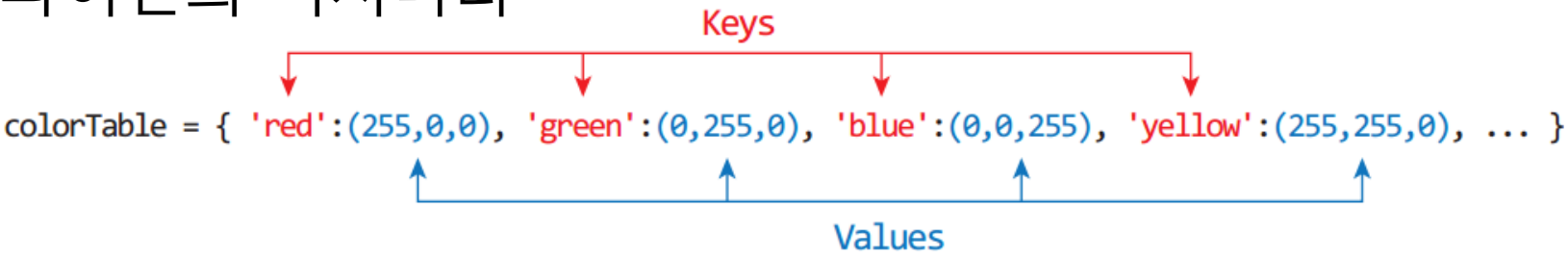


- 탐색, 맵, 엔트리, 딕셔너리
- 맵 ADT

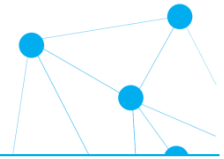
탐색, 맵, 엔트리, 딕셔너리



- 탐색
  - 테이블에서 원하는 탐색키를 가진 레코드를 찾는 작업
- 맵(map) 또는 딕셔너리 (dictionary)
  - 탐색을 위한 자료구조
  - 엔트리(entry), 또는 키를 가진 레코드 (keyed record)의 집합
- 엔트리
  - 키(key): 영어 단어와 같은 레코드를 구분할 수 있는 탐색키
  - 값(value): 단어의 의미와 같이 탐색키와 관련된 값
- 파이썬의 딕셔너리



맵 ADT



정의 7.1 Map ADT

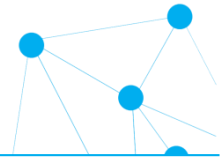
데이터: 키를 가진 레코드(엔트리)의 집합

연산

- search(key): 탐색키 key를 가진 레코드를 찾아 반환한다.
- insert(entry): 주어진 entry를 맵에 삽입한다.
- delete(key): 탐색키 key를 가진 레코드를 찾아 삭제한다.

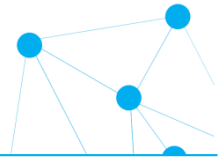
- 맵을 구현하는 방법
  - 리스트 이용: 정렬 / 비정렬
  - 이진 탐색 트리 이용 (9장)
  - 해싱 구조 이용

7.5 간단한 탐색 알고리즘



- 순차 탐색(sequential search)
- 이진 탐색(binary search)
- 보간 탐색(interpolation search)

순차 탐색(sequential search)



- 정렬되지 않은 배열에 적용 가능
  - 정렬되지 않은 배열을 처음부터 마지막까지 하나씩 검사
  - 가장 간단하고 직접적인 탐색 방법
  - 평균 비교 횟수: (n + 1)/2번 비교 (최악의 경우: n번)

8을 찾는 경우

9	5	8	3	7	9≠8 탐색 계속
9	5	8	3	7	5≠8 탐색 계속
9	5	8	3	7	8=8 탐색 성공

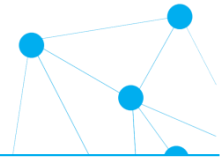
탐색성공 → 종료

2를 찾는 경우

9	5	8	3	7	9≠2 탐색 계속
9	5	8	3	7	5≠2 탐색 계속
9	5	8	3	7	8≠2 탐색 계속
9	5	8	3	7	3≠2 탐색 계속
9	5	8	3	7	7≠2 탐색 계속

더 이상 항목이 없음 → 탐색실패

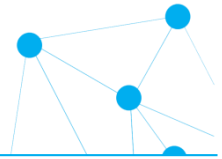
순차 탐색 알고리즘



```
def sequential_search(A, key, low, high) :      # 순차탐색
    for i in range(low, high+1) :               # i : low, low+1, ... high
        if A[i].key == key :                    # 탐색 성공하면
            return i                            # 인덱스 반환
    return None                                 # 탐색에 실패하면 None 반환
```

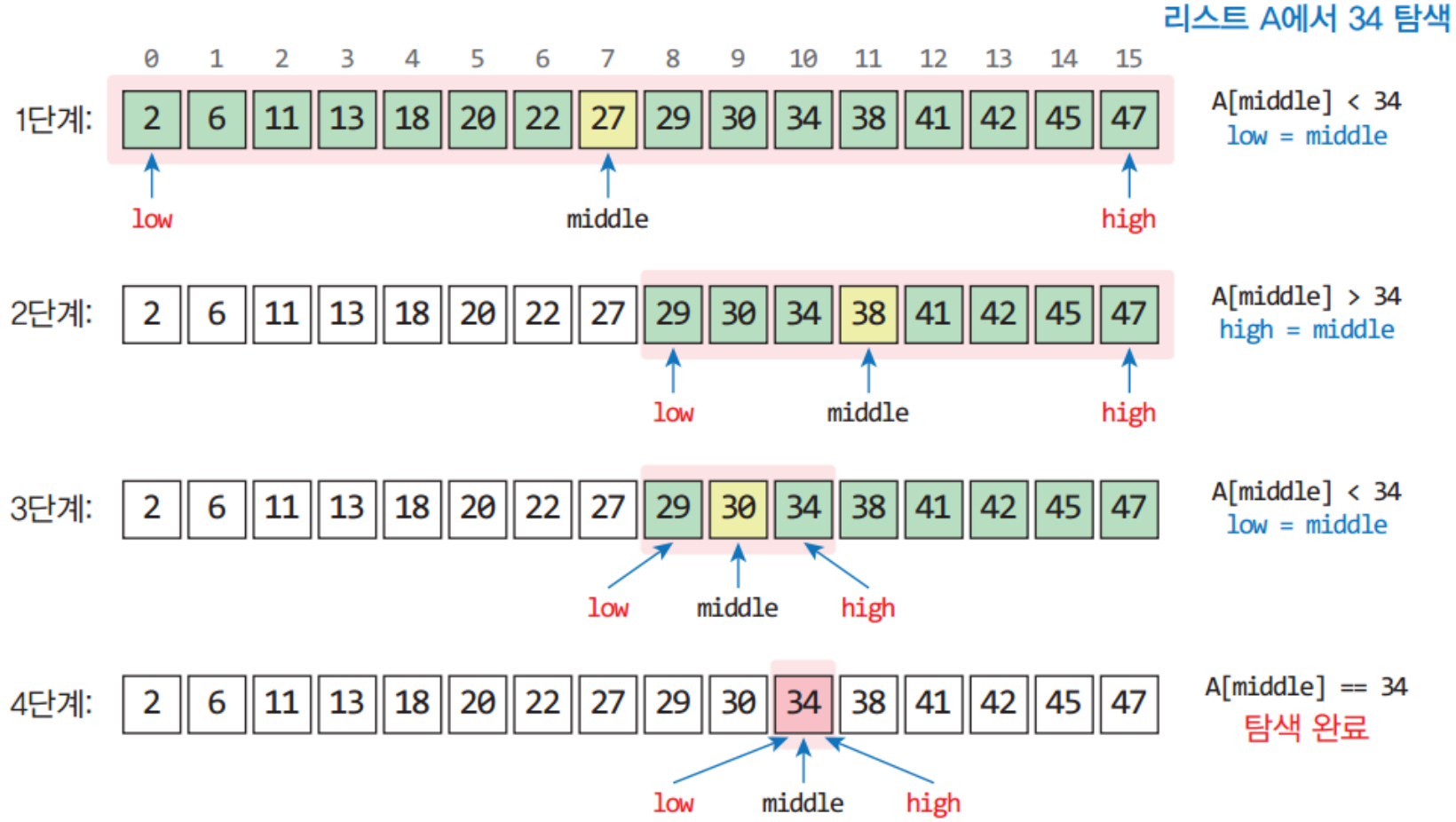
- 시간 복잡도: $O(n)$

이진 탐색(binary search)

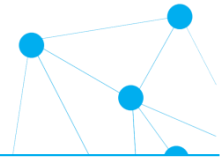


- 정렬된 배열의 탐색에 적합
  - 배열의 중앙에 있는 값을 조사하여 찾고자 하는 항목이 왼쪽 또는 오른쪽 부분 배열에 있는지를 알아내어 탐색의 범위를 반으로 줄여가며 탐색 진행
  - 예) 사전에서 단어 찾기
- (예) 10억 명중에서 특정한 이름 탐색
  - 이진탐색 : 단지 30번의 비교 필요
  - 순차 탐색 : 평균 5억 번의 비교 필요

이진 탐색


리스트 A에서 34 탐색
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1단계: 2 6 11 13 18 20 22 27 29 30 34 38 41 42 45 47
low
middle
high
A[middle] < 34
low = middle
2단계: 2 6 11 13 18 20 22 27 29 30 34 38 41 42 45 47
low
middle
high
A[middle] > 34
high = middle
3단계: 2 6 11 13 18 20 22 27 29 30 34 38 41 42 45 47
low
middle
high
A[middle] < 34
low = middle
4단계: 2 6 11 13 18 20 22 27 29 30 34 38 41 42 45 47
low
middle
high
A[middle] == 34
탐색 완료

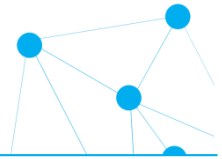
이진 탐색 알고리즘



```
def binary_search(A, key, low, high) :
    if (low <= high) :                          # 항목들이 남아 있으면(종료 조건)
        middle = (low + high) // 2              # 정수 나눗셈 //에 주의할 것.
        if key == A[middle].key :               # 탐색 성공
            return middle
        elif (key<A[middle].key) :              # 왼쪽 부분리스트 탐색
            return binary_search(A, key, low, middle - 1)
        else :                                  # 오른쪽 부분리스트 탐색
            return binary_search(A, key, middle + 1, high)
    return None                                 # 탐색 실패
```

- 시간 복잡도: $O(logn)$
- 반복으로 구현 가능

보간 탐색(interpolation search)



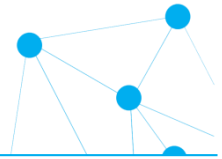
- 탐색키가 존재 할 위치를 예측하여 탐색
  - 예) 사전이나 전화번호부를 탐색할 때
    - 'ㅎ'으로 시작하는 단어는 사전의 뒷부분에서 찾음
    - 'ㄱ'으로 시작하는 단어는 앞부분에서 찾음

- 리스트를 불균등하게 분할하여 탐색
  - 탐색 값과 위치는 비례한다는 가정

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

```
middle = int(low + (high-low) * (key-A[low].key) / (A[high].key-A[low].key))
```

7.6 고급 탐색 구조: 해싱

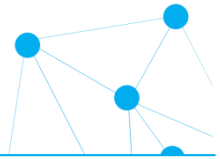


- 해싱이란?
- 선형 조사에 의한 오버플로 처리
- 체이닝(chaining)에 의한 오버플로 처리
- 해시 함수
- 탐색 방법들의 성능 비교

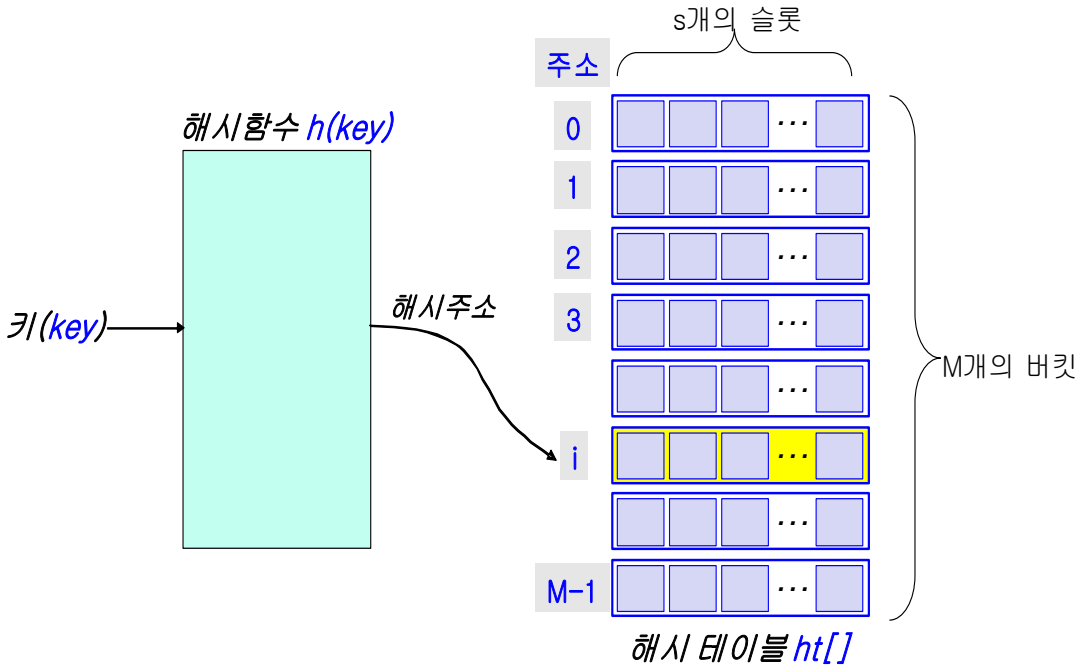
해싱이란?

- 해싱(hashing)
  - 키 값에 대한 산술적 연산에 의해 테이블의 주소를 계산
  - 해시 테이블(hash table)
    - 키 값의 연산에 의해 직접 접근이 가능한 구조

해싱의 구조



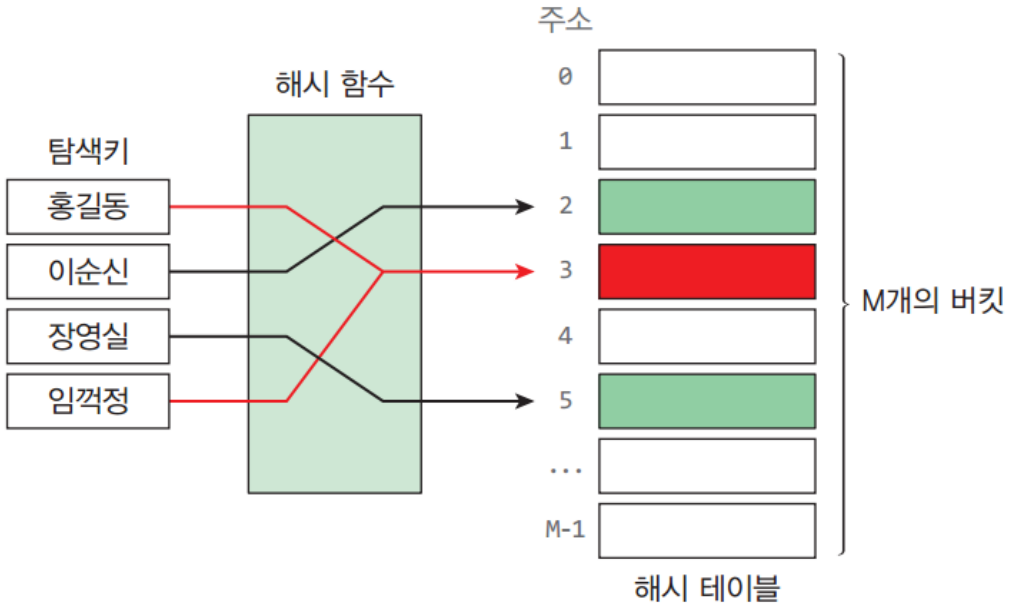
- 해시 테이블, 버킷, 슬롯
- 해시 함수(hash function)
  - 탐색키를 입력받아 해시 주소(hash address) 생성



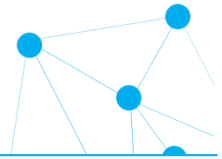
충돌과 오버플로

- 충돌
  - 서로 다른 키 가 해시 함수에 의해 같은 주소로 계산되는 상황
- 오버플로우
  - 충돌이 슬롯 수보다 많이 발생하는 것

h(홍길동)⇒3, h(이순신)⇒2, h(장영실)⇒5, h(임꺽정)⇒3



선형 조사에 의한 오버플로 처리

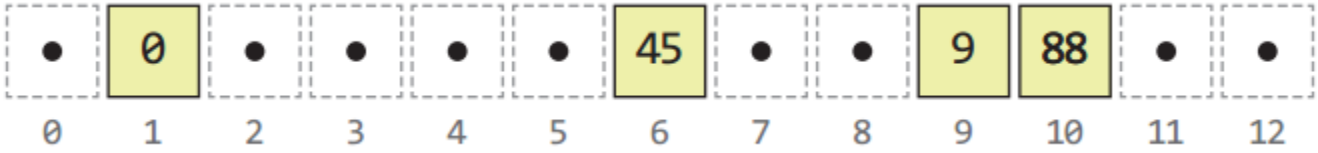


- 충돌이 일어나면 해시 테이블의 다음 위치에 저장
  - 다음 항목을 순서대로 조사: h(k), h(k)+1, h(k)+2,...
  - 빈 곳이 있으면 저장.

- 예) 45, 27, 88, 9, 71, 60, 46, 38, 24 저장 과정

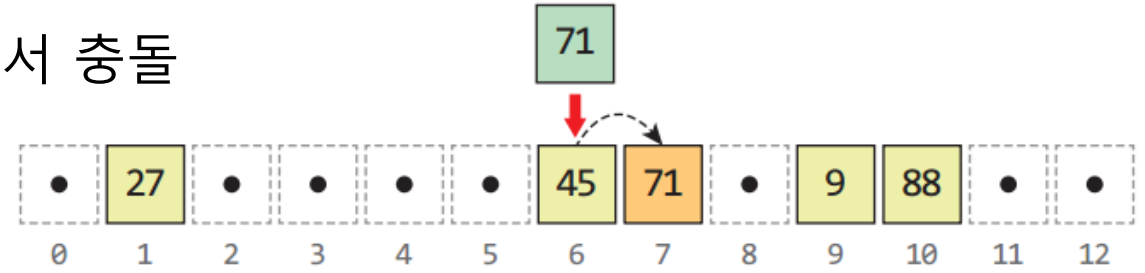
key	45	27	88	9	71	60	46	38	24
h(key)	6	1	10	9	6	8	7	12	11

① 45, 27, 88, 9 까지의 삽입



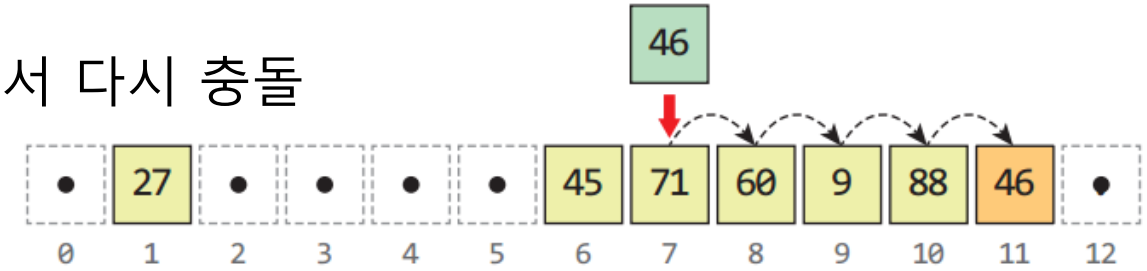
선형 조사: 삽입 연산

② 71의 삽입에서 충돌

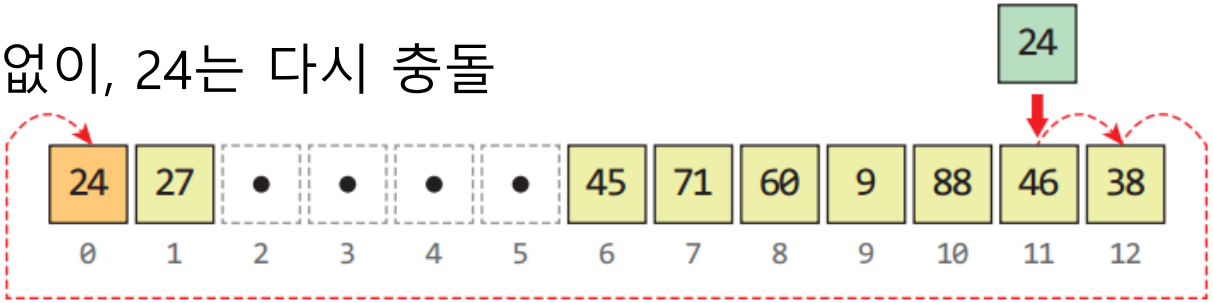


③ 60의 삽입

④ 46의 삽입에서 다시 충돌

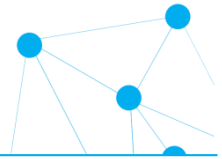


⑤ 38은 충돌 없이, 24는 다시 충돌

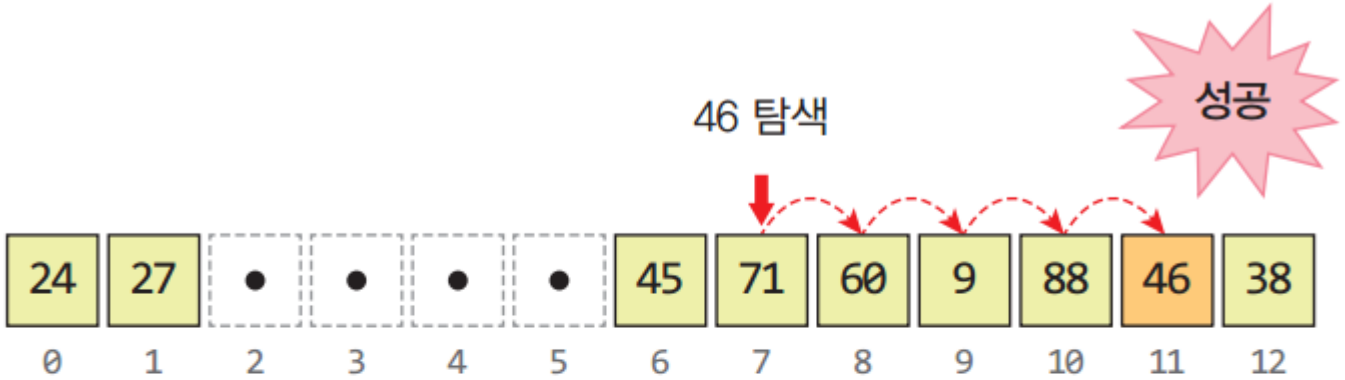


군집화 현상

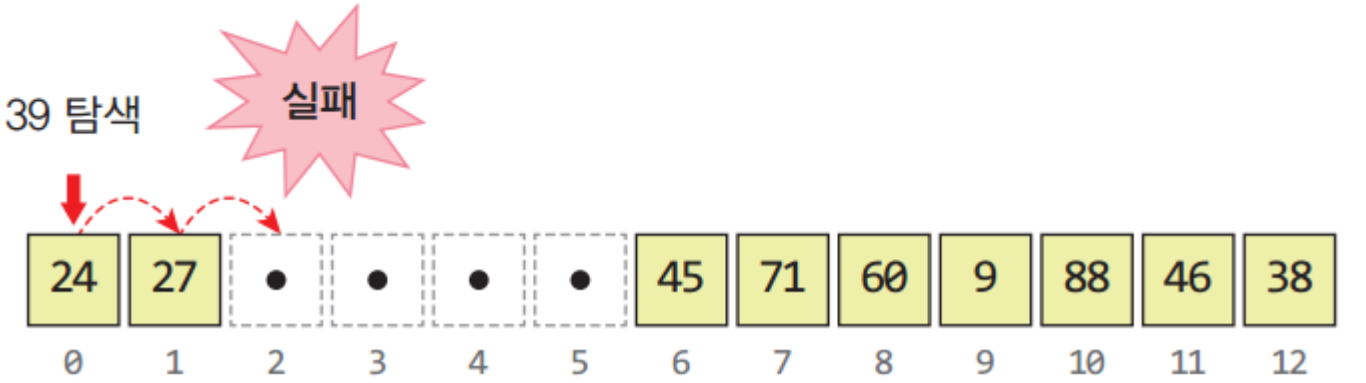
선형 조사: 탐색 연산



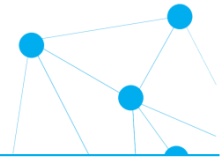
- 46 탐색



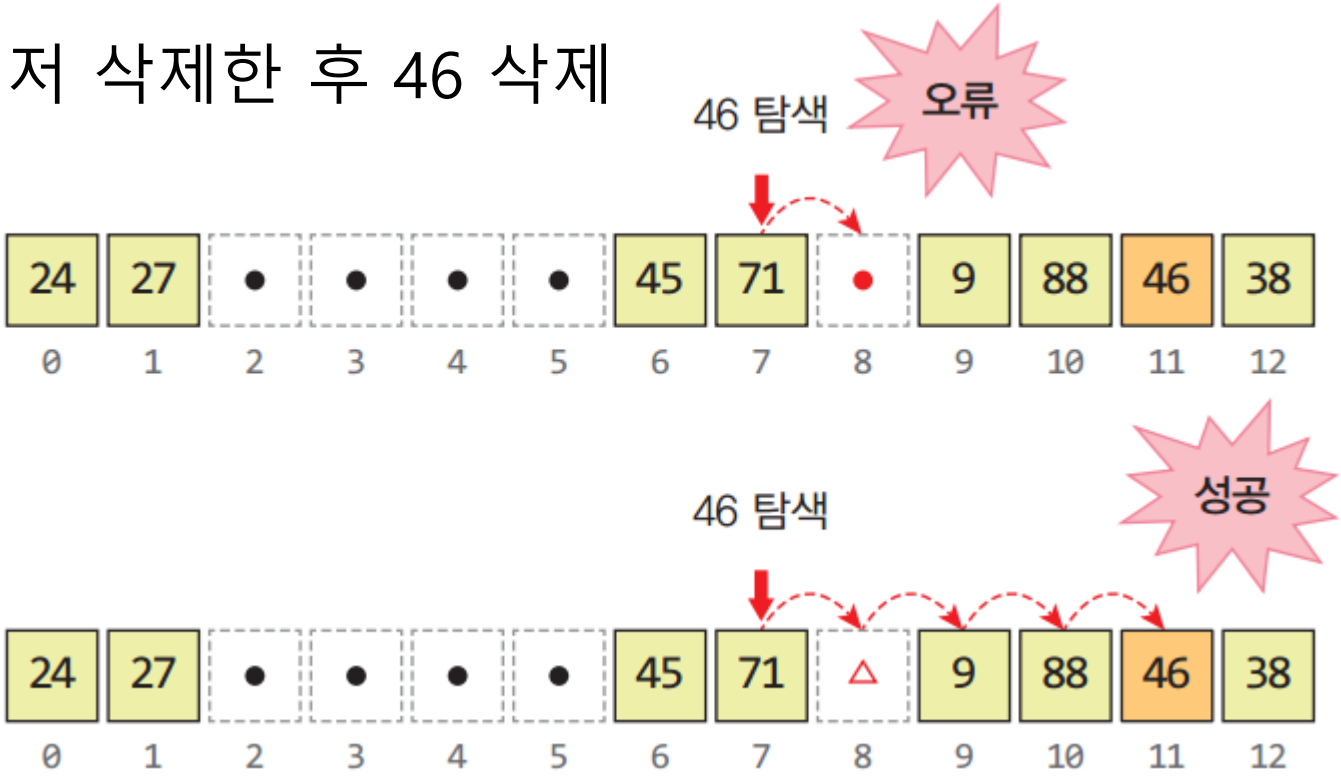
- 39 탐색



선형 조사: 삭제 연산

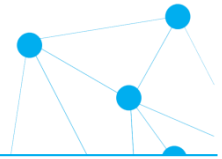


- 60을 먼저 삭제한 후 46 삭제



- 빈 버킷을 두 가지로 분류해야 함.

선형 조사 군집화 완화 방법

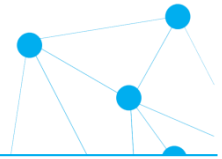


- 이차 조사법(quadratic probing)

$$(h(k)+i*i)\ \%\ M \quad \text{for} \quad i=0,1,\cdots,M-1$$

- 이중 해싱법(double hashing)
  - 재해싱(rehashing)
  - 충돌이 발생하면, 다른 해시 함수를 이용해 다음 위치 계산

체이닝에 의한 오버플로 처리



- 하나의 버킷에 여러 개의 레코드를 저장할 수 있도록 하는 방법

- 예) h(k)=k%7 을 이용해 8, 1, 9, 6, 13 을 삽입

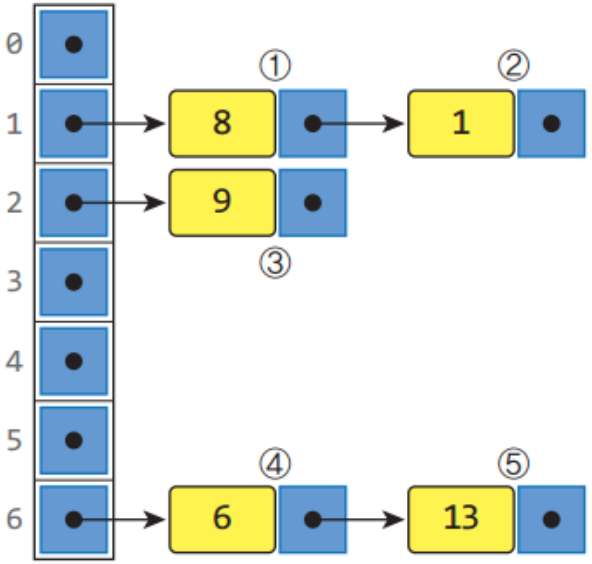
① 8 저장 : h(8) = 8 % 7 = 1 ⇒ 저장

② 1 저장 : h(1) = 1 % 7 = 1 ⇒ 충돌 ⇒ 새로운 노드 생성 및 저장

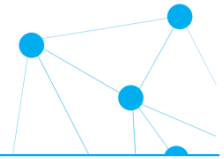
③ 9 저장 : h(9) = 9 % 7 = 2 ⇒ 저장

④ 6 저장 : h(6) = 6 % 7 = 6 ⇒ 저장

⑤ 13저장 : h(13) = 13 % 7 = 6 ⇒ 충돌 ⇒ 새로운 노드 생성 및 저장



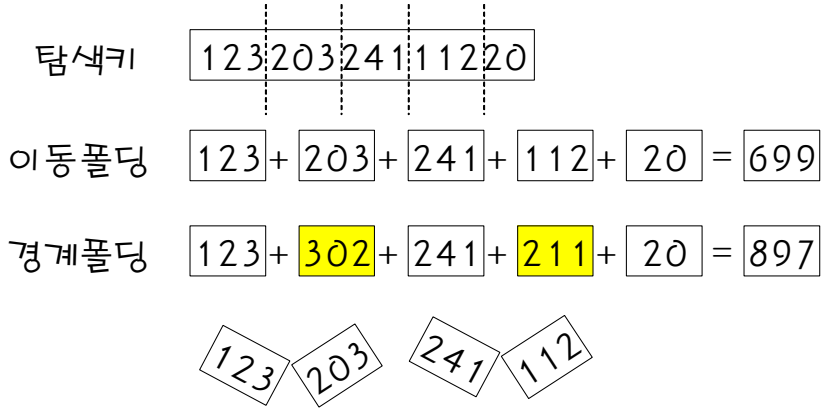
해시 함수



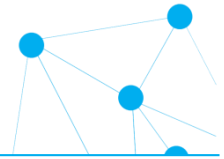
- 좋은 해시 함수의 조건
  - 충돌이 적어야 한다
  - 함수 값이 테이블의 주소 영역 내에서 고르게 분포되어야 한다
  - 계산이 빨라야 한다

- 제산 함수
  - h(k)=k mod M
  - 해시 테이블의 크기 M은 소수(prime number) 선택

- 폴딩 함수



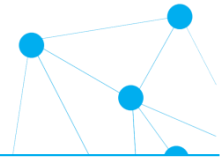
해시 함수



- 중간 제곱 함수
  - 탐색키를 제곱한 다음, 중간의 몇 비트를 취해서 해시 주소 생성
- 비트 추출 함수
  - 키를 이진수로 간주. 임의의 위치의 k개의 비트를 사용
- 숫자 분석 방법
  - 키에서 편중되지 않는 수들을 테이블의 크기에 적합하게 조합
- 탐색키가 문자열인 경우

```
def hashFn(key) :
    sum = 0
    for c in key :                  # 문자열의 모든 문자에 대해
        sum = sum +  ord(c)         # 그 문자의 아스키 코드 값을 sum에 더함
    return sum % M
```

탐색 방법들의 성능 비교



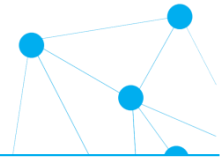
- 해싱의 적재 밀도(loading density) 또는 적재 비율
  - 저장되는 항목의 개수 n과 해시 테이블의 크기 M의 비율

$$\alpha = \frac{\text{저장된 항목의 개수}}{\text{해싱테이블의 버킷의 개수}} = \frac{n}{M}$$

- 다양한 탐색 방법의 성능 비교

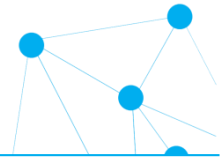
탐색방법		탐색	삽입	삭제
순차탐색		$O(n)$	$O(1)$	$O(n)$
이진탐색		$O(\log_2 n)$	$O(n)$	$O(n)$
이진탐색트리	균형트리	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
	경사트리	$O(n)$	$O(n)$	$O(n)$
해싱	최선의 경우	$O(1)$	$O(1)$	$O(1)$
	최악의 경우	$O(n)$	$O(n)$	$O(n)$

7.7 맵의 응용: 나의 단어장



- 리스트를 이용한 순차탐색 맵
- 체이닝을 이용한 해시 맵
- 파이썬의 딕셔너리를 이용한 구현

맵의 응용: 나의 단어장



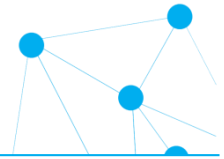
- 맵 ADT를 구현
  - 리스트를 이용해 순차 탐색 맵을 구현하는 방법
  - 리스트를 정렬해서 이진 탐색 맵을 구현하는 방법
  - 선형조사법으로 해시 맵을 구현하는 방법
  - 체이닝으로 해시 맵을 구현하는 방법

- 엔트리 클래스

```
class Entry:
    def __init__( self, key, value ):
        self.key = key
        self.value = value

    def __str__( self ):
        return str("%s:%s"%(self.key, self.value) )
```

리스트를 이용한 순차탐색 맵



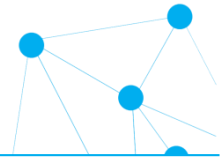
```
class SequentialMap:                                    # 순차탐색 맵
    def __init__( self ):
        self.table = []                                 # 맵의 레코드 테이블

    def insert(self, key, value) :                      # 삽입 연산
        self.table.append(Entry(key, value))            # 리스트의 맨 뒤에 추가

    def search(self, key) :                             # 순차 탐색 연산
        pos = sequential_search(self.table, key, 0, self.size()-1)
        if pos is not None : return self.table[pos]
        else : return None

    def delete(self, key) :                             # 삭제 연산: 항목 위치를 찾아 pop
        for i in range(self.size()):
            if self.table[i].key == key :               # 삭제할 위치를 먼저 찾고
                self.table.pop(i)                       # 리스트의 pop으로 삭제
                return
```

테스트 프로그램



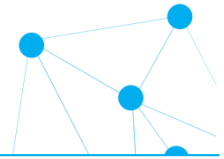
```
map = SequentialMap()
map.insert('data', '자료')
map.insert('structure', '구조')
map.insert('sequential search', '선형 탐색')
map.insert('game', '게임')
map.insert('binary search', '이진 탐색')
map.display("나의 단어장: ")

print("탐색:game --> ", map.search('game'))
print("탐색:over --> ", map.search('over'))
print("탐색:data --> ", map.search('data'))

map.delete('game')
map.display("나의 단어장: ")
```

```
C:₩WINDOWS₩system32₩cmd.exe
나의 단어장:
   data:자료
   structure:구조
   sequential search:선형 탐색
   game:게임
   binary search:이진 탐색
탐색:game -->  game:게임
탐색:over -->  None
탐색:data -->  data:자료
나의 단어장:
   data:자료
   structure:구조
   sequential search:선형 탐색
   binary search:이진 탐색
```

체이닝을 이용한 해시 맵



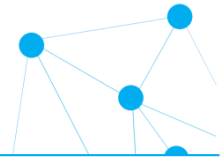
```
class HashChainMap:
    def __init__( self, M ):
        self.table = [None]*M
        self.M = M
```

```
    def hashFn(self, key) :
        sum = 0
        for c in key :
                sum = sum +  ord(c)
        return sum % self.M
```

```
    def insert(self, key, value) :                          # (key,value) 입력
        idx = self.hashFn(key)                              # 해시 주소 계산
        self.table[idx] = Node(Entry(key,value), self.table[idx])   # 전단 삽입
```

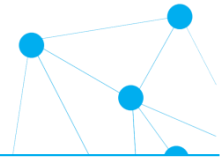
```
        entry = Entry(key,value)          # (1) 엔트리를 생성
        node = Node(entry)                # (2) 엔트리로 노드를 생성
        node.link = self.table[idx]       # (3) 노드의 링크필드 처리
        self.table[idx] = node            # (4) 테이블의 idx 항목: node로 시작
```

테스트 프로그램



```
C:₩WINDOWS₩system32₩cmd.exe
나의 단어장:
[ 3] -> sequential search:선형 탐색 ->
[ 7] -> binary search:이진 탐색 -> game:게임 -> data:자료 ->
[ 8] -> structure:구조 ->
 탐색:game -->  game:게임
 탐색:over -->  None
 탐색:data -->  data:자료
나의 단어장:
[ 3] -> sequential search:선형 탐색 ->
[ 7] -> binary search:이진 탐색 -> data:자료 ->
[ 8] -> structure:구조 ->
```

파이썬의 딕셔너리를 이용한 구현



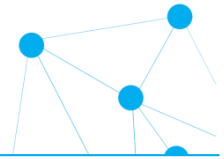
```
d = {}                                          # 딕셔너리(맵) 객체를 만듦
d['data'] =  '자료'                             # 맵에 엔트리를 삽입
d['structure'] = '구조'
d['sequential search'] = '선형 탐색'
d['game'] = '게임'
d['binary search'] = '이진 탐색'
print("나의 단어장:")
print(d)                                        # 맵 출력

if d.get('game') : print("탐색:game --> ", d['game'])      # 탐색
if d.get('over') : print("탐색:over --> ", d['over'])      # 탐색
if d.get('data') : print("탐색:data --> ", d['data'])      # 탐색

d.pop('game')
print("나의 단어장:")
print(d)
```

```
C:₩WINDOWS₩system32₩cmd.exe
나의 단어장:
{'data': '자료', 'structure': '구조', 'sequential search': '선형 탐색', 'game': '
게임', 'binary search': '이진 탐색'}
탐색:game -->  게임
탐색:data -->  자료
나의 단어장:
{'data': '자료', 'structure': '구조', 'sequential search': '선형 탐색', 'binary s
earch': '이진 탐색'}
```

7장 연습문제, 실습문제



감사합니다!