

# <자료구조 및 실습>


## 11. 검색

한국외국어대학교  
컴퓨터.전자시스템공학전공  
2016년 1학기  
고 석 훈

# 학습 목표

- 자료에 대한 검색의 개념을 이해한다.
- 순차 검색의 개념과 알고리즘을 알아본다.
- 이진 검색의 개념과 알고리즘을 알아본다.
- 해싱의 개념에 대해 이해한다.
- 해싱함수의 종류에 대해 알아본다.

# 검색(Search)

- 컴퓨터에 저장한 자료 중에서 원하는 항목을 찾는 작업
  - 검색 성공 - 원하는 항목을 찾은 경우
  - 검색 실패 - 원하는 항목을 찾지 못한 경우
- 탐색 키를 가진 항목을 찾는 것
  -  - 자료를 구별하여 인식할 수 있는 키
- 삽입/삭제 작업에서의 검색
  - 원소를 삽입하거나 삭제할 위치를 찾기 위해서 검색 연산 수행

# 검색 방법

- 수행 위치에 따른 분류
  - 내부 검색 – 메모리 내의 자료에 대해서 검색 수행
  - 외부 검색 – 보조 기억 장치에 있는 자료에 대해서 검색 수행
- 검색 방식에 따른 분류
  - 비교 검색 방식(comparison search method)
    - ◆ 검색 대상의 키를 비교하여 검색하는 방법
    - ◆ 순차 검색, 이진 검색, 트리 검색
  - 계산 검색 방식(non-comparison method)
    - ◆ 계수적인 성질을 이용한 계산으로 검색 하는 방법
    - ◆ 해싱
- 검색 방법의 선택
  - 자료 구조의 형태와 자료의 배열 상태에 따라 최적의 검색 방법 선택

# 순차 검색

- 순차 검색(sequential search, 선형 검색, linear search)
  - 일렬로 된 자료를 처음부터 마지막까지 순서대로 검색하는 방법
  - 배열이나 연결 리스트로 구현된 순차 자료 구조에서 사용
  - 가장 간단하고 직접적인 검색 방법
    - ◆ 장점 – 알고리즘이 단순하여 구현이 용이함
    - ◆ 단점 – 검색 대상 자료가 많은 경우에 비효율적

# 순차 검색 [1/2]

## ● 검색 방법

- 첫 번째 원소부터 시작하여 마지막 원소까지 순서대로 키 값이 일치하는 원소가 있는지를 비교하여 찾는다.
- 키 값이 일치하는 원소를 찾으면 ➔ 몇 번째 원소인지 반환
- 마지막까지 키 값이 일치하는 원소가 없으면 ➔ 검색 실패

# 순차 검색 [2/2]

## ● 검색을 성공하는 경우

9를 검색하는 경우

① $8 \neq 9$	8	30	1	9	11	19	2
② $30 \neq 9$	8	30	1	9	11	19	2
③ $1 \neq 9$	8	30	1	9	11	19	2
④ $9 = 9$	8	30	1	9	11	19	2

## ● 검색을 실패하는 경우

6을 검색하는 경우

① $8 \neq 6$	8	30	1	9	11	19	2
② $30 \neq 6$	8	30	1	9	11	19	2
③ $1 \neq 6$	8	30	1	9	11	19	2
④ $9 \neq 6$	8	30	1	9	11	19	2
⑤ $11 \neq 6$	8	30	1	9	11	19	2
⑥ $19 \neq 6$	8	30	1	9	11	19	2
⑦ $2 \neq 6$	8	30	1	9	11	19	2

# 순차 검색 알고리즘 [1/2]

```
sequentialSearch1(a[], n, key)
```

```
  i ← 0;
```

```
  while (i < n and a[i] ≠ key) do {
```

```
    i ← i + 1;
```

```
  }
```

```
  if (i < n) then return i;
```

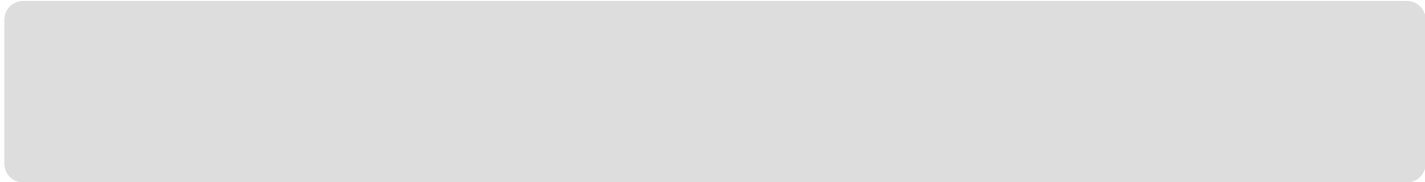
```
  else return -1;
```

```
end sequentialSearch1()
```



# 순차 검색 알고리즘 [2/2]

- 비교횟수 – 찾고자 하는 원소의 위치에 따라 결정
  - 찾는 원소가 첫 번째 원소라면 비교횟수는 1번,  
두 번째 원소라면 비교횟수는 2번,  
세 번째 원소라면 비교횟수는 3번,  
찾는 원소가  $i$ 번째 원소이면  $i$ 번, ...
  - 정렬되지 않은 원소에서의 순차 검색의 평균 비교 횟수



- 평균 시간 복잡도



# 순차 검색(정렬)

## ● 정렬된 자료에 대한 순차 검색 방법

- 첫 번째 원소부터 순서대로 키 값이 일치하는 원소를 찾는다.
- 키 값이 일치하는 원소를 찾으면 → 몇 번째 원소인지 반환
- 원소의 키 값이 찾는 값보다 크면 → 검색 종료

## ● 정렬되어있는 자료에 대한 순차 검색

9를 검색하는 경우

① $1 < 9$	1	2	8	9	11	19	29
② $2 < 9$	1	2	8	9	11	19	29
③ $8 < 9$	1	2	8	9	11	19	29
④ $9 = 9$	1	2	8	9	11	19	29

(a) 검색 성공의 경우

6을 검색하는 경우

① $1 < 6$	1	2	8	9	11	19	29
② $2 < 6$	1	2	8	9	11	19	29
③ $8 > 6$	1	2	8	9	11	19	29

→ 검색 종료

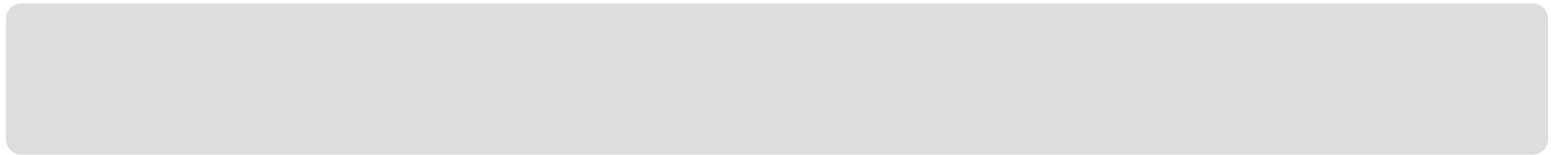
(b) 검색 실패의 경우

# 순차 검색 알고리즘(정렬) [1/2]

```
sequentialSearch2(a[], n, key)
    i ← 0;
    while (a[i] < key) do {
        i ← i + 1;
    }
    if (a[i] = key) then return i;
    else return -1;
end sequentialSearch2()
```

# 순차 검색 알고리즘(정렬) [2/2]

- 비교횟수 – 찾고자 하는 원소의 위치에 따라 결정
  - 검색 실패의 경우에 평균 비교 횟수가 반으로 줄어든다.
  - 정렬되어있는 원소에서의 순차 검색의 평균 비교 횟수



- 평균 시간 복잡도



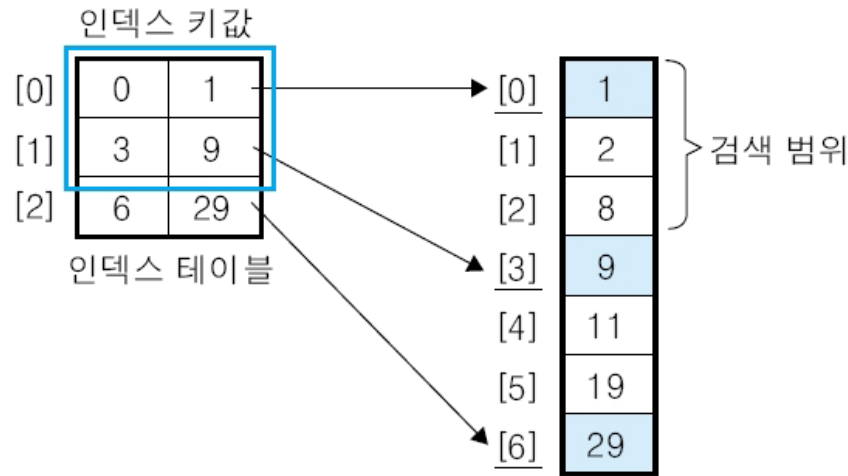
# 색인 순차 검색 [1/3]

- 색인 순차 검색(index sequential search, 사전 검색)
  - 정렬되어있는 자료에 대한 인덱스 테이블(index table)을 추가로 사용하여 탐색 효율을 높인 검색 방법
- 인덱스 테이블
  - 배열에 정렬되어있는 자료 중에서 일정한 간격으로 떨어져있는 원소들을 저장한 테이블
    - ◆ 자료가 저장된 배열 크기가  $n$ 이고 인덱스 테이블 크기가  $m$ 일 때, 배열에서  $n/m$  간격으로 떨어져있는 원소와 그의 인덱스를 인덱스 테이블에 저장
- 검색 방법
  - $\text{indexTable}[i].\text{key} \leq \text{key} < \text{indexTable}[i+1].\text{key}$ 를 만족하는  $i$ 를 찾아서 배열의 어느 범위에 있는지를 먼저 알아낸 후에 해당 범위에 대해서만 순차 검색 수행

# 색인 순차 검색 [2/3]

## ● 색인 순차 검색 예

- 검색 대상 자료 : {1, 2, 8, 9, 11, 19, 29}
- 크기가 3인 인덱스 테이블 작성
- 인덱스 테이블에서 탐색 키를 검색하여 검색 범위를 확인하고, 해당 범위에 대해서만 순차 검색 실행



# 색인 순차 검색 [3/3]

## ● 색인 순차 검색의 성능

### ■ 인덱스 테이블의 크기에 따라 결정

- ◆ 인덱스 테이블의 크기를 줄이면 배열의 인덱스를 저장하는 간격이 커지므로 배열에서 검색해야하는 범위도 커진다.
- ◆ 인덱스 테이블의 크기를 늘리면 배열의 인덱스를 저장하는 간격이 작아지므로 배열에서 검색해야하는 범위는 작아지지만 인덱스 테이블을 검색하는 시간이 늘어난다.

## ● 색인 순차 검색의 시간 복잡도

- ### ■ 배열 크기 $n$ , 인덱스 테이블 크기 $m$

# 이진 검색

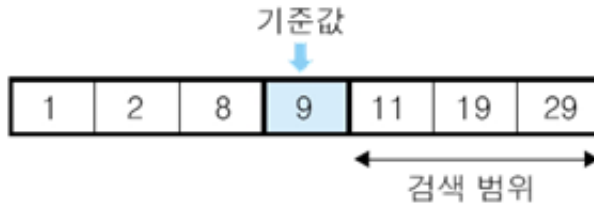
- 이진 검색(binary search, 보간 검색, interpolation search)
  - 정렬되어있는 자료에 대해서 수행하는 검색 방법
  - 자료의 가운데에 있는 항목을 키 값과 비교하여 다음 검색 위치를 결정하여 검색을 계속하는 방법
    - ◆ 찾는 키 값 > 원소의 키 값 : 오른쪽 부분에 대해서 검색 실행
    - ◆ 찾는 키 값 < 원소의 키 값 : 왼쪽 부분에 대해서 검색 실행
  - 키를 찾을 때까지 검색 범위를 반으로 줄여가면서 이진 검색을 순환적으로 반복 수행함으로써 빠르게 검색
    - ◆ 정복 기법을 이용한 검색 방법
    - ◆ 검색 범위를 반으로 분할하는 작업과 검색 작업을 반복 수행



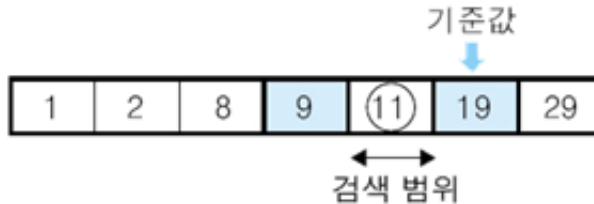
# 이진 검색의 예

11을 검색하는 경우

①  $11 > 9 \rightarrow$  오른쪽 검색



②  $11 < 19 \rightarrow$  왼쪽 검색

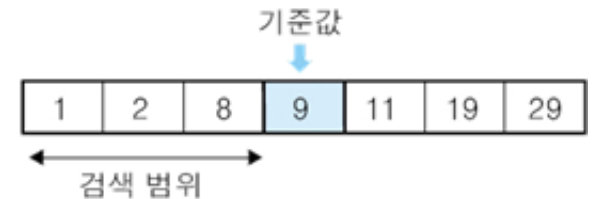


③  $11 = 11 \rightarrow$  검색 성공

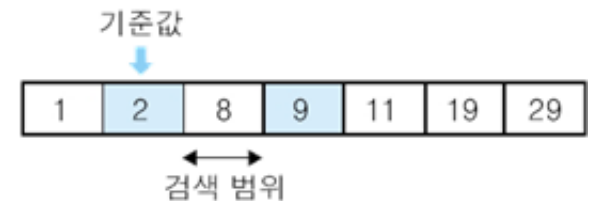
(a) 검색 성공의 경우

6을 검색하는 경우

①  $6 < 9 \rightarrow$  왼쪽 검색



②  $6 > 2 \rightarrow$  오른쪽 검색



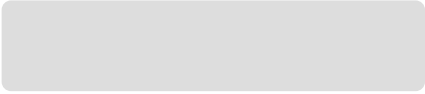
③  $6 \neq 8 \rightarrow$  검색 종료

(b) 검색 실패의 경우

# 이진 검색 알고리즘 [1/2]

```
binarySearch(a[], low, high, key)  
  mid  $\leftarrow$  (low + high) / 2;  
  if (key = a[mid]) then return i;  
  else if (key < a[mid]) then  
    binarySearch(a[], low, mid - 1, key);  
  else if (key > a[mid]) then  
    binarySearch(a[], mid + 1, high, key);  
  else return -1;  
end binarySearch()
```

# 이진 검색 알고리즘 [2/2]

- 삽입이나 삭제가 발생했을 경우에 항상 배열을 정렬 상태로 유지하는 추가적인 작업 필요
- 시간 복잡도 

# 이진 트리 검색

## ● 이진 트리 검색(Binary Tree Search)

- 이진 탐색 트리(Binary Search Tree)를 사용한 검색 방법
- 원소의 삽입 또는 삭제 연산에 대해 항상 이진 탐색 트리를 재구성하는 작업 필요

## ● 이진 탐색 트리의 정의

1. 모든 원소는 서로 다른 유일한 키를 갖는다.
2. 왼쪽 서브트리에 있는 원소의 키들은 그 루트의 키보다 작다.
3. 오른쪽 서브트리에 있는 원소의 키들은 그 루트의 키보다 크다.
4. 왼쪽 서브트리와 오른쪽 서브트리도 이진 탐색 트리이다.

# 이진 트리 검색 알고리즘

```
bstSearch(bst, x)
```

```
  p ← bst;
```

```
  if (p = null) then return null;
```

```
  if (x = p.key) then return p;
```

```
  if (x < p.key) then return searchBST(p.left, x);
```

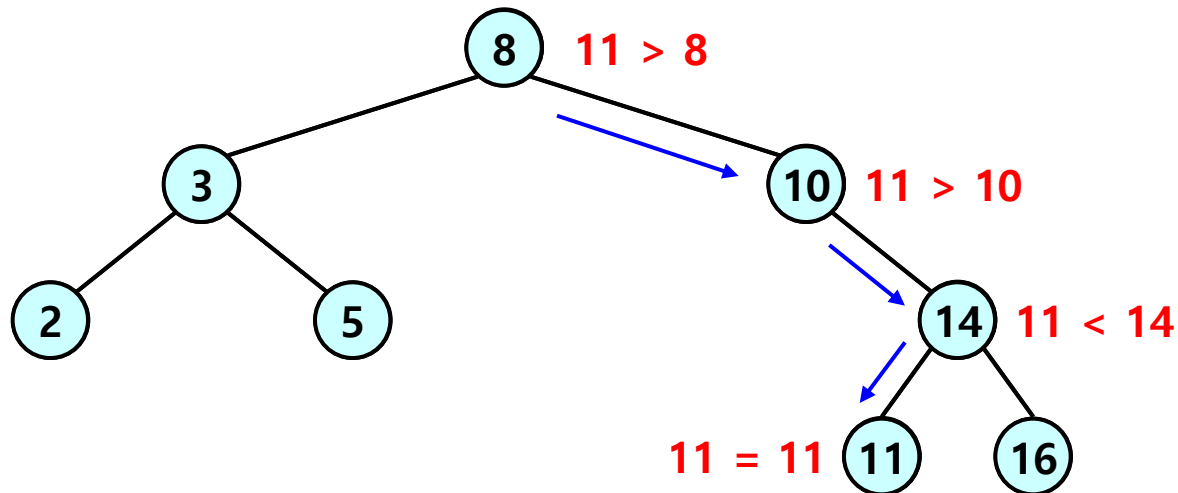
```
  if (x > p.key) then return searchBST(p.right, x);
```

```
end searchBST()
```

# 이진 트리 검색의 예

## ● 탐색 연산 예) 원소 11 탐색하기

1. 찾는 키 값 11을 루트노드의 키 값 8과 비교  
(찾는 키 값 11 > 노드의 키 값 8) 이므로 오른쪽 서브트리를 탐색
2. (찾는 키 값 11 > 노드의 키 값 10) 이므로 오른쪽 서브트리를 탐색
3. (찾는 키 값 11 < 노드의 키 값 14) 이므로 왼쪽 서브트리를 탐색
4. (찾는 키 값 11 = 노드의 키 값 11) 이므로 탐색 성공!(연산 종료)



# 해싱 [1/2]

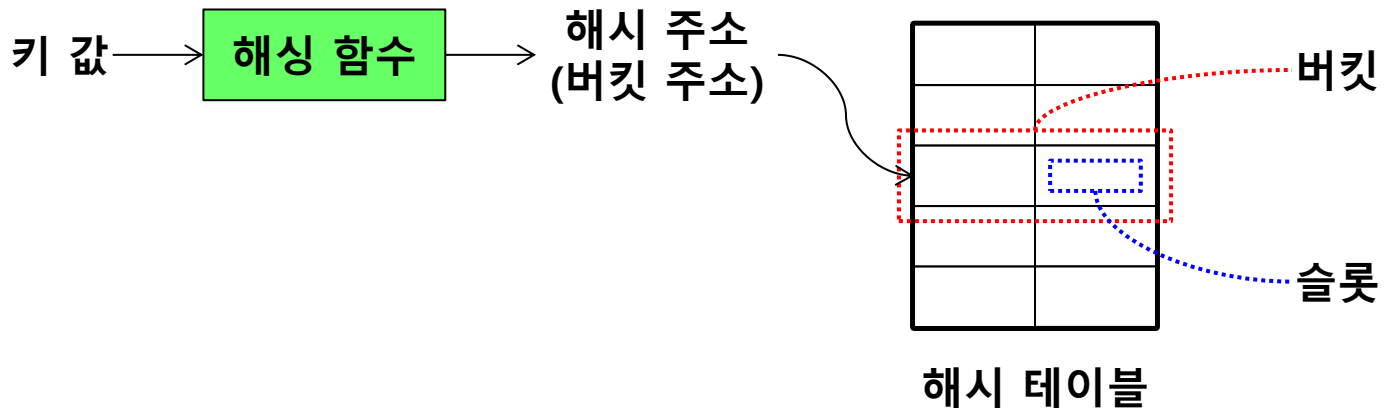
## ● 해싱(hashing)

- 산술적인 연산을 이용하여 키가 있는 위치를 계산하여 바로 찾아가는 계산 검색 방식
- 해싱 함수(hashing function)
  - ◆ 식별자(키 값)를 원소의 위치로 변환하는 함수
- 해시 테이블(hash table)
  - ◆ 해싱 함수에 의해 계산된 주소에 식별자를 저장하는 표
  - ◆ 해시 테이블의 하나의 주소를 버킷이라 부른다.
  - ◆ 하나의 버킷에는 복수개의 슬롯이 존재할 수 있다.

# 해싱 [2/2]

## ● 검색 방법

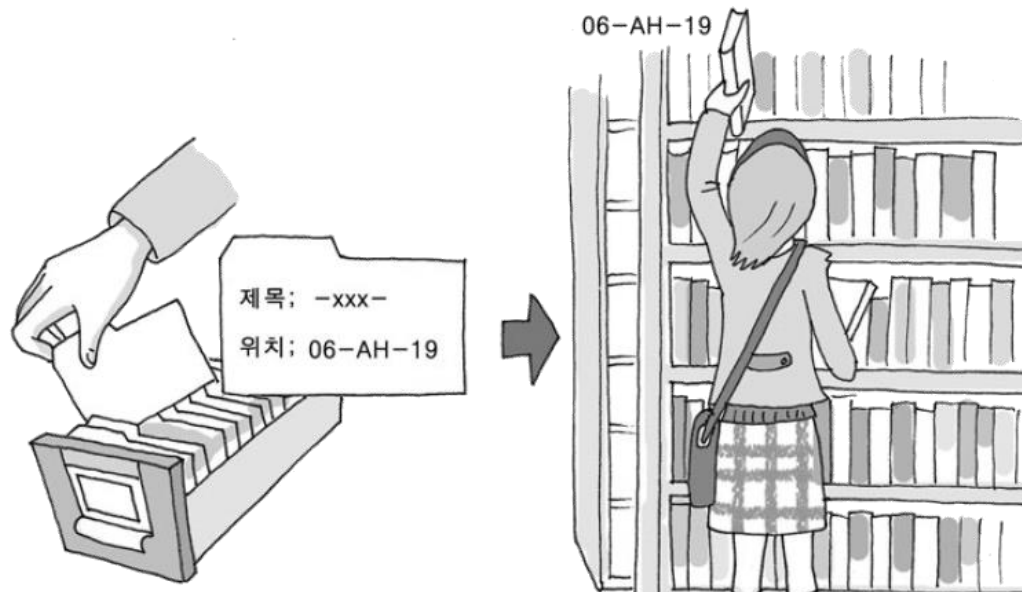
- 식별자에 대해서 해싱 함수를 계산하여 버킷 주소를 구하고, 구한 주소에 해당하는 버킷으로 바로 이동
- 해당 주소에 찾는 항목이 있으면 검색 성공, 없으면 검색 실패





# 해싱의 예

- 도서관에서의 도서 검색
  - 도서 이름으로 위치번호를 찾아 도서 찾기



# 해싱 테이블 작성의 예

- 알파벳 단어를 저장하는 해싱 테이블
  - 버킷 크기  $b = 26$ , 슬롯 크기  $s = 2$
  - 해싱 함수는 단어의 첫글자 a-z를 0-25로 대응

**word = ( 'acos', 'define', 'float', 'exp', 'char',  
'atan', 'ceil', 'floor', 'clock', 'ctime' )**

버킷#	슬롯0	슬롯1
0		
1		
2		
3		
4		
5		
6		
...		
25		

# 해싱 용어 정리 [1/2]

## ● 충돌(collision)

- 서로 다른 식별자에 대해서 동일한 버킷 주소가 계산된 경우
- 충돌이 발생한 경우 빈 슬롯에 동거자 관계로 키 값 저장

## ● 동거자(synonym)

- 서로 다른 값을 가지지만 해싱 함수에 의해서 같은 버킷에 저장된 식별자들

## ● 오버플로우(overflow)

- 버킷에 비어있는 슬롯이 없는 포화 버킷 상태에서 충돌이 발생하여 해당 버킷에 키 값을 저장할 수 없는 상태

버킷#	슬롯0	슬롯1
0	<b>acos</b>	<b>atan</b>
1		
2	<b>char</b>	<b>ceil</b>
3	<b>define</b>	
4	<b>exp</b>	
5	<b>float</b>	<b>floor</b>
6		
...		
25		

# 해싱 용어 정리 [2/2]

## ● 식별자 밀도(identifier density)

- 사용 가능한 식별자 값들 중에서 현재 해시 테이블에 저장되어서 실제 사용되고 있는 식별자 값의 개수 정도

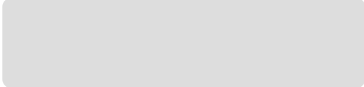
$$\text{식별자 밀도} = \frac{\text{테이블에서 실제 사용중인 식별자 개수 } n}{\text{사용 가능한 식별자 개수 } T}$$

## ● 적재 밀도(loading density)

- 해시 테이블에 저장 가능한 식별자 값의 개수 중에서 현재 해시 테이블에 저장되어서 실제 사용되고 있는 식별자 값의 개수 정도

$$\text{적재 밀도} = \frac{\text{테이블에서 실제 사용중인 식별자 개수 } n}{\text{슬롯 개수 } s \times \text{버킷 개수 } b}$$

# 해싱의 시간 복잡도

- 해싱의 시간 복잡도 
- 해싱 함수 계산 시간 + 버킷 탐색 시간
- 해싱 함수 계산 시간
  - ◆ 식별자 개수  $n$ 과 무관하게 계산됨
- 버킷 탐색 시간
  - ◆ 버킷의 개수는 충분히 작아 순차 탐색하더라도 식별자 개수  $n$ 과 무관
- 단지, 충돌에 대한 오버헤드는 피할 수 없다.

# 해싱 함수(Hasing Function)

## ● 해싱 함수의 조건

- 해싱 함수는 계산이 쉬워야 한다.
  - ◆ 비교 검색 방법에서 검색하는 시간보다 빨라야 의미가 있다.
- 해싱 함수는 충돌이 적어야 한다.
  - ◆ 오버플로우 발생 확률이 낮아야 좋은 해싱 함수이다.
- 해시 테이블에 고르게 분포할 수 있도록 주소를 만들어야 한다.
  - ◆ 식별자의 모든 부분이 결과에 영향을 미쳐야 한다.
  - ◆ 첫 글자만 사용 → 특정 글자의 영향력이 큼

# 해싱 함수의 종류 [1/6]

## ● 진법 변환 함수

- 식별자 값이 10진수가 아닌 다른 진수일 때, 10진수로 변환하고 해시 테이블 주소로 필요한 자릿수만큼만 하위자리의 수를 사용하는 방법

## ● 비트 추출 함수

- 해시 테이블의 크기가  $2^k$ 일 때 키 값을 이진 비트로 놓고 임의의 위치에 있는 비트들을 추출하여 주소로 사용하는 방법
- 이 방법에서는 충돌이 발생할 가능성이 많으므로 테이블의 일부에 주소가 편중되지 않도록 키 값들의 비트들을 미리 분석하여 사용해야 한다.

# 해싱 함수의 종류 [2/6]

## ● 중간 제공 함수

- 식별자 값을 제공한 값에서 중간에 적당한 비트를 주소로 사용
- 제공한 값의 중간 비트들은 대개 키의 모든 값과 관련이 있기 때문에 서로 다른 키 값은 서로 다른 중간 제공 함수 값을 갖는다.
- 예) 키 값 00110101 10100111에 대한 해시 주소 구하기

	00110101	10100111
X	<u>00110101</u>	<u>10100111</u>
00001011	00111110	10010010 11110001



# 해싱 함수의 종류 [3/6]

## ● 제산 함수

- 나머지 연산자를 사용하는 방법
- 제산함수  $h(k) = k \bmod M$ 
  - ◆ 식별자 값  $k$ 를 해시 테이블의 크기  $M$ 으로 나눈 나머지 값  $0 \sim (M-1)$ 을 해시 주소로 사용
- 식별자를 나누는  $M$ 의 선택이 중요
  - ◆  $M$ 이 약수가 많은 수인 경우, 식별자 순열의 패턴에 따라 결과가 편중되기 쉽다. 예) 식별자가 모두 짝수
  - ◆ 충돌이 발생하지 않고 해시테이블 주소가 고르게 분포하도록 해시 테이블 크기  $M$ 은 소수(prime number)를 사용한다.

# 해싱 함수의 종류 [4/6]

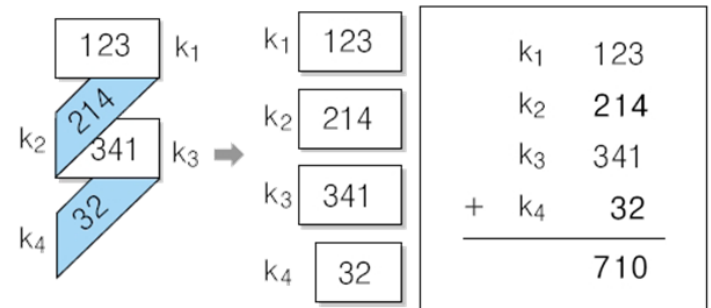
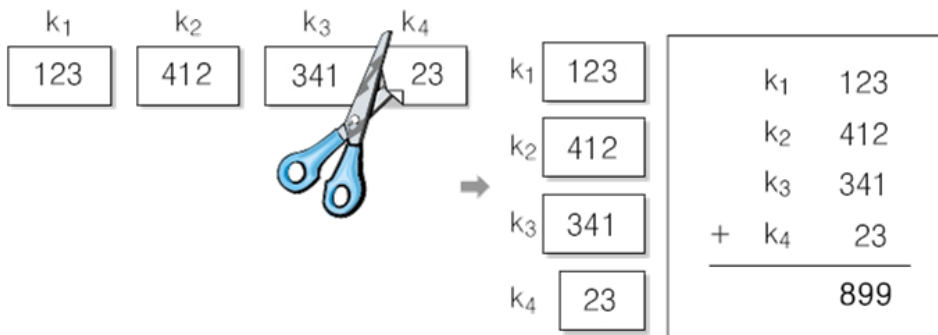
## ● 승산 함수

- 곱하기 연산을 사용하는 방법
- 식별자 값  $k$ 와 정해진 실수  $\alpha$ 를 곱한 결과에서 소수점 이하 부분( $0 \sim 1$ 미만의 값)을 테이블 크기  $M$ 과 곱하여 그 정수 값  $0 \sim (M-1)$ 을 해시 주소로 사용

# 해싱 함수의 종류 [5/6]

## ● 접지 함수(folding function)

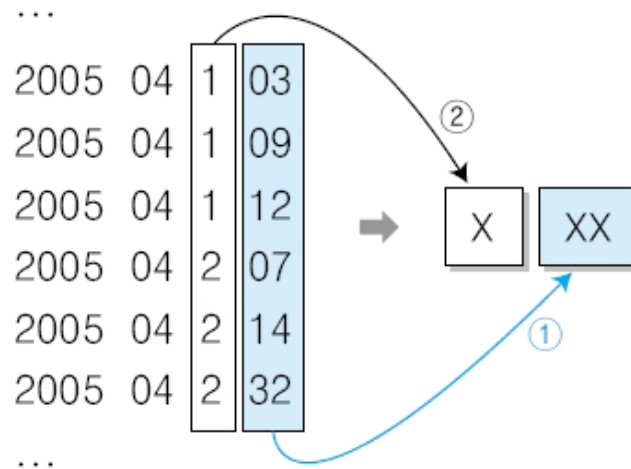
- 식별자를 여러 조각으로 나누고 연산하여 해시 주소 생성
  - ◆ 식별자의 범위가 해시 주소의 범위보다 매우 큰 경우 주로 사용
  - ◆ 식별자의 모든 범위에서 해시 주소에 영향을 미칠 수 있음
- 이동 접지 함수(shift folding)
  - ◆ 식별자 자리수를 분할하고 끝자리를 맞춰 더하는 방법
- 경계 접지 함수(folding at the boundary)
  - ◆ 분할된 각 경계를 기준으로 접으면서 순서를 바꾸어 더하는 방법



# 해싱 함수의 종류 [6/6]

## ● 숫자 분석 함수

- 식별자 값의 각 자릿수의 분포를 분석하여 해시 주소로 사용하는 방법
- 식별자 값을 적절한 진수로 변환한 후에 각 자릿수의 분포를 분석하여 가장 편중된 분산을 가진 자릿수는 생략하고, 가장 고르게 분포된 자릿수를 선택하여 해시 테이블 주소로 사용
- 예) 식별자가 학번이고 해시 테이블 주소의 자릿수가 3자리인 경우



# 오버플로우 처리방법

- 선형 개방 주소법 (선형 조사법, linear probing)
  - 해싱 함수로 구한 버킷에 빈 슬롯이 없어 오버플로우가 발생하면, 그 다음 버킷에 빈 슬롯이 있는지 조사한다.
    - ◆ 빈 슬롯이 있으면 - 키 값을 저장
    - ◆ 빈 슬롯이 없으면 - 다시 그 다음 버킷을 조사
    - ◆ 이 과정을 되풀이 하며 해시 테이블 내에 비어있는 슬롯을 순차적으로 찾아서 사용하는 방식으로 오버플로우 문제를 처리

# 오버플로우 처리방법

- 선형 개방 주소 법을 이용한 오버플로우 처리 예
  - 해시 테이블의 크기  $M = 5$
  - 해시 함수는 제산함수 사용. 해시 함수  $h(k) = k \bmod 5$
  - 저장할 식별자 값 ( 45, 9, 10, 96, 25 )

- ① 45 저장  $h(45) = 45 \bmod 5 = 0 \Rightarrow$  버킷 0번에 45 저장
- ② 9 저장  $h(9) = 9 \bmod 5 = 4 \Rightarrow$  버킷 4번에 9 저장
- ③ 10 저장  $h(10) = 10 \bmod 5 = 0 \Rightarrow$  충돌 발생!  
 $\Rightarrow$  다음 버킷 중 비어있는 버킷 1에 10 저장
- ④ 96 저장  $h(96) = 96 \bmod 5 = 1 \Rightarrow$  충돌 발생!  
 $\Rightarrow$  다음 버킷 중 비어있는 버킷 2에 96 저장
- ⑤ 25 저장  $h(25) = 25 \bmod 5 = 0 \Rightarrow$  충돌 발생!  
 $\Rightarrow$  다음 버킷 중 비어있는 버킷 3에 25 저장

# 오버플로우 처리방법

- 선형 개방 주소 법을 이용한 오버플로우 처리 예
  - 해시 테이블의 크기  $M = 5$
  - 해시 함수는 제산함수 사용. 해시 함수  $h(k) = k \bmod 5$
  - 저장할 식별자 값 ( 45, 9, 10, 96, 25 )



# 오버플로우 처리방법

## ● 체이닝(chaining)

- 해시 테이블의 구조를 변경하여 각 버킷에 하나 이상의 값을 저장할 수 있도록 하는 방법
- 연결 리스트를 사용하여 동적으로 슬롯을 삽입하고 삭제
  - ◆ 각 버킷에 대한 헤드노드를 1차원 배열로 만들고  
각 버킷에 대한 헤드노드는 슬롯들을 연결 리스트로 가지고 있어서  
슬롯의 삽입이나 삭제 연산을 쉽게 수행할 수가 있다.
  - ◆ 버킷 내에서 원하는 슬롯을 검색하기 위해서는  
버킷의 연결 리스트를 선형 검색한다.



# 오버플로우 처리방법

- 체이닝을 이용한 오버플로우 처리 예
  - 해시 테이블의 크기  $M = 5$
  - 해시 함수는 제산함수 사용. 해시 함수  $h(k) = k \bmod 5$
  - 저장할 식별자 값 ( 45, 9, 10, 96, 25 )

- ① 45 저장  $h(45) = 45 \bmod 5 = 0 \Rightarrow$  버킷 0번에 노드를 삽입하고 45 저장
- ② 9 저장  $h(9) = 9 \bmod 5 = 4 \Rightarrow$  버킷 4번에 노드를 삽입하고 9 저장
- ③ 10 저장  $h(10) = 10 \bmod 5 = 0 \Rightarrow$  버킷 0번에 노드를 삽입하고 10 저장
- ④ 96 저장  $h(96) = 96 \bmod 5 = 1 \Rightarrow$  버킷 1번에 노드를 삽입하고 10 저장
- ⑤ 25 저장  $h(25) = 25 \bmod 5 = 0 \Rightarrow$  버킷 0번에 노드를 삽입하고 10 저장

# 오버플로우 처리방법

- 체이닝을 이용한 오버플로우 처리 예
  - 해시 테이블의 크기  $M = 5$
  - 해시 함수는 제산함수 사용. 해시 함수  $h(k) = k \bmod 5$
  - 저장할 식별자 값 ( 45, 9, 10, 96, 25 )

