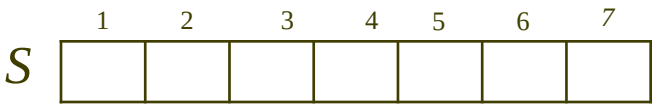
8장 계산복잡도: 검색 문제


홍길동의 전화번호를 더 빨리 찾는
방법이 없을까?

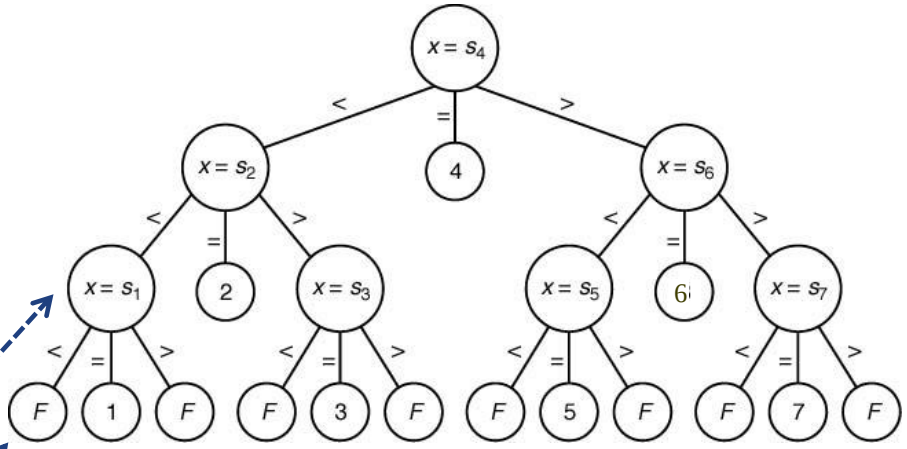
키를 비교함으로만 검색을 수행하는 경우의 하한

- **검색(Searching) 문제:**
  - ✓ n개의 키를 가진 배열 S와 어떤 키 x가 주어졌을 때, $x = S[i]$가 되는 첨자 i를 찾는다.
  - ✓ 만약 x가 배열 S에 없을 때는 오류로 처리한다.

- **이분(이진) 검색 알고리즘:**
  - ✓ 배열이 정렬이 되어 있는 경우 시간복잡도가 $W(n) = \lfloor \lg n \rfloor + 1$ 이 되어서, 매우 효율적인 알고리즘이라고 할 수 있다.
  - ✓ 이 보다 더 좋은(빠른) 알고리즘은 존재할까?
    [답] 키를 비교함으로만 검색을 수행하는 경우에는 이분 검색알고리즘 보다 더 좋은 알고리즘은 존재할 수 없다.



x?

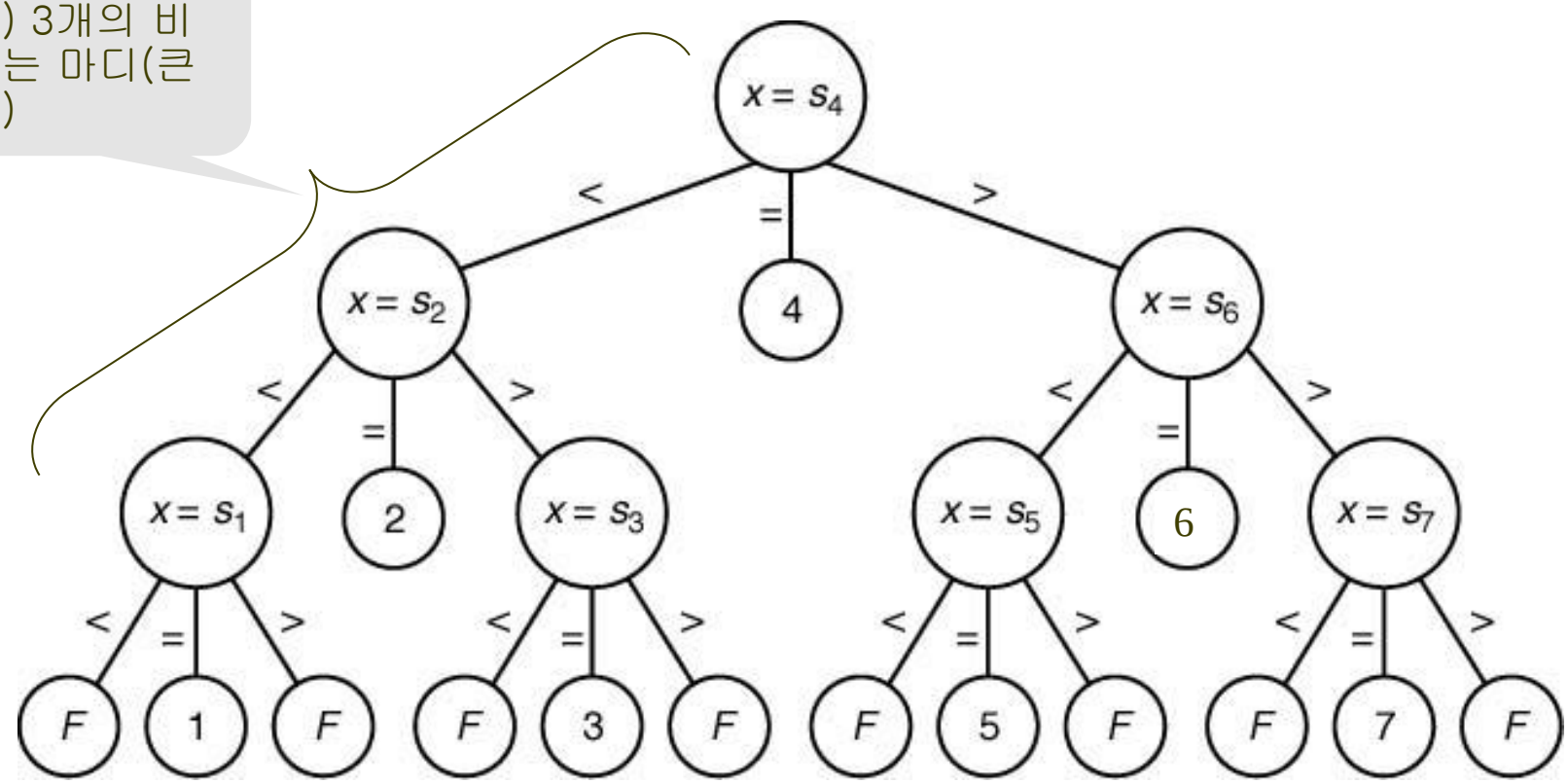


- 보기: 7개의 키를 검색하는 문제를 생각해 보자.

- 검색의 결정트리 (decision tree)
  - ✓ 큰 마디(내부마디) – 키와 각 아이템을 비교하는 마디
  - ✓ 작은 마디(잎마디) - 검색결과
  - ✓ 뿌리마디에서 잎마디까지의 경로는 검색하면서 비교하는 과정을 보여준다.

7개의 키를 검색하는 문제의 이진검색에 상응하는 결정트리

최대한(최악의 경우) 3개의 비교하는 마디(큰 마디)

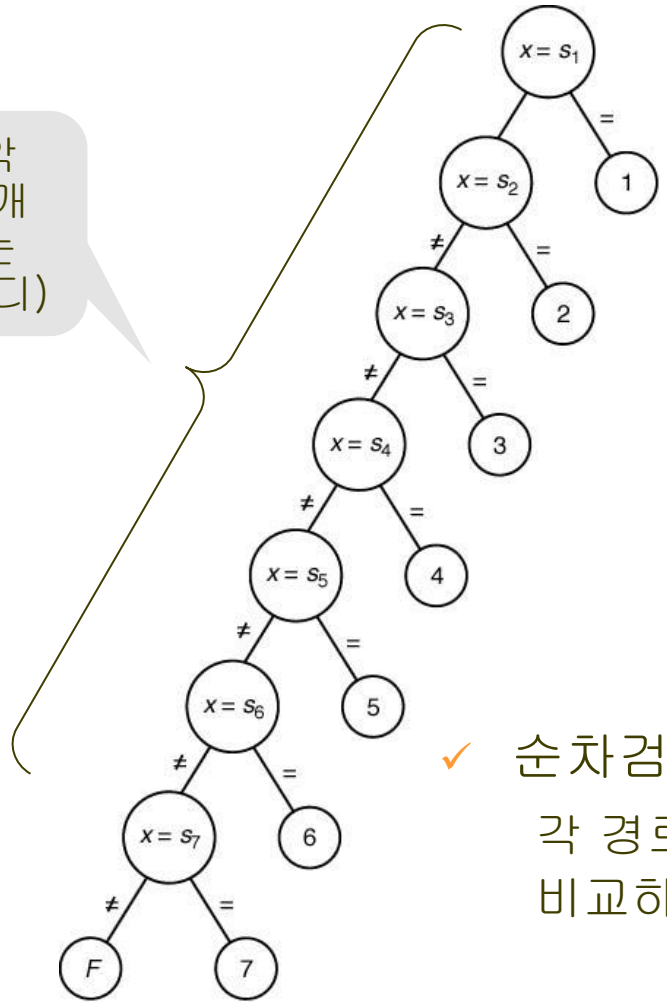


- ✓ 이진검색의 결정트리

  각 경로는 최대한(최악의 경우) 3개의 비교하는 마디(큰 마디)를 가진다.

순차검색의 결정트리

최대한(최악의 경우) 7개의 비교하는 마디(큰 마디)

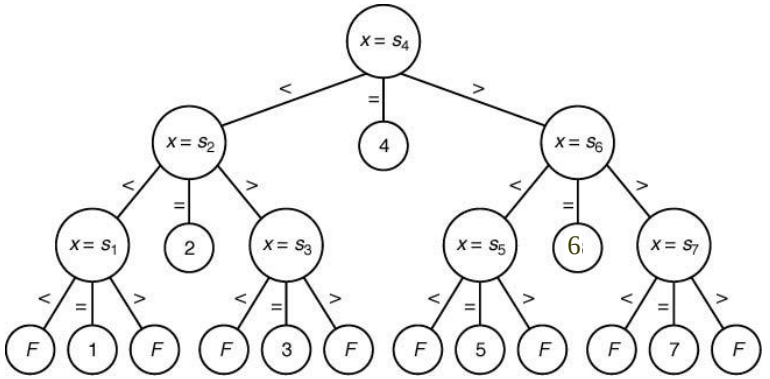


- ✓ 순차검색의 결정트리

  각 경로는 최대한(최악의 경우) 7개의 비교하는 마디(큰 마디)를 가진다.

- 결정트리는 x를 찾기 위해서 n개의 키를 검색하기에 유효하다(valid)
  - ✓ 배열 S에 가능한 모든 결과에 대해서 그 결과를 알려주는 뿌리마디에서 잎마디로의 경로가 있을 때
  - ✓ There must be paths for $x = s_i$ for $1 \le i \le n$ and a path that leads to failure.

- 모든 잎이 도달 가능한 결정 트리를 가지쳐 졌다(pruned)라고 한다.
- 값을 비교할 때의 결과는 3가지(<, =, >) ⇒ A node in decision tree has 3 children (<, =, >) .
- If we consider only the comparison nodes, it becomes a binary tree.
- 값을 비교하여 검색하는 것을 결정트리의 비교노드 만을 포함하는 트리로 표현 ⇒ 이진트리



7개의 데이터인 경우 최대 검색 회수의 하한은 무엇인가?

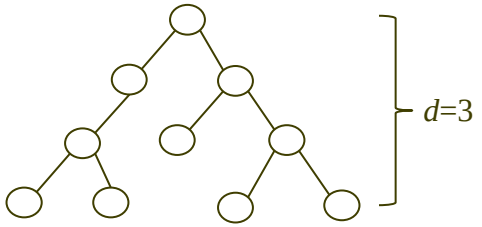
이분검색: 3회

순차검색: 7회

최악의 경우 하한 찾기

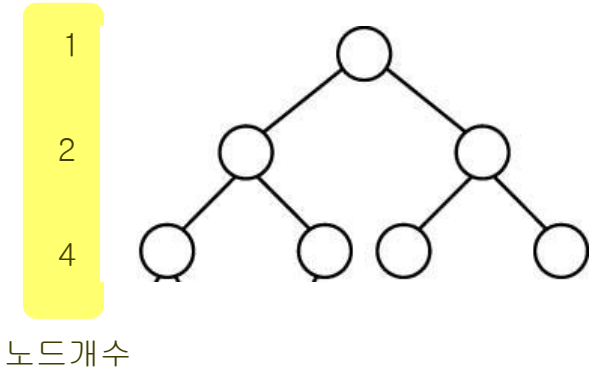
- 최악의 경우 비교하는 횟수는 결정트리의 뿌리마디에서 잎마디까지 가장 긴 경로 상의 비교마디의 수 인데, 이는 트리의 깊이(depth) + 1이다.

  → 트리의 하한을 확인하고자 함.



- **보조정리 1**: n이 이진트리의 마디의 개수이고, d가 깊이 이면, $d \geq \lfloor \lg n \rfloor$

  증명:



$$n \leq 1 + 2 + 2^2 + \cdots + 2^d$$

$$n \leq 2^{d+1} - 1$$

$$n < 2^{d+1}$$

$$\lg n < d + 1$$

$$\lfloor \lg n \rfloor \leq d$$

complete(perfect) binary tree인 경우가 같은 트리의 깊이인 경우 가장 많은 노드를 가짐

최악의 경우 하한 찾기

- **보조정리 2**: n개의 다른 키 중에서 어떤 키 x를 찾는 가지친 유효 결정트리 (pruned, valid decision tree)는 비교하는 마디가 최소한 n개 있다.

- **정리8.1**: n개의 다른 키가 있는 배열에서 어떤 키 x를 찾는 알고리즘은 키를 비교하는 횟수를 기준으로 했을 때 최악의 경우 최소한 $\lfloor \lg n \rfloor + 1$만큼의 비교를 한다.

  증명: 그 알고리즘의 결정트리에서 비교하는 마디 만으로 이루어진 이진트리를 보면, 최악의 경우 비교횟수는 이진트리의 뿌리마디에서부터 잎마디까지의 모든 경로 가운데 가장 긴 경로의 마디 수가 된다. 그 수는 이진트리의 깊이 + 1이 된다. 그런데 보조정리 2에서 이 이진트리의 마디 수는 n이라 하였으므로, 그 이진트리의 깊이는 $\lfloor \lg n \rfloor$보다 크거나 같다. 따라서, 최소한 $\lfloor \lg n \rfloor + 1$만큼의 비교를 한다.

- ✓ 검색 문제의 속성 상 최소한 $\lfloor \lg n \rfloor + 1$만큼의 비교를 한다.
- ✓ 그런데, 이진 검색의 복잡도가 $\lfloor \lg n \rfloor + 1$ 이다.
- ✓ 따라서, 이진 검색 방법이 검색문제의 최선의 해결 알고리즘이다.

conclusion: Binary search is optimal.

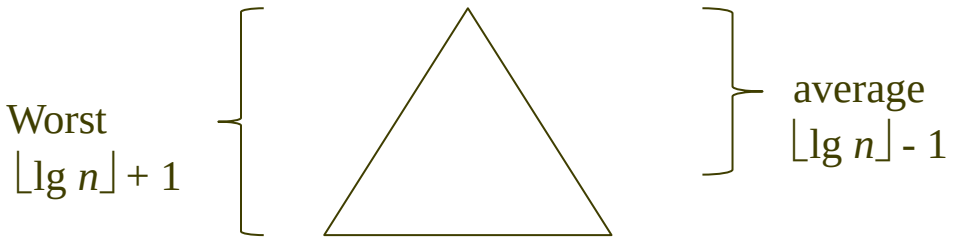
평균의 경우 하한 찾기

- **Theorem** 8.2

  Among deterministic algorithm that searches for a key x in an array of n distinct keys only by comparisons of keys, Binary Search is optimal in its average-case performance if we assume that x is in the array and that all array slots are equally probable. Therefore, under these assumptions, any such algorithm must on the average do at least approximately

  $$\lfloor \lg n \rfloor - 1 \text{ comparisons of keys.}$$

- 최악의 경우와 비슷한 방법으로 구해보면, 검색문제의 평균 하한은 $\lfloor \lg n \rfloor - 1$이 된다.



Binary Search

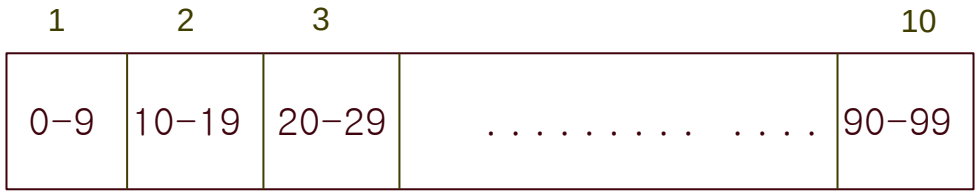
```
void binsearch(int n,
               const keytype S[],
               keytype x,
               index& location) {    // output
  index low, high, mid;

  low = 1; high = n;
  location = 0;
  while (low <= high && location == 0) {
    mid = (low + high) / 2;    // divided by an integer
    if (x == S[mid])
          location = mid;
    else if (x < S[mid])
          high = mid - 1;
    else
          low = mid + 1;
  }
}
```

보간 검색

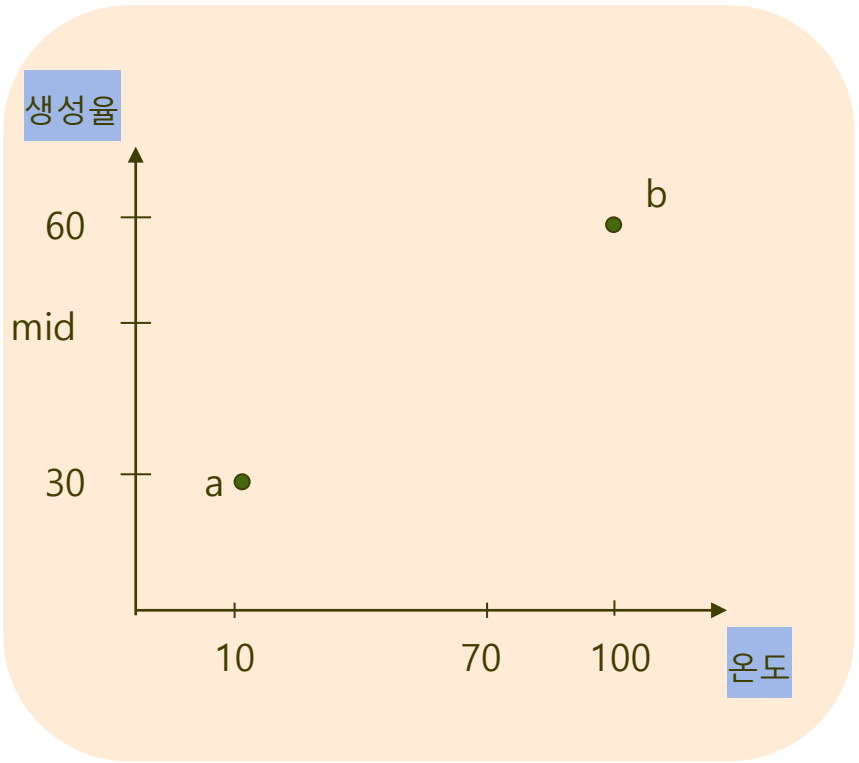
- 검색의 하한을 개선할 수 있는가?
  - ✓ 비교하는 이외에, 다른 추가적인 정보를 이용하여 검색할 수 있다면 가능하다.

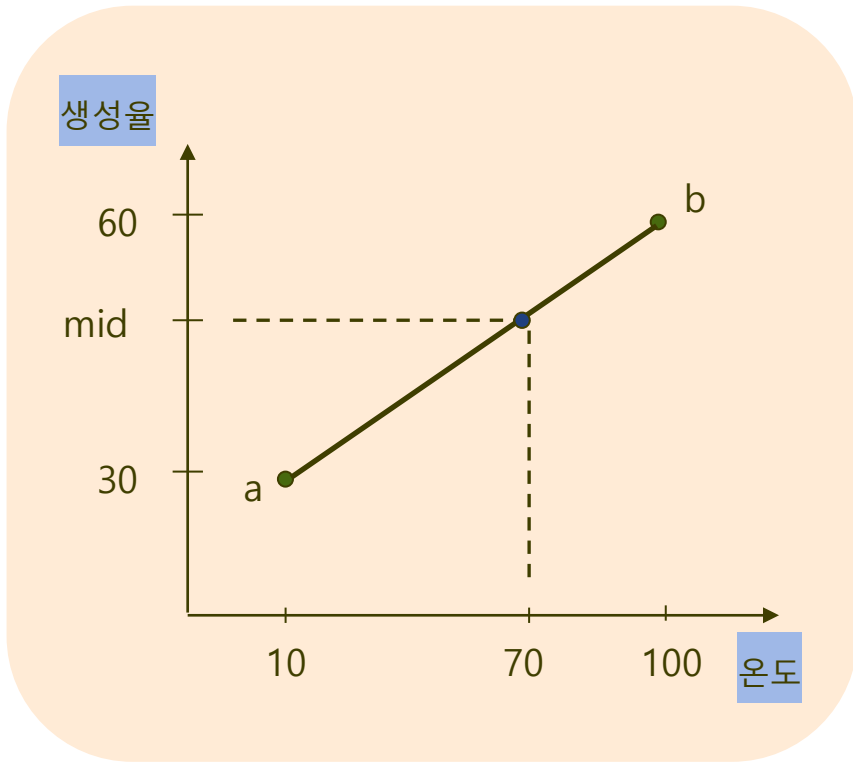
- 보기: 10개의 정수를 검색하는데, 여기서 첫 번째 정수는 0-9중에 하나이고, 두 번째 정수는 10-19중에 하나이고, 세 번째 20-29중에 하나이고, …, 열 번째 정수는 90-99중에 하나라고 하자. 이 경우 만약 검색 키 x가 0보다 작거나, 99보다 크면 바로 검색이 실패임을 알 수 있고, 그렇지 않으면 검색 키 x와 S[1+x div 10]과 비교해 보면 된다.
  - ✓ x=25는 S[1+25 div 10]=S[3]과 비교



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?





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

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*,

  - ✓ 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 평균적인 시간복잡도는 $A(n) \approx \lg(\lg n)$.
    (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 최악의 경우 시간복잡도는 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;
           }
}
```

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 평균 시간복잡도는 $A(n) \approx \Theta(\lg(\lg n))$ 이 되고, 최악의 경우는 $W(n) \approx \Theta((\lg n)^2)$

트리 구조를 사용한 동적검색

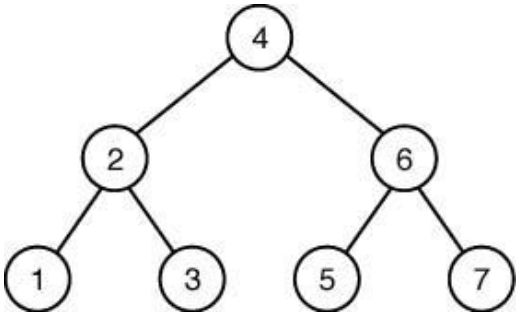
- **정적검색(static searching):** 데이터가 한꺼번에 저장되어 추후에 추가나 삭제가 이루어지지 않는 경우에 이루어 지는 검색, 즉 예를 들면 OS명령에 의한 검색.

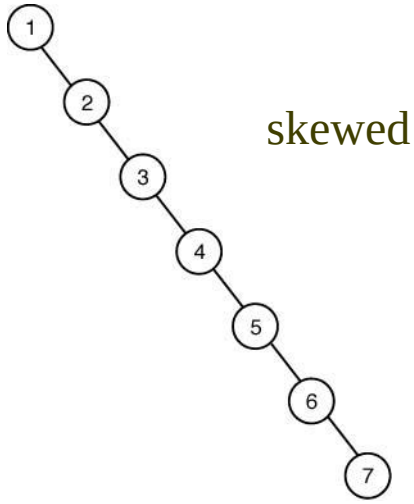
- **동적검색(dynamic searching):** 데이터가 수시로 추가 삭제되는 유동적인 경우, 예로서 비행기 예약 시스템.

- 배열 자료구조를 사용하면, 정적검색의 경우는 문제없이 이진검색이나 보간검색 알고리즘을 적용할 수 있지만, 동적검색의 경우는 이 알고리즘을 적용하기가 불가능하다. 따라서 트리 구조를 사용하여야 한다.

이진검색트리(binary search tree)

- **정의**: 이진검색트리는 다음 조건을 만족하는 이진트리이다.
  - ✓ 각 마디 마다 하나의 키가 할당되어 있다.
  - ✓ 어떤 마디 *n*의 왼쪽부분트리(left subtree)에 속한 모든 마디의 키는 그 마디 *n*의 키 보다 작거나 같다.
  - ✓ 어떤 마디 *n*의 오른쪽부분트리(right subtree)에 속한 모든 마디 *n*의 키 보다 크거나 같다.

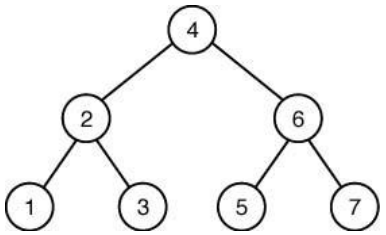




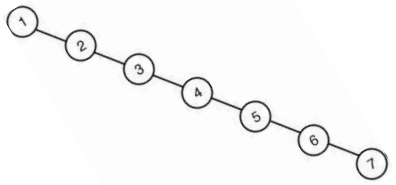
skewed tree

이진검색트리를 사용하는 이유

- 부모중간횡단(inorder traversal)을 하면 정렬된 순서로 아이템을 추출



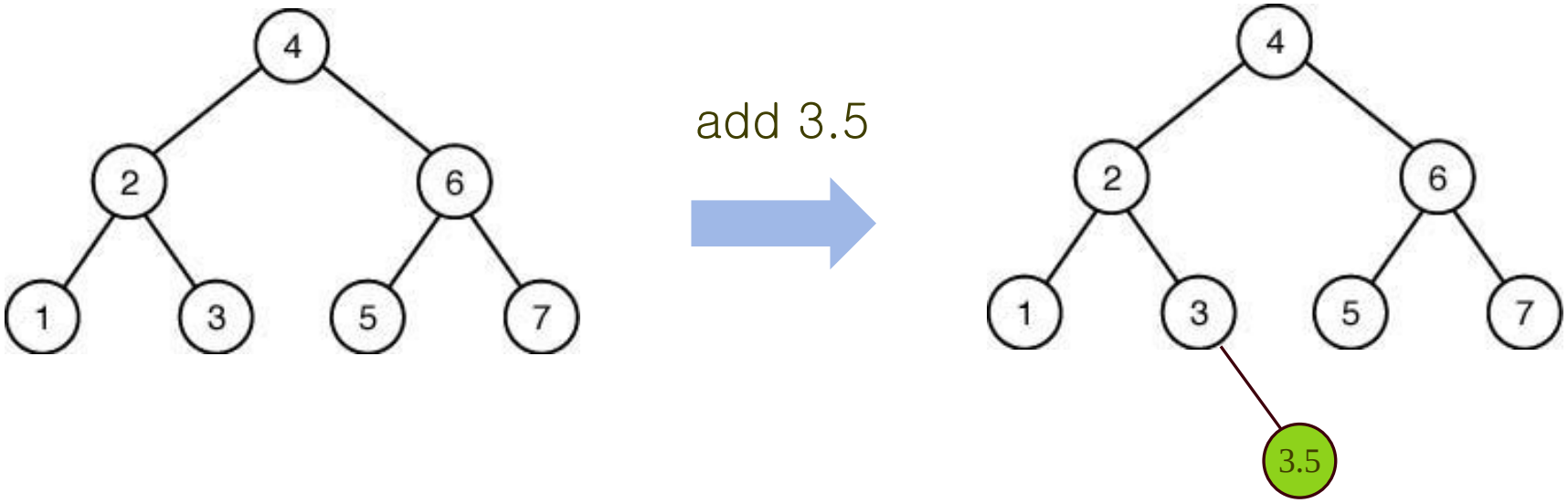
- 평균 검색시간을 짧게 유지:
  - ✓ 동적으로 트리에 아이템이 추가되고 삭제되므로 트리가 항상 균형(balance)을 유지한다는 보장이 없다.
  - ✓ 최악의 경우는 연결된 리스트(linked list)를 사용하는것(skewed tree)과 같은 효과.



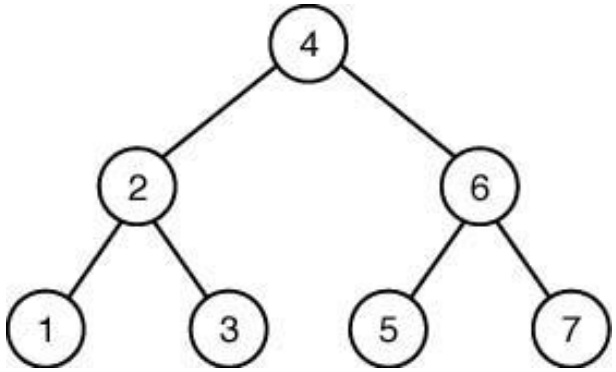
  - ✓ 랜덤(random)하게 아이템이 트리에 추가되는 경우는 대체로 트리가 균형을 유지 - 평균적으로 효율적인 검색시간을 기대할 수 있다.

Reason to Use Binary Search Tree

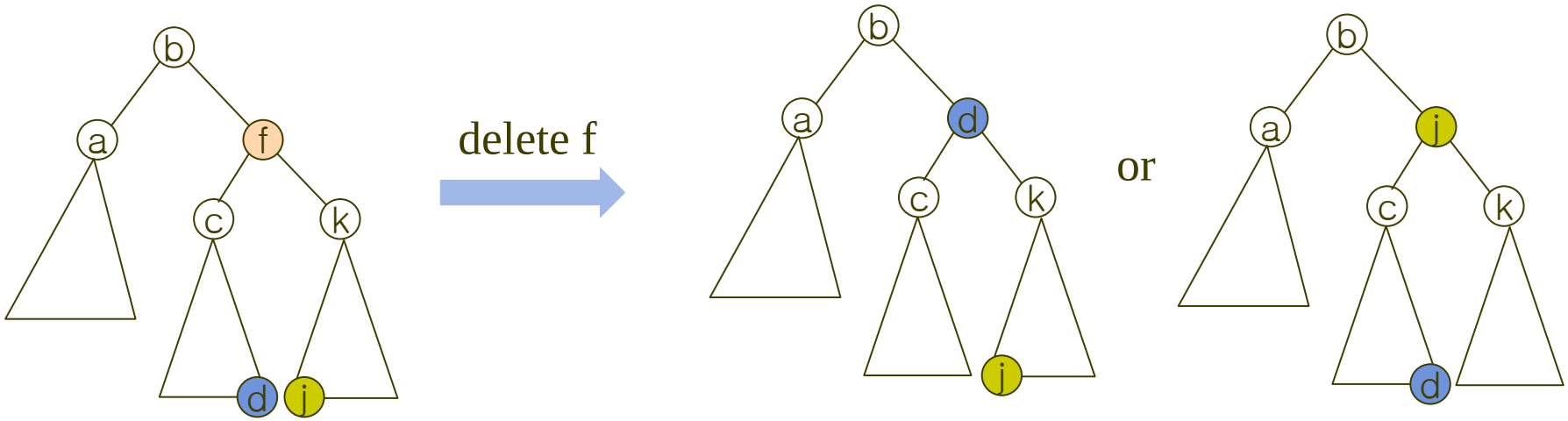
- We can add keys to and delete keys efficiently from the tree.
  - ✓ addition: trivial
  - ✓ deletion: use a simple operation



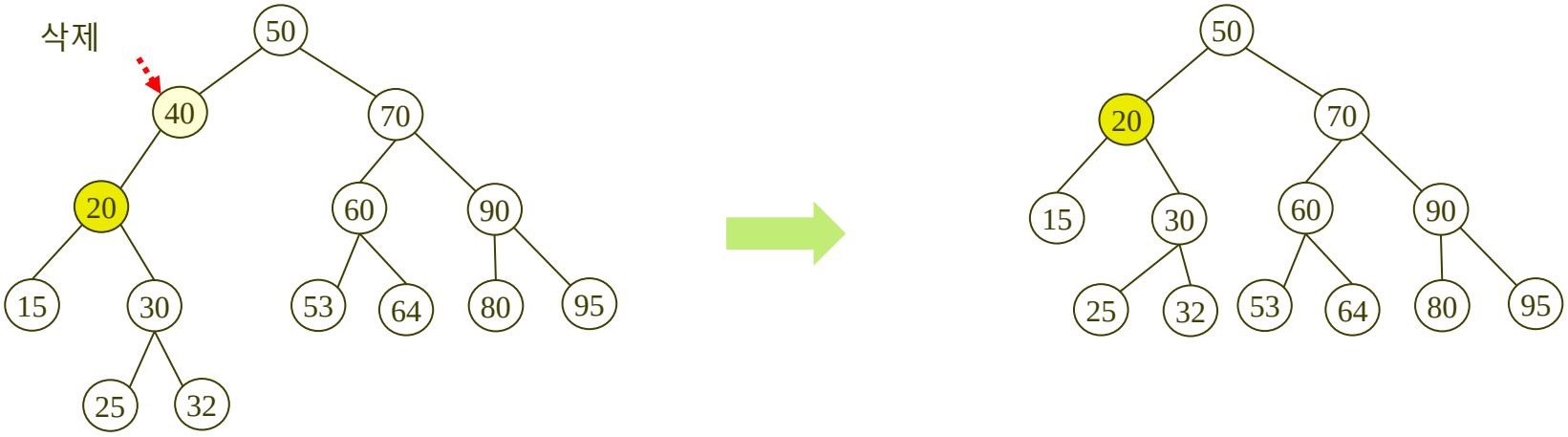
- 두 개의 자식노드가 있는 노드가 삭제될 경우



✓ delete 6.

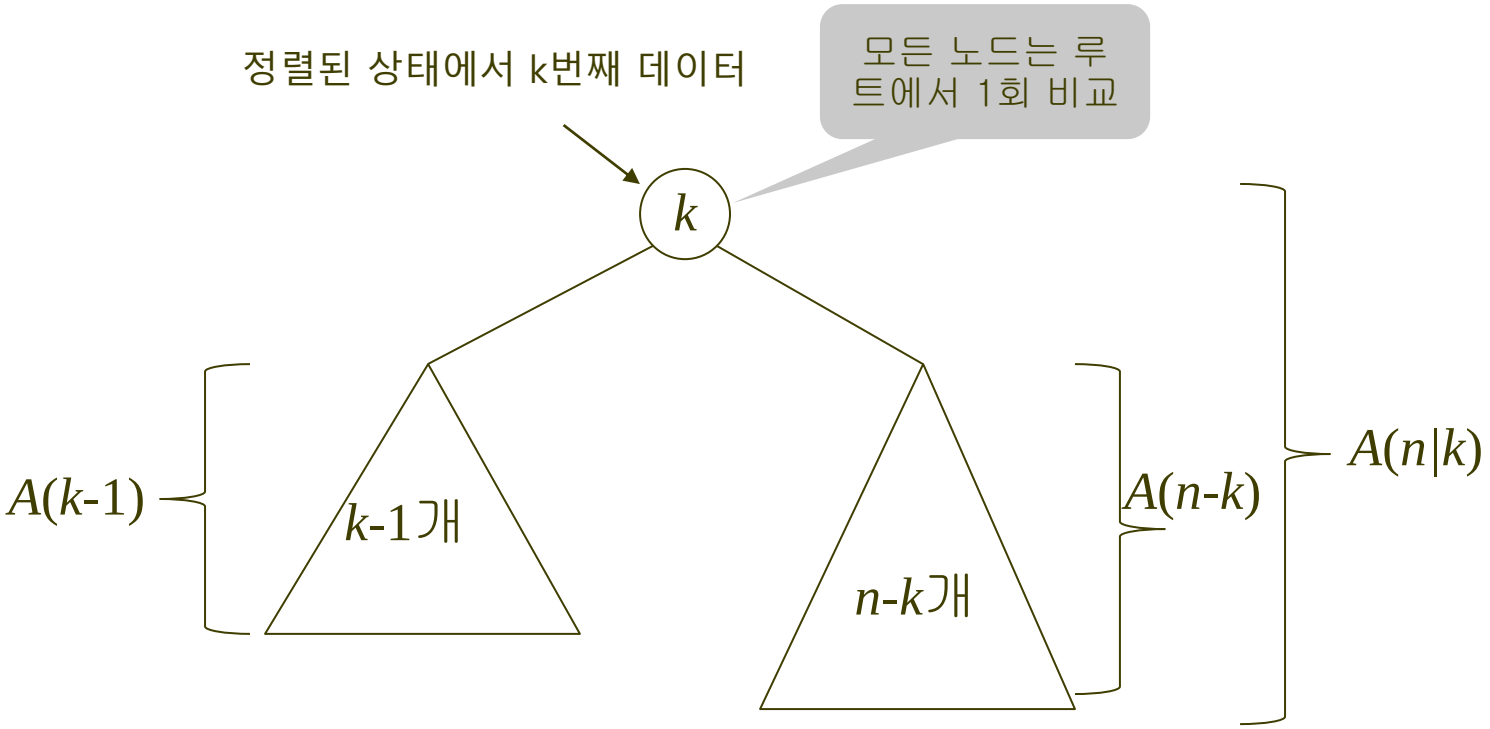


- 한 개의 자식노드가 있는 노드가 삭제될 경우



이진검색트리 검색의 분석

- **정리**: 검색하는 아이템 x가 n개의 아이템 중의 하나가 될 확률이 동일하다고 가정하면, 이진검색트리의 평균검색시간은 대략 $A(n) = 1.38 \lg n$이 된다.
  - ✓ 생성가능한 모든 이진검색트리에 대한 평균검색시간.
- **증명**
  - ✓ k번째로 작은 아이템이 뿌리마디에 위치하고 있는 n개의 아이템을 가진 이진검색트리 가정
  - ✓ 왼쪽 부분트리에 k-1개의 마디가 있고, 오른쪽 부분트리에 n-k개의 마디가 존재.
  - ✓ $A(k\text{-}1)$: 왼쪽부분트리를 검색하는 평균검색시간, $A(n\text{-}k)$:오른쪽 부분트리를 검색하는 평균검색시간
  - ✓ 그러면 x가 왼쪽 부분트리에 있을 확률은 $(k\text{-}1)/n$이 되고, 오른쪽 부분트리에 있을 확률은 $(n\text{-}k)/n$이 된다.
  - ✓ 이때 $A(n|k)$를 크기 n의 입력에 대하여 k번째로 작은 아이템이 뿌리마디에 위치하고 있는 이진검색트리에 대한 평균검색시간 이라고 하면,


정렬된 상태에서 k번째 데이터
모든 노드는 루트에서 1회 비교
k
A(k-1)
k-1개
n-k개
A(n-k)
A(n|k)

그러면

$$A(n \mid k) = A(k-1)\frac{k-1}{n} + A(n-k)\frac{n-k}{n} + 1$$

뿌리마디에서의 비교 $\sum_{i=1}^{n} 1 \times \frac{1}{n}$

$$A(n) = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{k-1}{n}A(k-1) + \frac{n-k}{n}A(n-k) + 1\right]$$

특정노드가 뿌리가 될 확률 1/n

여기서 $C(n) = nA(n)$으로 놓으면,

$$\frac{C(n)}{n} = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{k-1}{n}\frac{C(k-1)}{k-1} + \frac{n-k}{n}\frac{C(n-k)}{n-k} + 1\right]$$

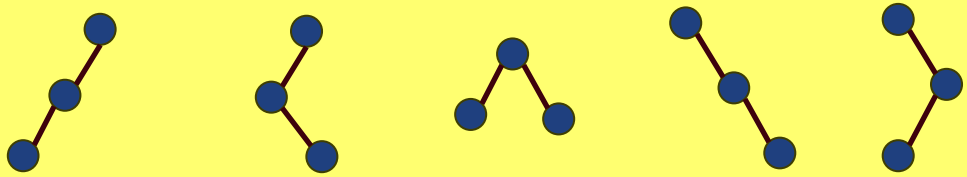
$$C(n) = \sum_{k=1}^{n}\left[\frac{C(k-1)}{n} + \frac{C(n-k)}{n} + 1\right] \qquad C(1) = 1, A(1) = 1$$

$$= \sum_{k=1}^{n}\frac{1}{n}[C(k-1) + C(n-k)] + n$$

- ✓ 이 재현식(recurrence)은 빠른 정렬의 평균의 경우의 시간복잡도와 거의 같다. 따라서 $C(n) \approx 1.38(n+1)\lg n$. 결과적으로 $A(n) \approx 1.38 \lg n$

- 최악의 경우의 시간복잡도는 $\Theta(n)$이므로 이 방법이 항상 효율적이라고 할 수는 없다.

[예] A(3)은 다음 이진검색트리들의 평균 비교 횟수



1+2+3=6　　1+2+3=6　　1+2+2=5　　1+2+3=6　　1+2+3=6

A(3)=(6+6+5+6+6)/(5*3)=29/15

$$
\begin{aligned}
A[1][n] &= A[1][k-1] && /* \text{왼쪽 부분트리에서 평균시간} \\
&+ p_1 + \ldots + p_{k-1} && /* \text{뿌리에서 비교하는데 드는 추가시간} \\
&+ p_k && /* \text{뿌리를 검색하는 평균시간} \\
&+ A[k+1][n] && /* \text{오른쪽 부분트리에서 평균시간} \\
&+ p_{k+1} + \ldots + p_n && /* \text{뿌리에서 비교하는데 드는 추가시간} \\
&= A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m
\end{aligned}
$$

✓ 동적계획법의 최적이진검색트리 구축 방법에서는

- 각 트리 내에 각 노드를 찾을 확률을 고려 – $A[i][i]=p_i$로 설정
- 하나의 고정된 모양의 이진검색트리에서 검색하는데 필요한 평균 시간 계산
- 따라서 다른 연쇄식이 구축된다.

$$
\begin{aligned}
A[i][j] &= \min_{i \le k \le j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \,, \; i < j \\
A[i][i] &= p_i
\end{aligned}
$$

✓ 본 장에서는 가능한 모든 이진검색트리 형태에 대한 평균 시간 계산

$$
A(n) = \frac{1}{n} \sum_{k=1}^{n} \left[\frac{k-1}{n} A(k-1) + \frac{n-k}{n} A(n-k) + 1 \right]
$$

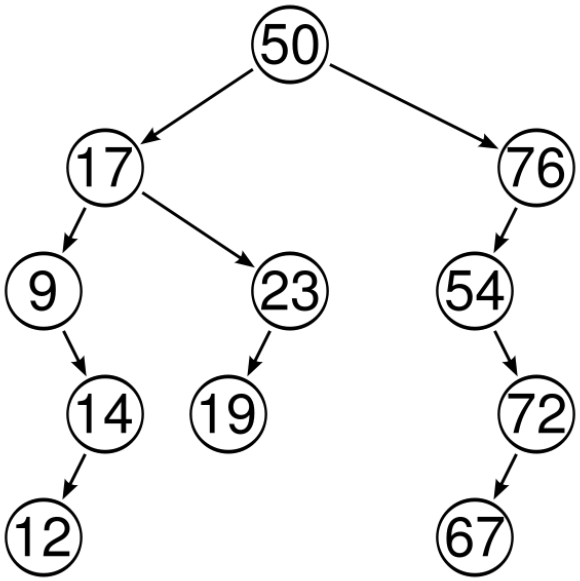
note A(1)=1

검색시간 향상을 위한 트리구조

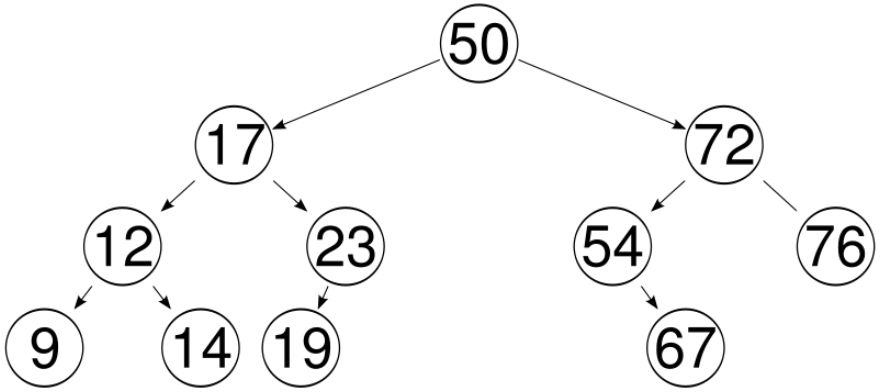
- 항상 균형을 유지하는 이진트리 활용

- **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 $\Theta(\lg n)$

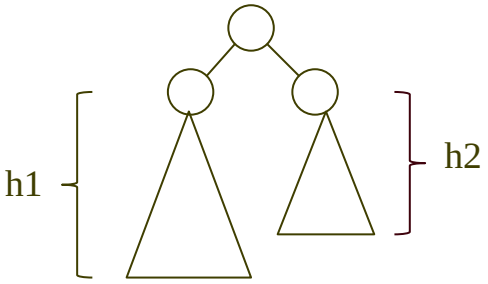
- **AVL 트리**: 좌,우subtree의 높이의 차가 최대 1인 이진탐색트리
- 러시아의 두 수학자인 G.M. Adelson-Velsky 와 d E.M. Landi, 1962년



일반적인 이진탐색트리



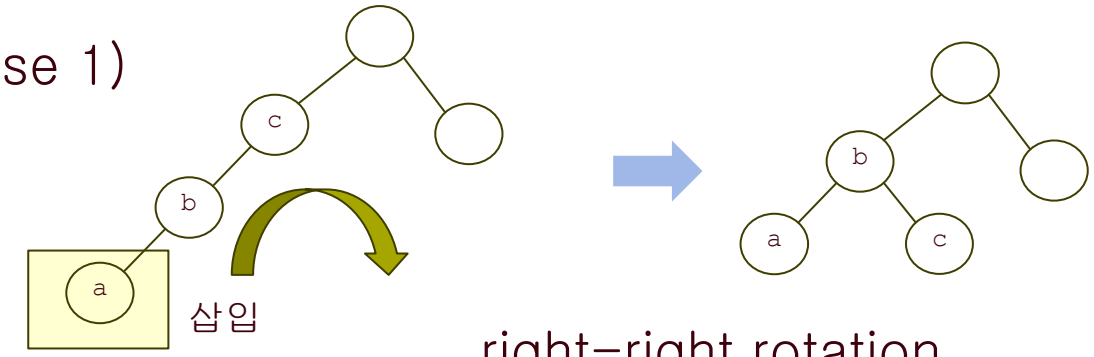
AVL 트리



$|h2-h1| \leq 1$

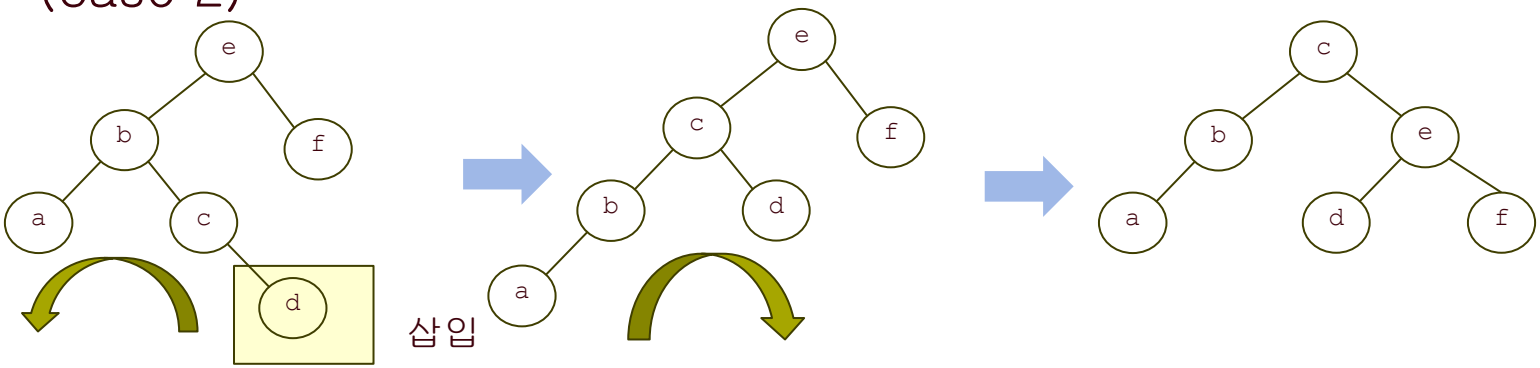
AVL 트리에서 데이터 추가 시 균형을 유지하는 방법

(case 1)



right-right rotation

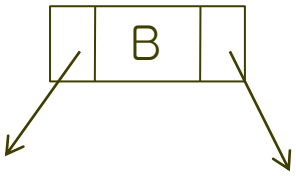
(case 2)

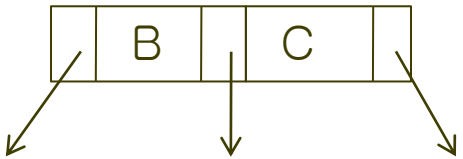


left-right rotation

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

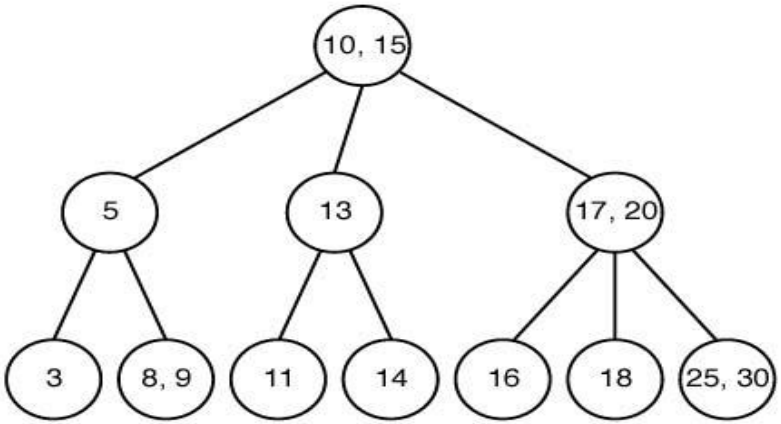
- B-tree 중 가장 간단한 형태인 2-3 트리
- 성질
  - ✓ 각 마디에는 키가 하나 또는 둘 존재
  - ✓ 각 내부마디의 자식 수는 키의 수+1
  - ✓ 어떤 주어진 마디의 왼쪽(오른쪽) 부분트리의 모든 키들은 그 마디에 저장되어 있는 마디보다 작거나(크거나) 같다.
  - ✓ 모든 잎마디는 수준이 같다
  - ✓ 데이터는 트리 내의 모든 노드에 저장



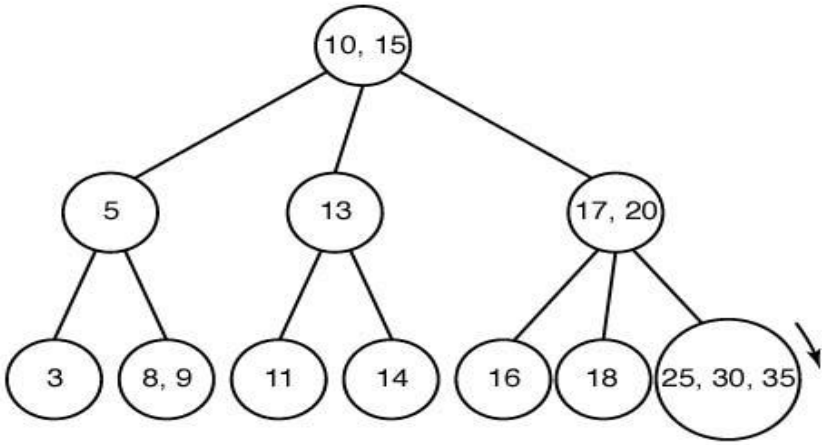


2-3 트리에서 데이터 추가 시 균형을 유지하는 방법

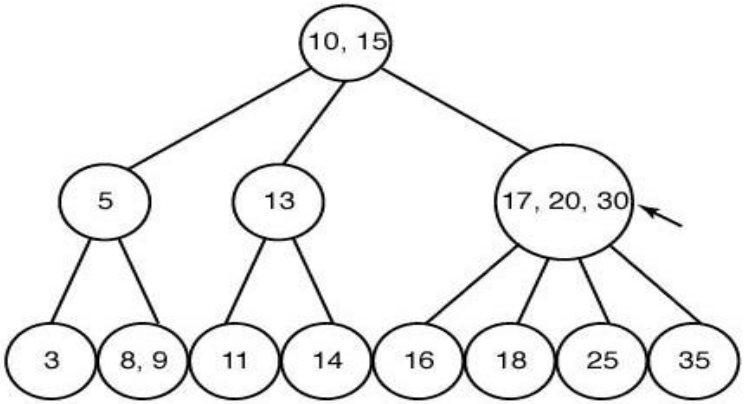
(a) A 3–2 tree



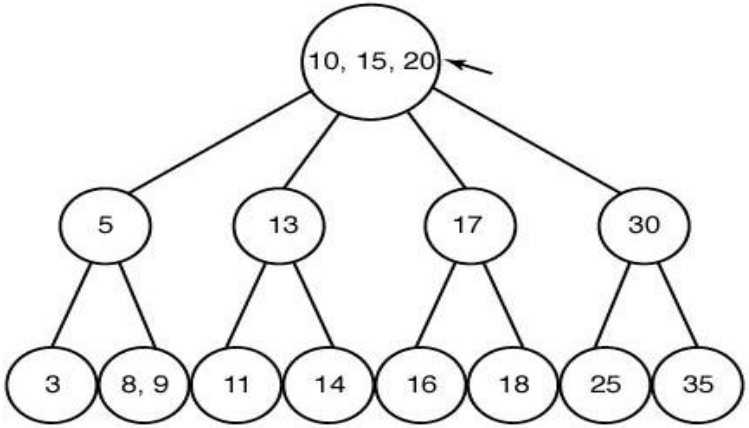
(b) 35 is added to the tree in sorted sequence in a leaf.



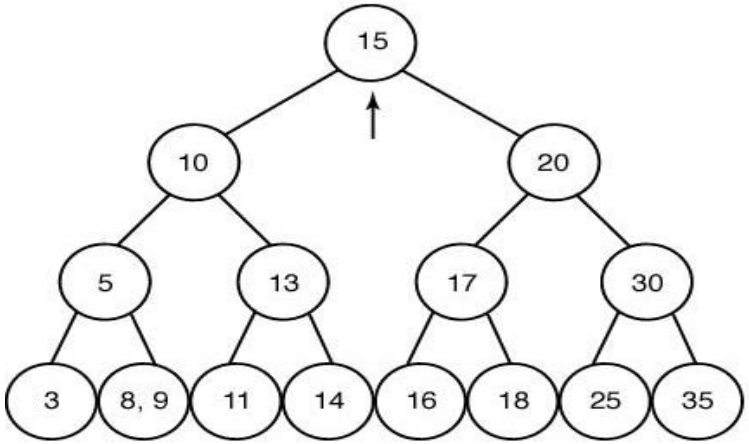
(c) If the leaf contains three keys, it breaks into two nodes and sends the middle key up to its parent.



(d) If the parent now contains three keys, the process of breaking open and sending the middle key up repeats.



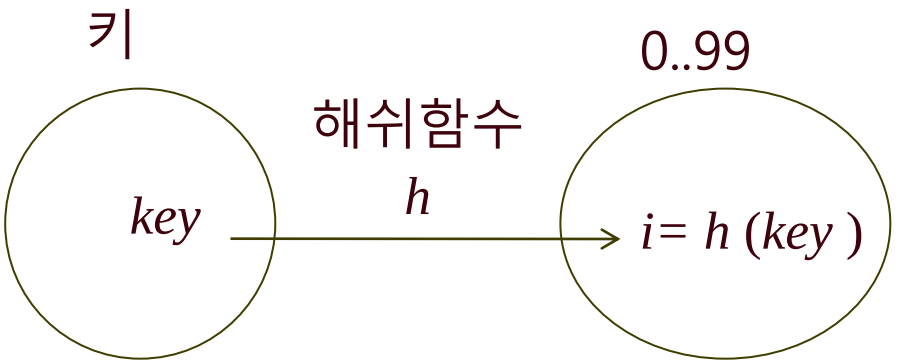
(e) Finally, if the root contains three keys, it breaks open and sends the middle key up to a new root.



해싱(Hashing)

- 만약 키가 주민등록번호라면 해당 번호의 저장소를 모두 만들 수는 없음 – 10^{13} 개 필요

- **해법**: 0..99의 첨자를 가진 크기가 100인 배열을 만든 후에, 키를 0..99 사이의 값을 가지도록 해쉬(hash)한다. 여기서 해쉬함수는 키를 배열의 첨자 값으로 변환하는 함수이다. 해쉬함수의 예: $h(\text{key}) = \text{key} \% 100$



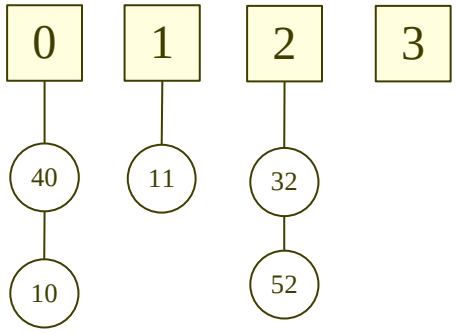
- 100개의 키가 있고, 이들이 모두 다른 해쉬값을 가질 확률: 즉 100개의 데이터가 각기 다른 방에 저장될 확률.

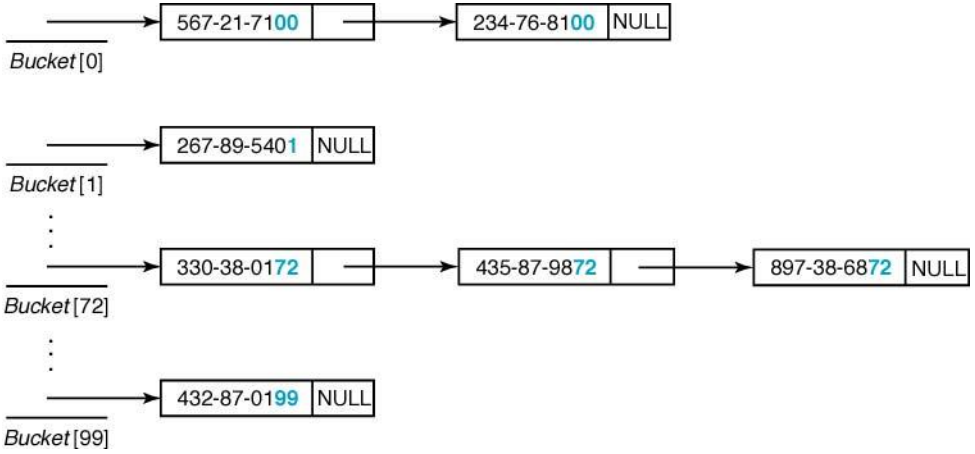
$$\frac{100!}{100^{100}} \approx 9.3 \times 10^{-43}$$

- 2개 이상의 키가 같은 해쉬값을 갖는 경우 충돌(collision)이 발생

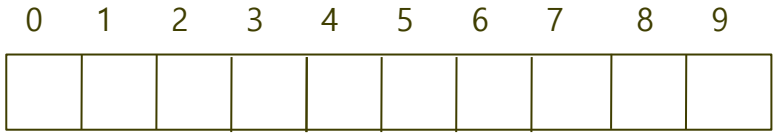
- solution to avoid collision:
  - open hashing (=closed addressing, chaining, separate chaining),
  - closed hashing (=open addressing): linear probing, quadratic probing, double hashing

✓ open hashing





✓ closed hashing



Store the keys only in the bucket

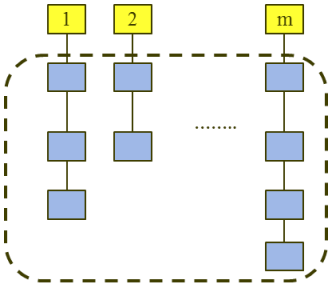
open hashing (=closed addressing, chaining, separate chaining

- 모든 키가 같은 해쉬값을 가질 확률 → 순차검색과 같은 상황

$$100\times(\frac{1}{100})^{100}=10^{-198}$$

- 해쉬이 효율적이 되기 위해서는 키가 바구니에 균일하게 분포하여야 함. n개의 키와 m개의 바구니가 있을 때, 각 바구니에 평균적으로 n/m개의 키를 갖게 하면 된다.

- **정리**: n개의 키가 m개의 바구니에 균일하게 분포되어 있다면, 검색에 실패한 경우 비교 횟수는 n/m이다.



- **정리**: n개의 키가 m개의 바구니에 균일하게 분포되어 있고, 각 키가 검색하게 될 확률이 모두 같다면, 검색에 성공한 경우 비교 횟수는 $\frac{n}{2m}+\frac{1}{2}$

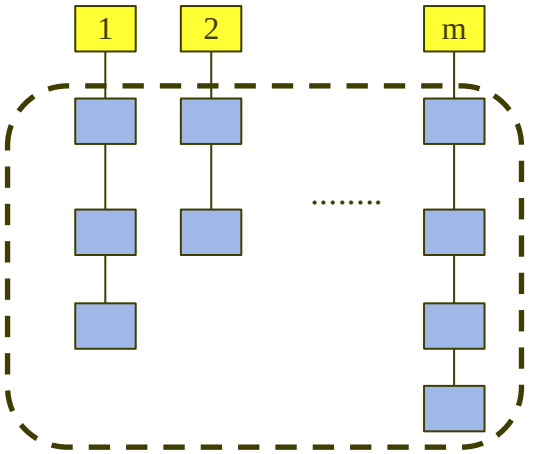
✓ **증명**: 각 바구니의 평균 검색시간은 $\frac{n}{m}$ 개의 키를 순차검색하는 평균시간과 같다. x개의 키를 순차검색하는데 걸리는 평균 검색시간은

$$1\times\frac{1}{x}+2\times\frac{1}{x}+\ldots+x\times\frac{1}{x}=\frac{1}{x}\sum_{i=1}^{x}i$$

$$=\frac{1}{x}\times\frac{x(x+1)}{2}$$

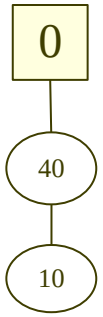
$$=\frac{x+1}{2}$$

따라서 $\frac{\frac{n}{m}+1}{2}=\frac{n}{2m}+\frac{1}{2}$



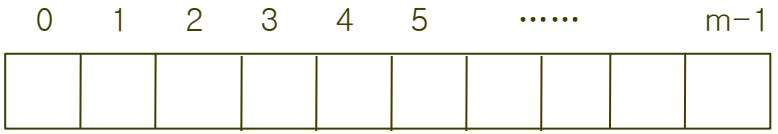
n개의 데이터

- 보기: 키가 균일하게 분포되어 있고 $n = 2m$일 때
  - ✓ 검색 실패 시 걸리는 시간 = $\frac{2m}{m} = 2$
  - ✓ 검색 성공 시 걸리는 시간 = $\frac{2m}{2m} + \frac{1}{2} = \frac{3}{2}$



closed hashing (Open addressing) :

- Linear probing
- Quadratic probing
- Double hashing

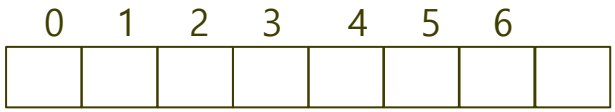


bucket 내에만 데이터 저장

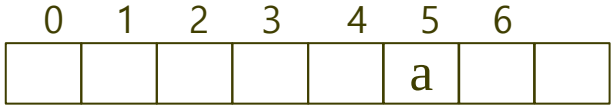
- ✓ k= key, m=hash table 크기
- ✓ linear probing: 오른쪽으로 이동하면서 빈칸에 저장
  - h(i,k)=(h(k)+i) mod m, i=0,1,2,3,...
- ✓ quadratic probing: $h(i,k)=(h(k)+i^2)$ mod m, i=0,1,2,3...
- ✓ double hashing: $h(i,k)=(h_1(k)+i*h_2(k))$ mod m, i=0,1,2,3,... 해쉬함수 $h_1(k)$사용. 충돌 시 해쉬함수 $h_2(k)$ 사용
- ✓ 자료가 삭제될 경우 문제발생 가능성

Linear probing 경우 데이터가 삭제될 경우의 문제점

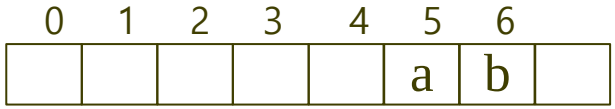
h(a)=h(b)=5



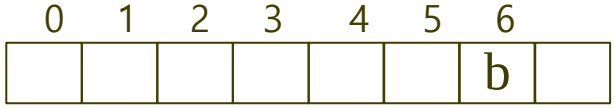
1) a 저장:



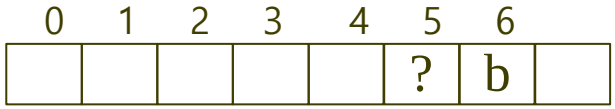
2) b 저장:



3) a 삭제:



4) b 검색

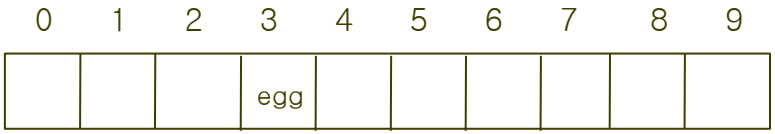


Linear Probing

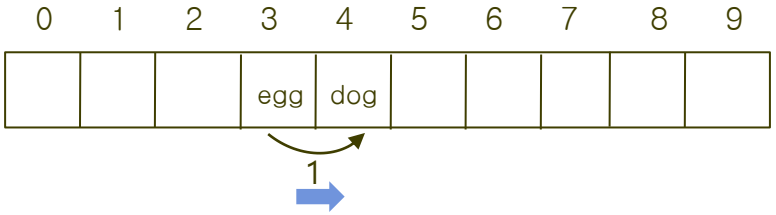
- 오른쪽으로 이동하면서 빈칸에 저장
- h(i,k)=(h(k)+i) mod m, i=0,1,2,3,...
- k: key, h: 해쉬함수
- 데이터 입력 순서: egg, dog, cat

우측으로 이동하다가 더 이상 칸이 없는 경우 0의 위치 확인

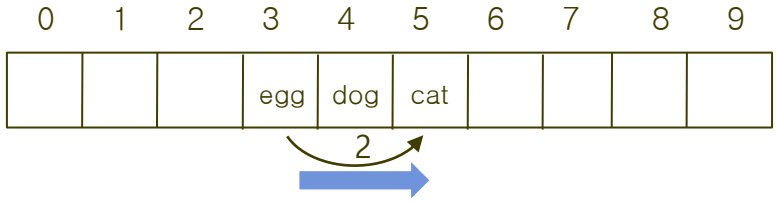
h(egg)=3



h(dog)=3



h(cat)=3 이 입력되면,

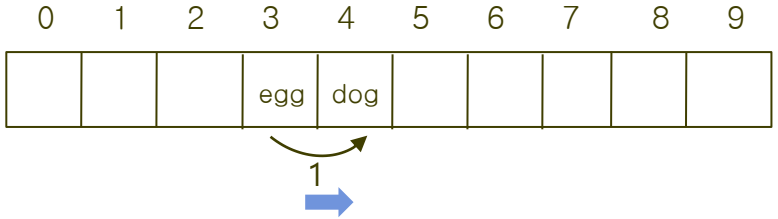


Quadratic Probing

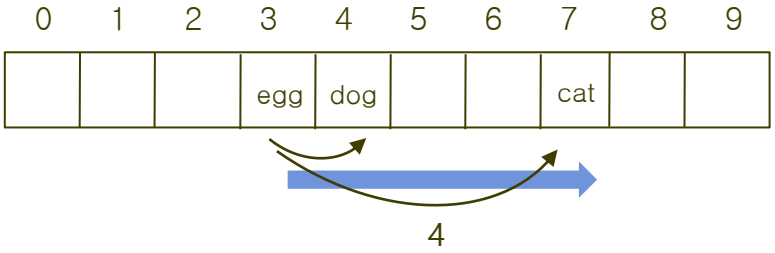
- $h(i,k)=(h(k)+i^2) \bmod m$, $i=0,1,2,3,\ldots$
- h(k)의 칸에 이미 데이터가 있을 때, i^2 값을 이용하여 저장 위치를 계산
- 데이터 입력 순서: egg, dog, cat

h(egg)=3

h(dog)=3



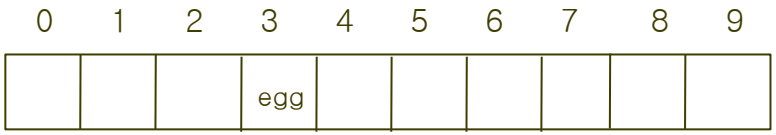
h(cat)=3 이 입력되면,



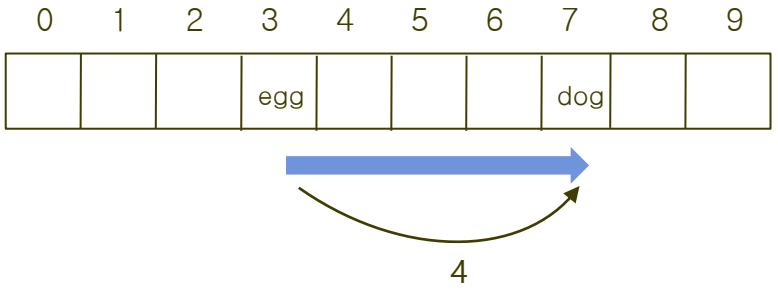
Double Hashing

- $h(i,k)=(h_1(k)+i*h_2(k)) \bmod m$, i=0,1,2,3,...
- $h_1(k)$의 칸에 이미 데이터가 있을 때, $i*h_2(k)$ 값을 이용하여 저장 위치를 계산
- 데이터 입력 순서: egg, dog, cat

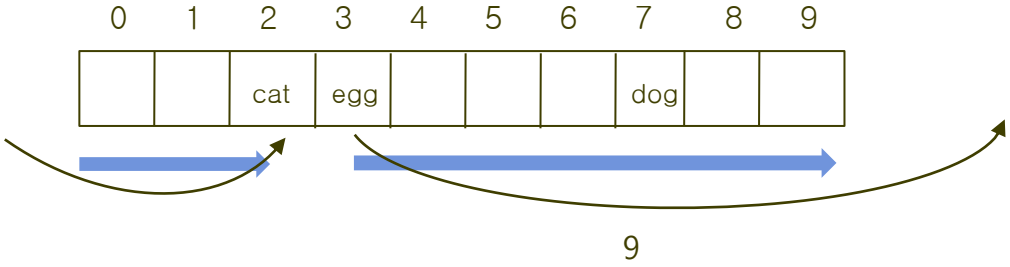
$h_1(egg)=3$



$h_1(dog)=3$, $h_2(dog)=4$



$h_1(cat)=3$, $h_2(cat)=9$



단순한 형태의 해쉬함수

$h(k) = k \bmod m$. (m slots)

- $m = 2^p$ or 10^p는 사용 안함 → k의 끝 p자리를 나타냄
- 2의 지수배가 아닌 prime number를 m으로 사용

(예) $n = 2000$, data 하나는 8bits, Chaining사용, 평균 3번의 Unsuccessful Search 감수

$a = 2000/3$, $m = 701$ (a 에 가까우면서 2^p에 가깝지 않은 소수)

$h(k) = k \bmod 701$

선택문제(selection problem)

- 키가 n개인 리스트에서 k번째로 큰(또는 작은) 키를 찾는 문제
- 키가 정렬되어 있지 않다고 가정
- 사례
  - ✓ k=1
  - ✓ 최대, 최소
  - ✓ k=2

최대(max)키 찾기

크기 n인 배열 S에서 최대키를 찾음

```
void find_largest(int n, const keytype S[], keytype& large){
     index i;

     large = S[1];
     for (i=2; i<= n; i++)
       if (S[i] > large)
              large = S[i];
  }
```

$$T(n) = n - 1$$

정리 8.7 키 비교만 수행하여, 가능한 모든 입력 n개 중에서 최대키를 찾을 수 있는 결정적 알고리즘은 어느 경우라도 반드시 최소한 n-1 번의 비교를 해야 한다.

최소키와 최대키 찾기

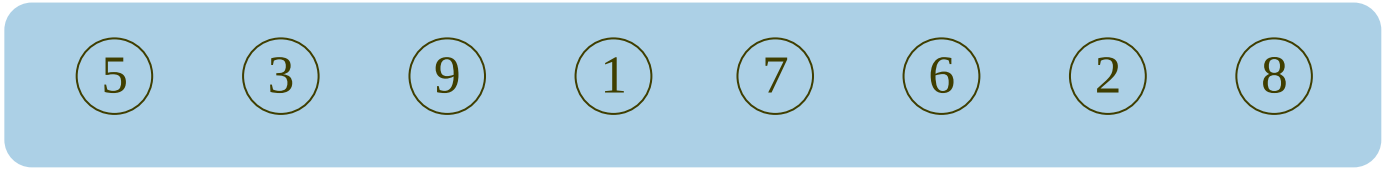
```
void find_both (int n, const keytype S[], keytype& small, keytype& large){
     index i;

     small = S[1];
     large = S[1];
     for (i=2; i<= n; i++)
       if (S[i] < small)
              small = S[i];
       else if (S[i]> large)
              large = S[i];
  }
```

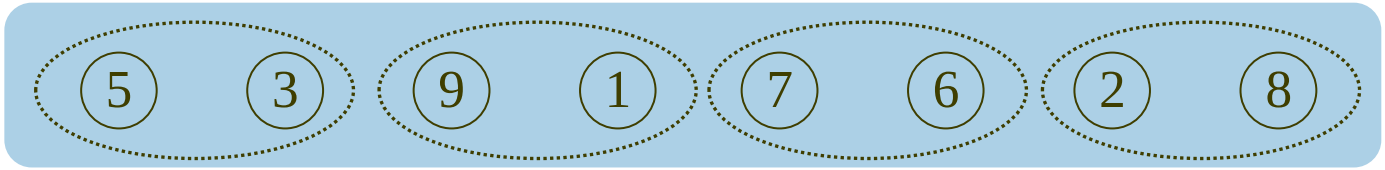
$$W(n) = 2(n\text{-}1)$$

worst case: S[1] is the smallest element of S.

키를 짝 지워서 최소키와 최대키 찾기



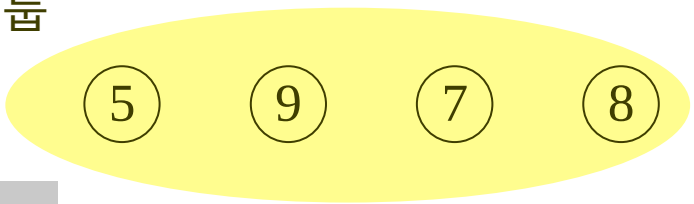
짝을 지어 L, S를 찾음



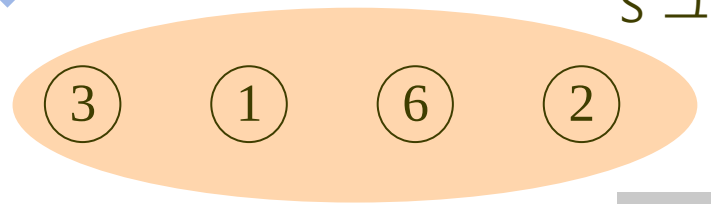
n/2 비교

L, S 그룹 생성

L 그룹



S 그룹



n/2-1 비교

Find MAX

Find MIN

n/2-1 비교

총 O(3*n*/2) 비교

키를 짝 지워서 최소키와 최대키 찾기

n: even number

```
void find_both2t(int n, const
keytype S[], keytype& small,
keytype& large){
     index i;

     if (S[1] < S[2]) {
        small = S[1];
        large = S[2];
        }
     else  {
        small = S[2];
        large = S[1];
        }
```

비교

(n-2)/2회 반복

```
for (i=3; i<= n-1; i=i+2){
      if (S[i] < S[i+1]){
         if ( S[i] < small)
             small = S[i];
         if ( S[i+1] > large)
             large = S[i+1];
         }
      else  {
         if ( S[i+1] < small)
             small = S[i+1];
         if ( S[i] > large)
             large = S[i];
         }
   }
}
```

비교

비교

비교

# of repetitions

$$T(n) \approx \frac{n-2}{2} \times 3$$

# of comparisons

최소키와 최대키 찾기

정리 8.8

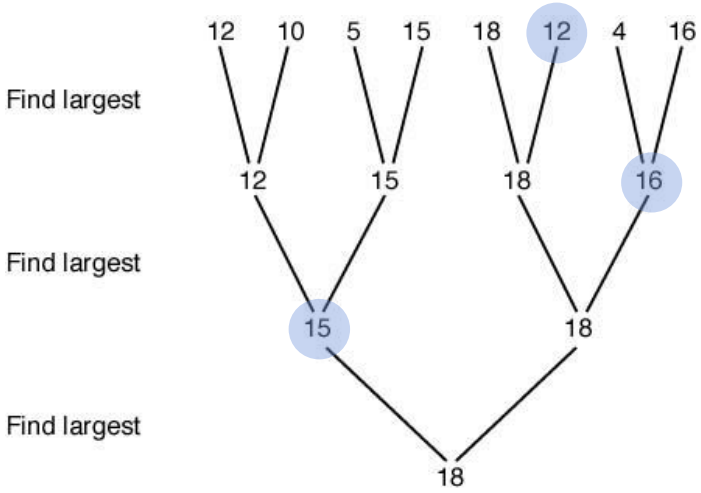
키를 비교만 해서, 모든 가능한 입력에서 n개의 키 가운데 최소키와 최대키를 모두 찾을 수 있는 결정적 알고리즘은 최악의 경우 최소한 다음 만큼의 비교를 수행해야 한다.

$$W(n) = \begin{cases} \frac{3n}{2} - 2 & \text{if } n \text{ is even} \\ \frac{3n}{2} - \frac{3}{2} & \text{if } n \text{ is odd} \end{cases}$$

차대키 (second largest key) 찾기

1. 단순한 방법: 먼저 최대키를 찾고(n-1회 비교), 이후 다음 최대키를 찾음 (n-2회 비교). 총 $2n$-3회 비교

2. Tournament Method:
   (단계1) 토너먼트를 시행하여 최대 우승자가 최대키이다.
   (단계2) 각 시합의 진 팀을 이긴 팀의 리스트로 만든다.
   (단계3) 우승팀의 리스트에서 최대값을 찾으면, 이것이 차대키이다.



2^{nd}=MAX{12,16,15}

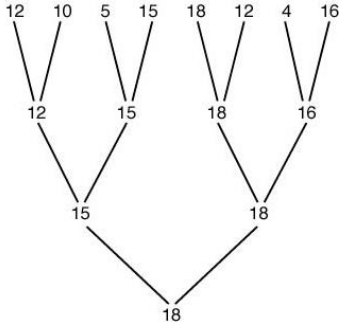
(단계1) 총 시합 횟수

$$T(n) = n \sum_{i=1}^{\lg n} \left(\frac{1}{2}\right)^i = n \left[\frac{1}{2} \times \frac{1-(1/2)^{\lg(n)}}{1-1/2}\right] = n-1$$

(단계3) 차대키 찾기

최대키의 리스트 크기는 $\lg n$.

그러므로 리스트 중 최대값 찾기는 $\lg n - 1$ 시간



합계: $T(n) = n\text{-}1+\lg n\text{ -}1 = n + \lg n\text{ -}2$

정리 8.9 키를 비교만 해서 모든 가능한 입력에서 n개의 키 가운데 차대키를 모두 찾을 수 있는 결정적 알고리즘은 최악의 경우 최소한 다음의 비교를 수행해야 한다.

$$n + \lceil \lg n \rceil - 2$$

k번째 작은 키 찾기

(1) 단순방법 : 정렬 후 k번째 선택 – $\Theta(n \lg n)$

(2) partition 사용: selection(1,n,k)

- ✓ quick sort의 partition을 사용
- ✓ $W(n) = n(n-1)/2$, $A(n) \approx 3n$

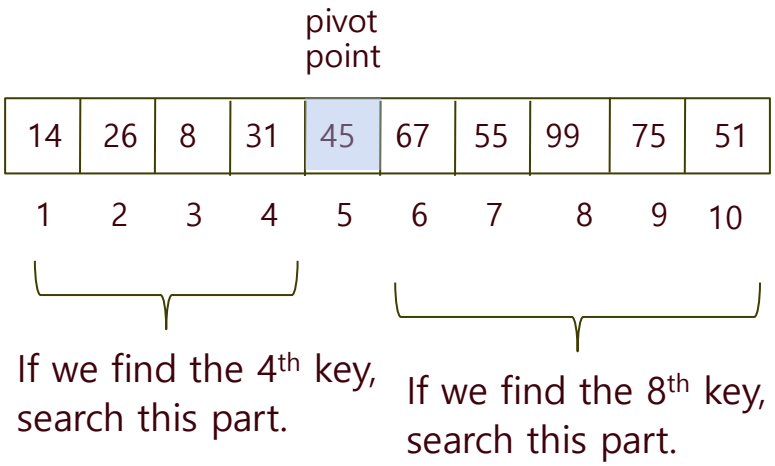
(3) $O(n)$ 방법

k번째 작은 키 찾기

(2) partition 사용: selection(1,n,k)

- ✓ quick sort의 partition을 사용
- ✓ $W(n) = n(n\text{-}1)/2$, $A(n) \approx 3n$

after partitioning with pivot element 45



```
keytype selection(index low, index high, index k){

  index pivotpoint;

  if (low == high)
     return S[low];
  else {
      partition (low, high, pivotpoint);
      if(k==pivotpoint)
         return S[pivotpoint];
      else if (k<pivotpoint)
         return selection(low, pivotpoint-1,k);
      else
         return selection(pivotpoint+1, high, k);
    }
}
```

```
void partition(index low, index high, index& pivotpoint){
  index i,j;
  keytype pivotitem;

  pivotitem = S[low];
  j=low;
  for(i=low+1; i<=high; i++)
      if(S[i]<pivotitem){
           j++;
           exchange S[i] and S[j];
      }
   pivotipoint = j;
   exchange S[low] and S[pivotpoint];
}
```

selection(1,*n*,*k*)의 시간 복잡도

- ✓ $W(n) = n(n-1)/2.$ $W(n) = W(n-1) + n-1$
- ✓ $B(n) = 2n.$ $B(n) = B(n/2) + n-1$
- ✓ $A(n) \approx 3n.$ 교재 참조

3. O(n) **방법**

$T(n) \le T(n/5)+T(3n/4)+cn$

```
procedure SELECT(k,S)
  if |S| < 50 then
      sort S
      return k^th smallest element in S
  else
      divide S into ⌊|S|/5⌋ sequences of 5 elements
      sort each 5-element sequence
      let M be the sequence of medians of the 5-element sets
      m ← SELECT(⌈|M|/2⌉, M)
      let S1, S2, and S3 be the sequences of elements in S less
        than, equal to, and greater than m, respectively.
      if |S1| ≥ k, then return SELECT(k,S1)
      else
          if |S1| +|S2|≥ k, then return m
          else return SELECT(k-|S1|-|S2|, S3)
```

O(n)

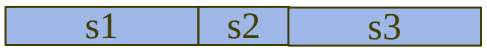
추가적으로 1~4개를 갖는 최대 하나의 그룹

T(n/5)

O(n)

T(3n/4)

뒤에 설명



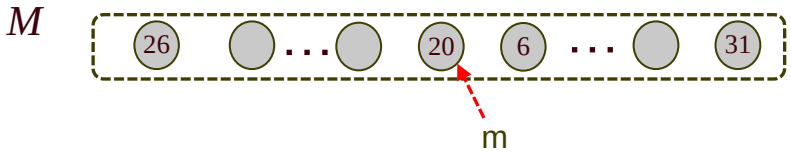
SELECT(k,S) 알고리즘의 수행 단계 설명

1. 데이터를 5개의 묶음으로 만듦

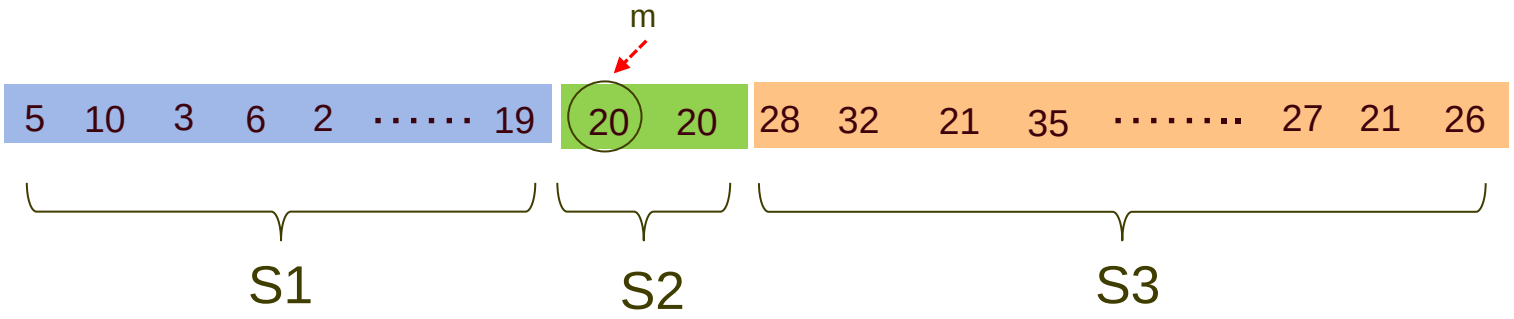
2. 5개의 묶음 내에서 정렬

3. 5개의 묶음의 가운데 값들을 모아 M 생성

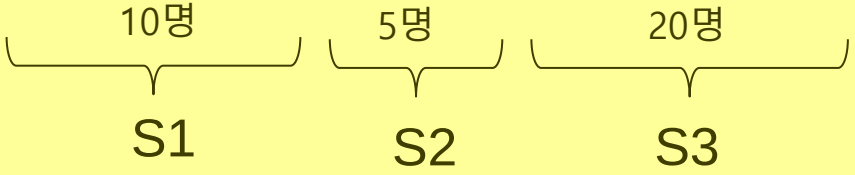
4. M에서 $\lceil |M|/2 \rceil$ 번째(가운데)를 찾음 : $m \leftarrow \text{SELECT}(\lceil |M|/2 \rceil, M)$



5. m보다 작은 데이터 S1, 같은 데이터 S2, 큰 데이터 S3를 생성



6. if |S1| ≥ k, then return SELECT(k,S1)
   else
      if |S1| +|S2|≥ k, then return *m*
      else return SELECT(k-|S1|-|S2|, S3)



(1) 5등을 찾는다면
(2) 12등을 찾는다면
(3) 23등을 찾는다면

7. 작아진 문제에 대해 select 수행

$T(n) \le T(n/5)+T(3n/4)+cn$

```
procedure SELECT(k,S)
  if |S| < 50 then
      sort S
      return k^th smallest element in S
  else
      divide S into ⌊|S|/5⌋ sequences of 5 elements
      sort each 5-element sequence
      let M be the sequence of medians of the 5-element sets
      m ← SELECT(⌈|M|/2⌉, M)
      let S1, S2, and S3 be the sequences of elements in S less
        than, equal to, and greater than m, respectively.
      if |S1| ≥ k, then return SELECT(k,S1)
      else
          if |S1| +|S2|≥ k, then return m
          else return SELECT(k-|S1|-|S2|, S3)
```

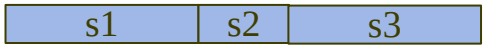
O(n)

T(n/5)

O(n)

T(3n/4)

뒤에 설명



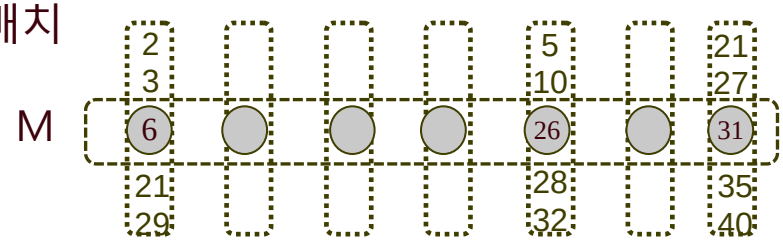
S1과 S3의 크기의 상한에 대한 설명

1. 데이터를 5개의 묶음으로 만듦

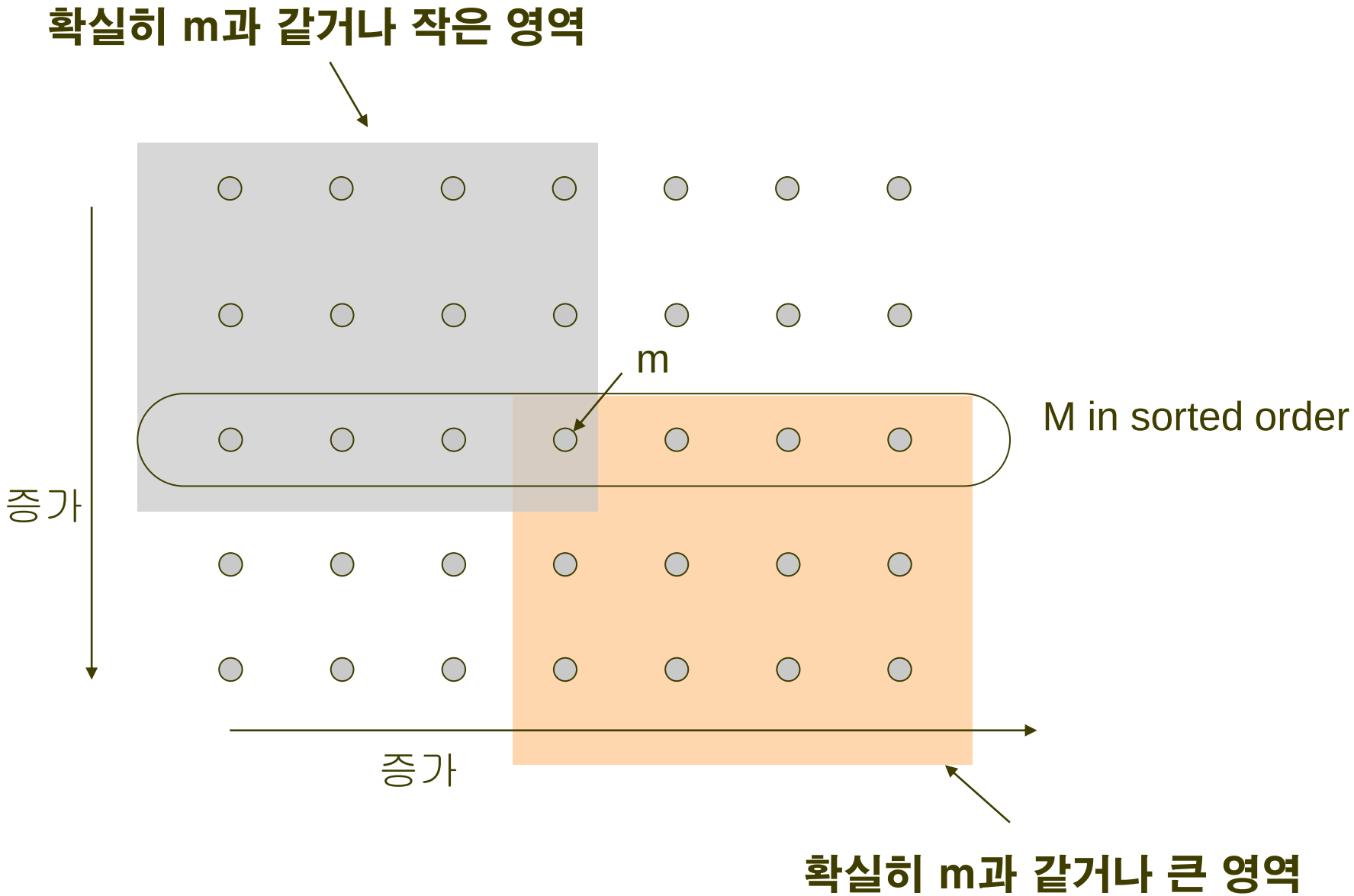
2. 5개의 묶음 내에서 정렬

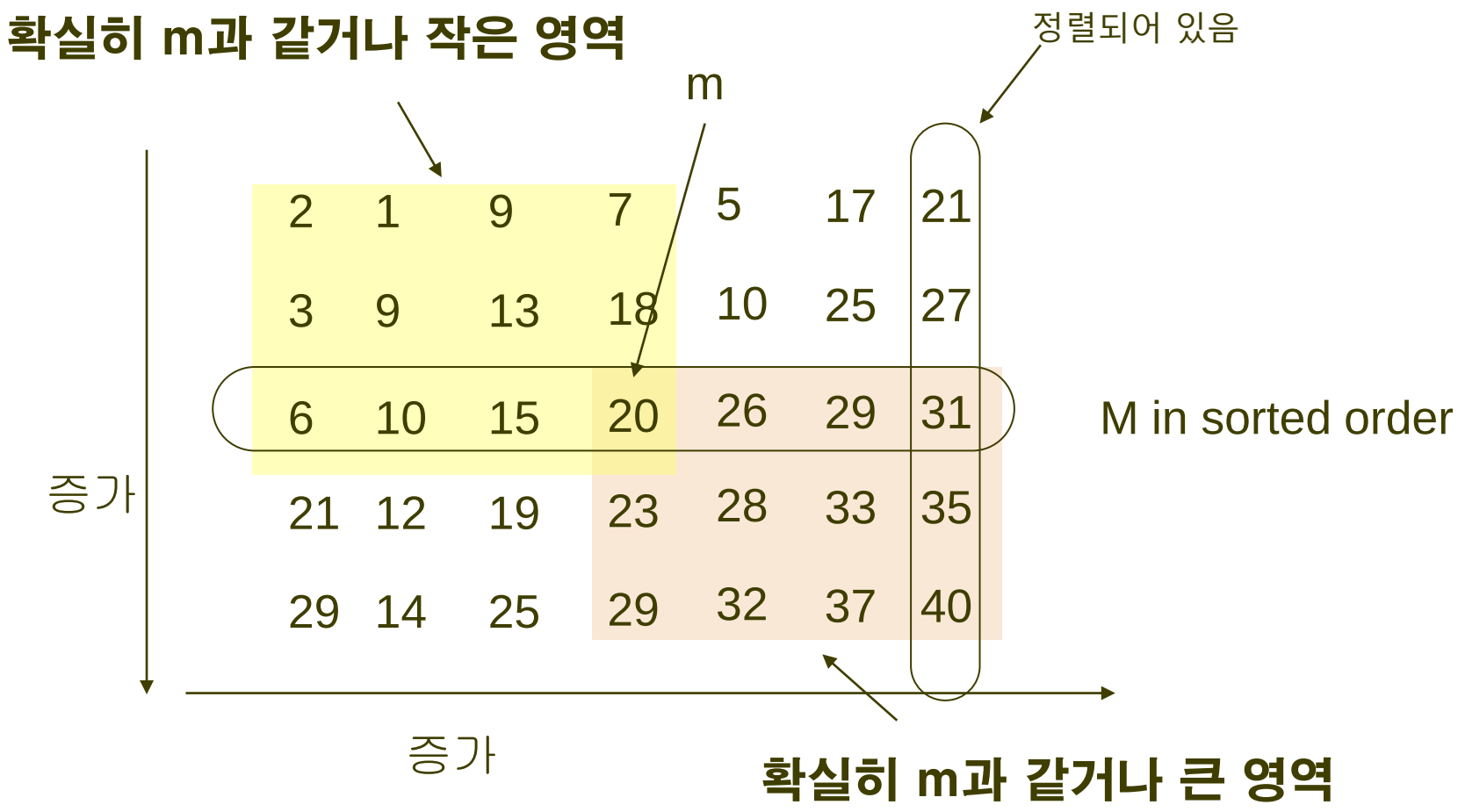
3. 5개의 묶음의 가운데 값들을 이용하여 묶음 정렬

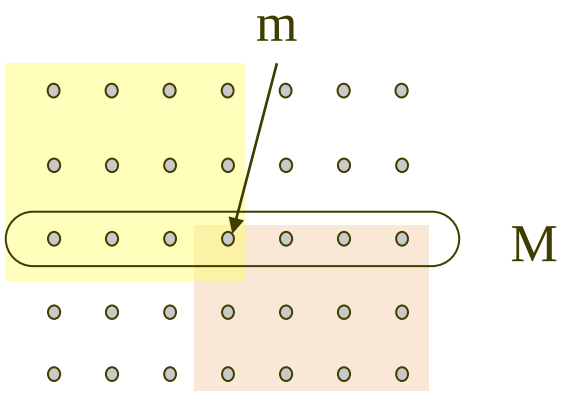
4. 묶음들을 재 배치



◆ 실제로 M은 정렬된 것은 아님


확실히 m과 같거나 작은 영역
m
M in sorted order
증가
증가
확실히 m과 같거나 큰 영역


확실히 m과 같거나 작은 영역
정렬되어 있음
m
2 1 9 7 5 17 21
3 9 13 18 10 25 27
6 10 15 20 26 29 31
21 12 19 23 28 33 35
29 14 25 29 32 37 40
M in sorted order
증가
증가
확실히 m과 같거나 큰 영역



let S1, S2, and S3 be the sequences of elements in S less than, equal to, and greater than *m*, respectively.

n/5/2

- S1과 S3는 최대 3n/4의 크기를 갖는다. S1과 S3는 m과의 비교로 정확히 파악 가능

[증명]

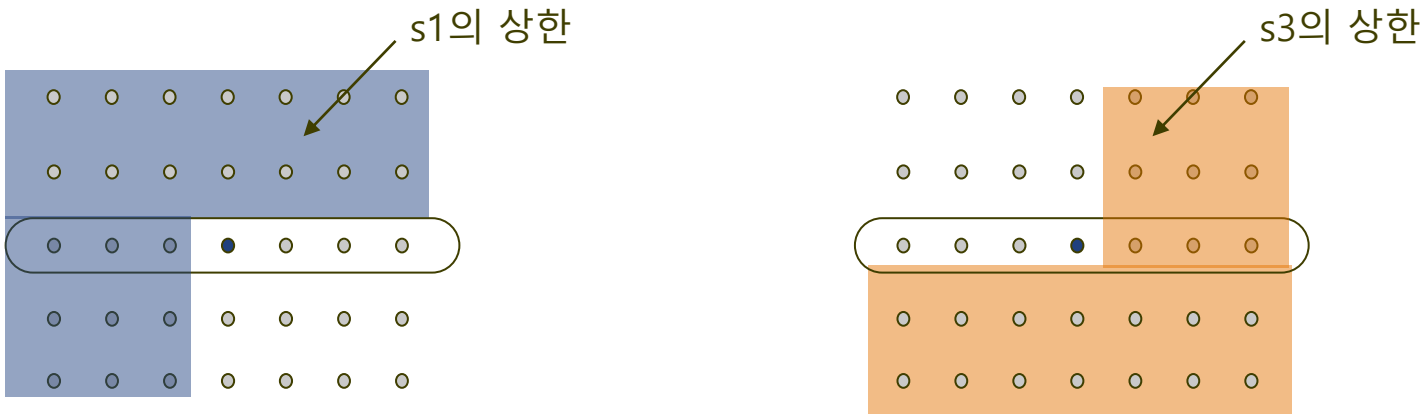
- ✓ M 중 적어도 $\lceil n/10 \rceil$ 개의 원소가 m보다 크거나 같다.
- ✓ 이들 각 원소에 대해 S중에 적어도 2개의 원소가 그 값보다 같거나 큰 원소가 존재
- ✓ n≥50에서 S1의 크기는 최대 $n - 3\lceil n/10 \rceil$. 이 값은 < 3n/4.
- ✓ for S3, similar.

$T(n) \le cn$ for $n \le 49$

$T(n) \le T(n/5)+T(3n/4)+cn$ for $n \ge 50$

이 관계로부터 $T(n) \le 20cn \in O(n)$

```
procedure SELECT(k,S)
 if |S| < 50 then
    sort S
    return k^th smallest element in S                    O(n)
 else
    divide S into ⌊|S|/5⌋ sequences of 5 elements        T(n/5)
    sort each 5-element sequence
    let M be the sequence of medians of the 5-element sets
    m ← SELECT(⌈|M|/2⌉, M)
    let S1, S2, and S3 be the sequences of elements in S less
      than, equal to, and greater than m, respectively.  O(n)
    if |S1| ≥ k, then return SELECT(k,S1)                T(3n/4)
    else
     if |S1| +|S2|≥ k, then return m
      else return SELECT(k-|S1|-|S2|, S3)
```



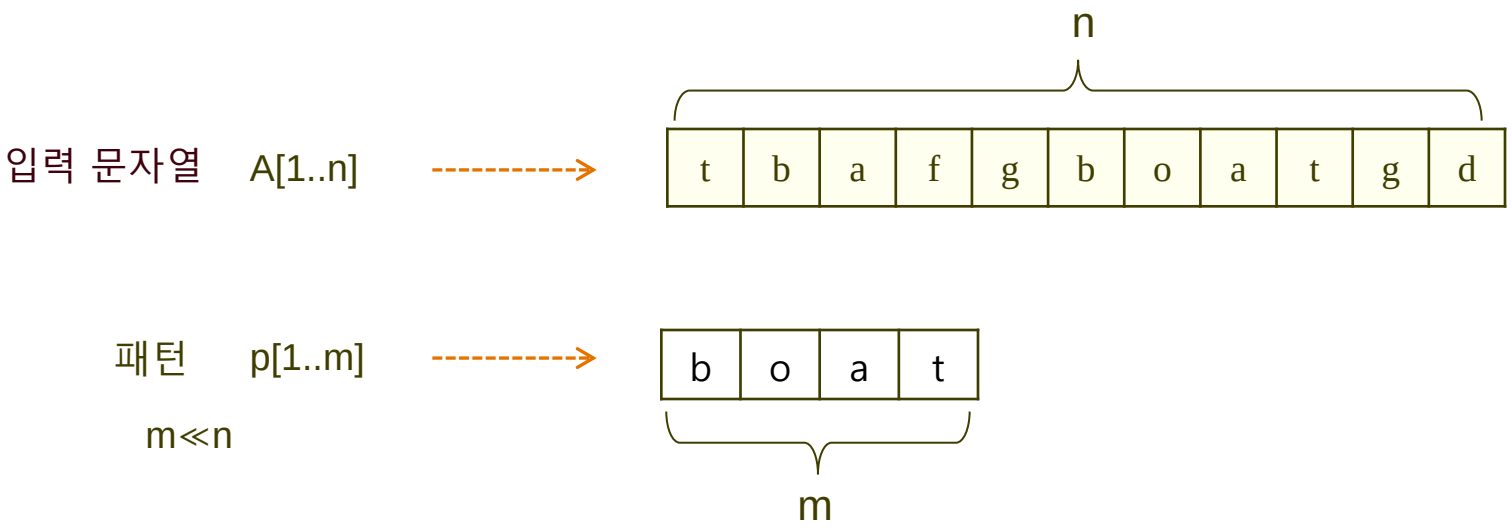
- ✓ selection( ) - 단순 partition을 사용할 경우
  - 임의의 pivot item 선정

- ✓ select( ) - median of median 방법
  - 한 쪽의 데이터 개수를 3/4 n 으로 한정

문자열 매칭

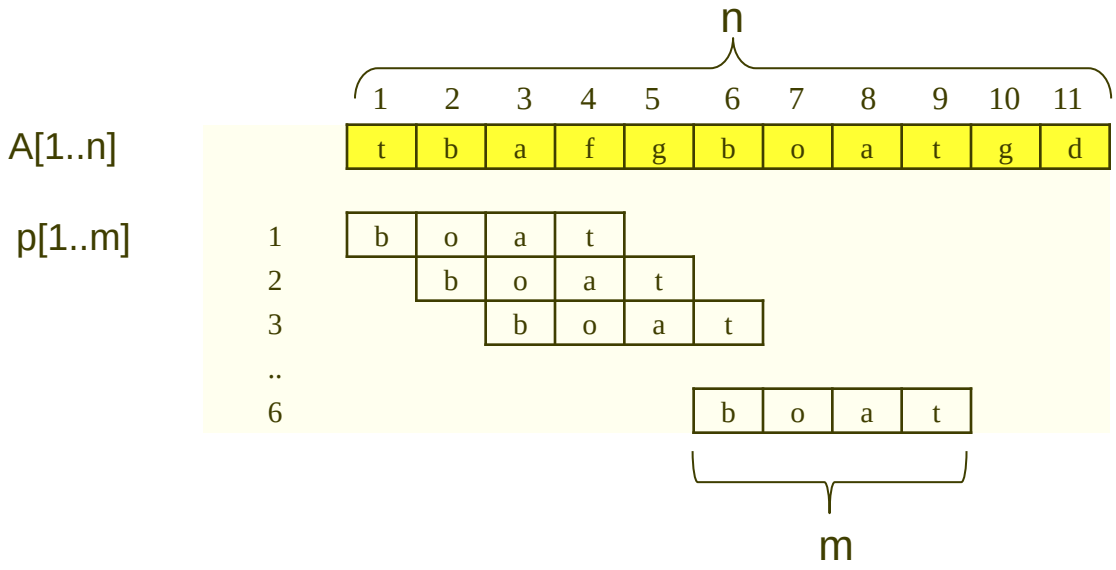
- 입력 문자열에서 패턴을 찾는 문제

1. 원시적인 매칭 방법
2. 오토마타를 이용한 방법
3. Rabin-Karp 알고리즘
4. Boyer-Moore 알고리즘



– 문자열 매칭은 '쉽게 배우는 알고리즘'(문병로 저)에서 발췌

1. 원시적인 매칭 방법



✓ 한 칸씩 오른쪽으로 이동하면서 매칭 시도

```
naiveMatching(A,p) {
    for i=1 to n-m+1 {
        if (p[1..m]==A[i..i+m-1])
            matching is found at A[i]
}
```

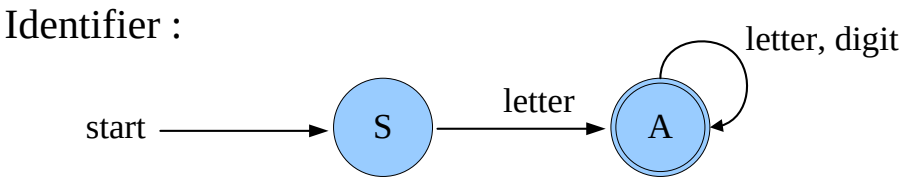
n-m+1 반복

O(m)

- 패턴의 앞부터 검사
- total O(mn) (m ≪ n)

2. 오토마타를 이용한 방법

- 상태 전이 함수 δ를 이용한 방법
- 어휘분석
- state transition diagram




```
FiniteAutomataMatcher(A,δ,f) {//f: final state
    q = 0;
    for i=1 to n {
        q = δ(q, A[i]);
        if (q=f)
              matching is found at A[i-m+1]
}
```

- total $\Theta(n)$

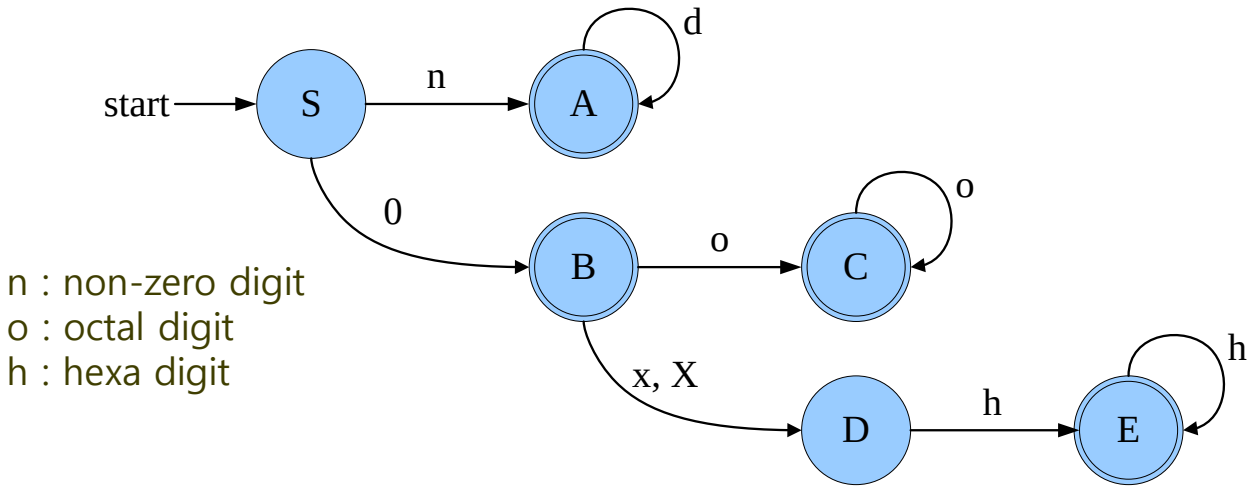
Integer number Recognition

- Form : 10진수, 8진수, 16진수로 구분되어진다.

  10진수 : 0이 아닌 수 시작

  8진수 : 0으로 시작, 16진수 : 0x, 0X로 시작

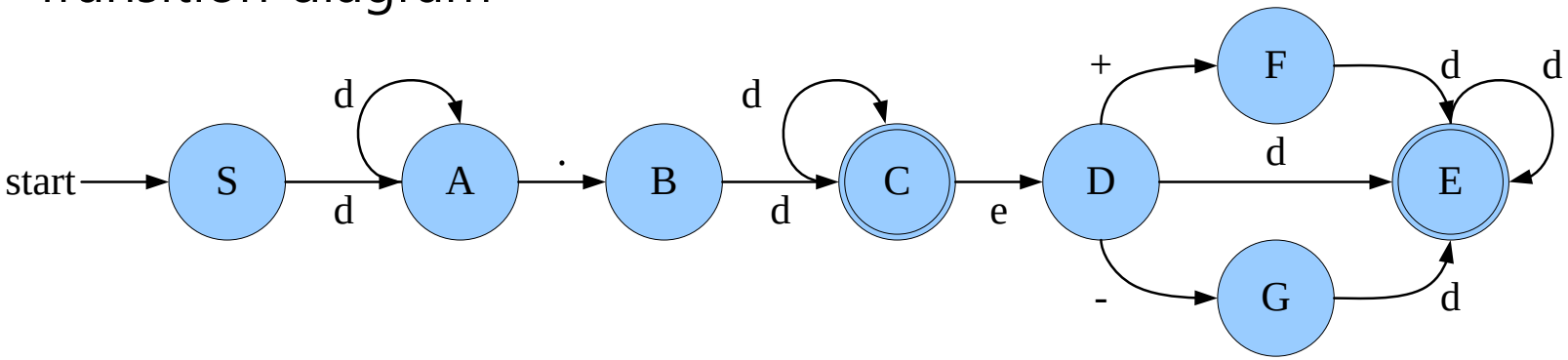
- Transition diagram



123a ?

Real number Recognition

- Form : Fixed-point number & Floating-point number
- Transition diagram



3. Rabin-Karp 알고리즘

- 문자열패턴을 수치로 바꾸어 수치 비교를 통해 패턴을 찾음
- p[1..m]이 10진수 이면

$$v = p[m]+10 \times p[m-1]+10^2 \times p[m-2]+\ldots+10^{m-1} \times p[1]$$

$$= p[m]+10 \times (p[m-1]+10 \times (p[m-2]+\ldots+10 \times p[1]))..)$$

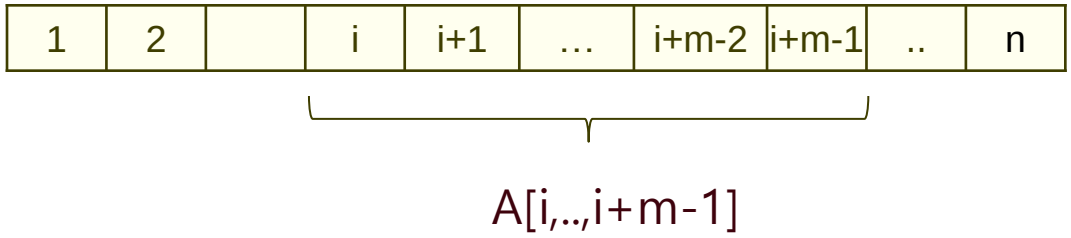
8	5	6	1

- $v = 1+10 \times 6+10^2 \times 5+10^3 \times 8$

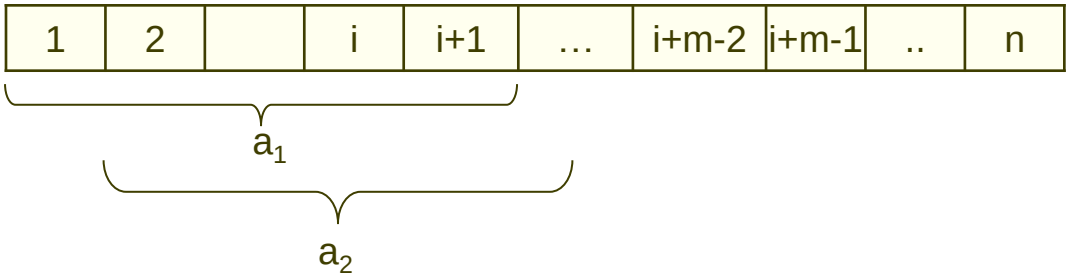
$$= 1+10 \times (6+10 \times (5+10 \times 8))$$

- 문자열 A[i,..,i+m-1]의 값은

  $a_i = A[i+m-1]+10 \times (A[i+m-2]+10 \times (A[i+m-3]+\ldots+10 \times A[i]))..)$



- a_i 계산은 $\Theta(m)$ 필요
- 입력 문자열에 대해 처음 m개의 문자별로 a_i를 계산

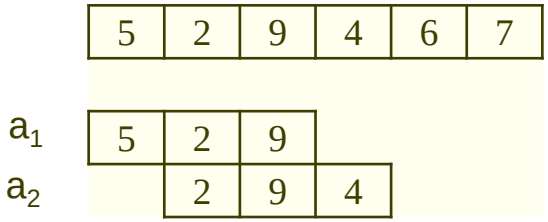


  - 횟수는 n-m+1번 (i=1 to n-m+1).
  - 따라서 a_i 값과 v를 i를 변경하면서 비교하면 총 수행시간 $\Theta(mn)$
  - 문자를 d 진수로 표현한 경우, 10을 d로 변경.
  - d는 다른 base를 사용 가능

개선방안

- 매번 모든 계산을 해야 하는 것은 아니다.
- a_i 계산시 a_{i-1} 이용

$$a_i = A[i+m-1]+10 \times (A[i+m-2]+10 \times (A[i+m-3]+\ldots+10 \times A[i])))$$
$$= 10 \times (a_{i-1} - 10^{m-1} \times A[i-1]) + A[i+m-1]$$



$a_1 = 529$

$a_2 = 10 \times (529-10^2 \times 5)+4 = 294$

- 이 방식으로는 한번의 a값을 계산하는데 곱셈 2회, 덧셈 2회 필요
- n-m+1 반복, m≪n
- 총 $\Theta(n)$
- 문제점: 계산값이 너무 커질 수 있다.

```
basicRabinKarp {

    v ← 0; a_1 ← 0; //d 진수

for i=1 to m
    v = d x v + p[i];
    a_1 = d x a_1 + A[i]

for i=1 to n-m+1
    if(i≠1) a_i ← d x (a_{i-1} - d^{m-1} x A[i-1]) + A[i+m-1]
    if(v = a_i ) matching is found at i
}
```

mod 함수를 이용한 개선

- q: prime number
- d x q 가 컴퓨터가 처리 가능한 범위가 되도록 q설정
- a_i 대신 $b_i = a_i \bmod q$ 사용

원래 방법

$a_i = 10 \times (a_{i-1} - 10^{m-1} \times A[i-1]) + A[i+m-1]$

$b_i = (d \times (b_{i-1} - (d^{m-1} \bmod q) \times A[i-1]) + A[i+m-1]) \bmod q$

- ✓ $d^{m-1} \bmod q$ 는 미리 계산해서 활용
- ✓ (예)

	5	2	9	4	6	7
a_1	5	2	9			
a_2		2	9	4		

q=11 이면

$b_1 = 529 \bmod 11 = 1$

$b_2 = (10 \times (1 - (10^2 \bmod 11) \times 5) + 4) \bmod 11$

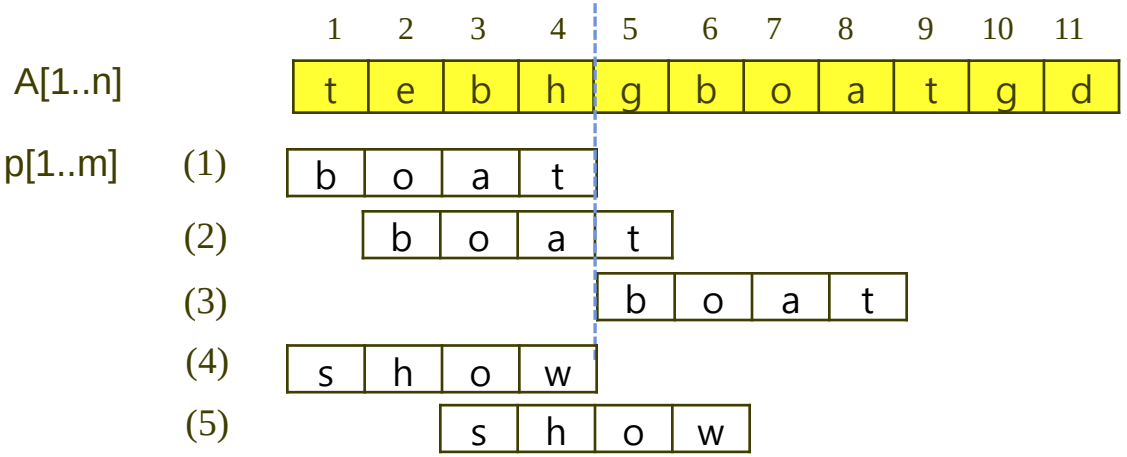
$= 8$

```
RabinKarp {
   v ← 0; b1 ← 0; //d 진수
   for i=1 to m
      v = (d x v + p[i]) mod q;
      b1 = (d x b1 + A[i]) mod q
   end for
   h = d^(m-1) mod q
   for i=1 to n-m+1
        if(i≠1) bi ← (d x (bi-1 - h x A[i-1]) + A[i+m-1]) mod q
        if(v == bi){  // 같은 값이면 매칭이 실제 되는지 확인해야 함
           if(p[1..m] == A[i..i+m-1]) matching is found at i
           }
   end for
```

- ✓ b_i 계산에 2회 곱셈, 2회 덧셈. mod 1회
- ✓ n-m+1 반복 → 총 $\Theta(n)$
- ✓ 매칭확인 시간 → $\Theta(m)$ (mod 계산에 의해 패턴이 다른 데이터에서 매칭이 일어날 수 있다)
- ✓ 총 k번의 매칭 확인이 필요하다면 총 $\Theta(n+km)$
- ✓ 충분히 큰 q>n 일 때 false matching은 1회 미만.
- ✓ 결론: $\Theta(n)$

4. Boyer-Moore 알고리즘

Boyer-Moore-Horspool: Boyer-Moore의 약식 알고리즘(단순 형태)



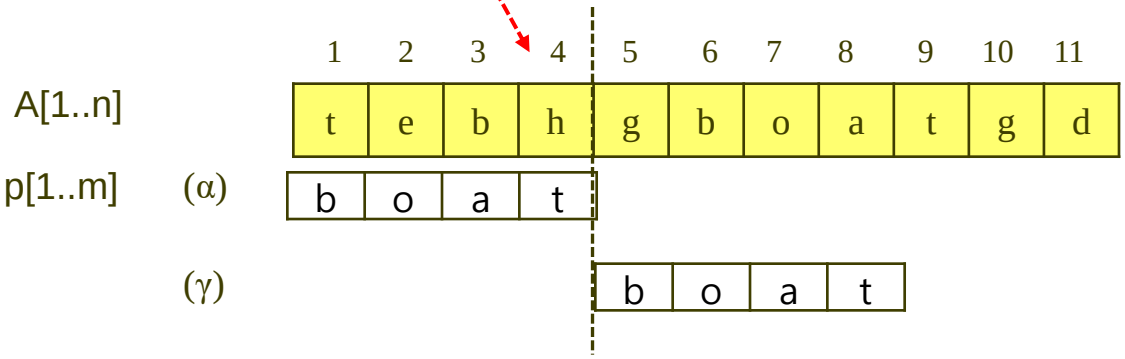
- ✓ (1)에서 $A[1] \neq p[1]$ $(t \neq b)$일 때 단순한 방법에서는 (2)로 이동
- ✓ 만일 $A[4] \neq p[4]$ 를 먼저 확인한다면, (1)에서 (3)으로 이동 가능
  - ❖ (이유) A[4]=h가 패턴 내에 없으므로
- ✓ 만일 패턴이 'show' 라면 (4)에서 비교해 본 후, 다음에는 (5) 로 이동. 6번째 칸(b와 w)부터 확인 시작.
- ✓ 패턴의 제일 뒤(오른쪽) 부터 검사

매칭 실패 시 이동 거리 표

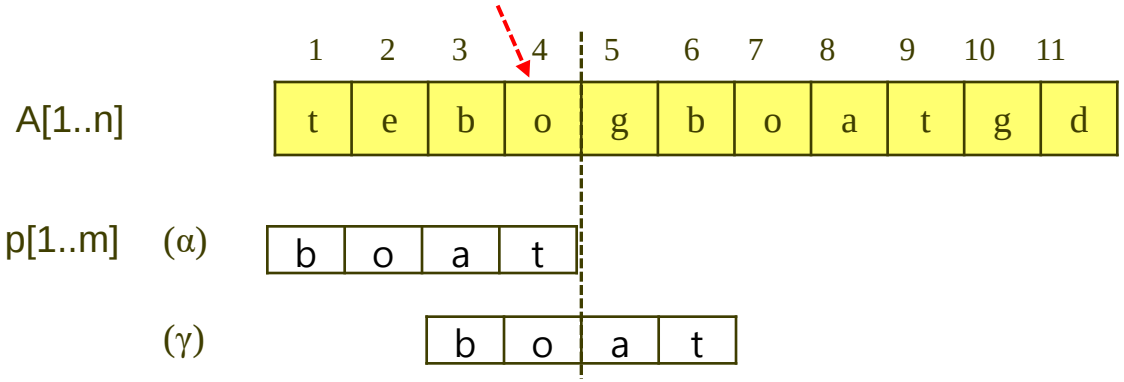
오른쪽끝문자*	b	o	a	t	etc
jump	3	2	1	4	4

* 입력문자열의 현재 매칭 확인하는 구역의 오른쪽 끝 문자

4칸 점프



2칸 점프



```
naiveMatching(A,p) {

  i = 1;
  while(i ≤ n-m+1) {
    if(p[1..m] == A[i..i+m-1])
        a matching is found at A[i]
    i=i+1
  }
}
```

```
BoyerMooreHorspool(A,p) {
  computeJump(p, jump);
  i = 1;
  while(i ≤ n-m+1) {
    j = m;
    k=i+m-1;
    while(j>0 and p[j]==A[k]){
       j--;
       k--;}
    if(j==0)
        a matching is found at A[i]
    i= i + jump[A[i+m-1]];
  }
}
```

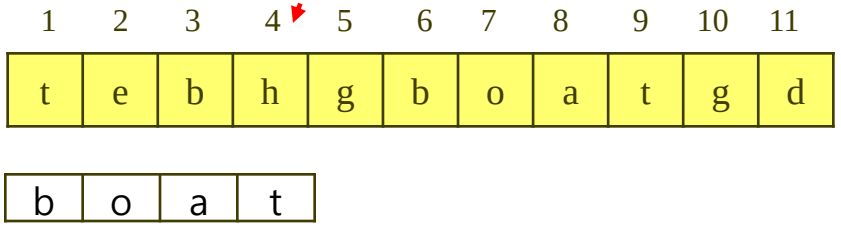
jump 정보를 계산 θ(m)

O(n-m+1)

O(m)

단순 방법

Boyer Moor Horspool 방법



- jump 정보 (입력문자열의 현재 매칭 확인하는 구역의 오른쪽 끝 문자에 따라)

오른쪽끝문자	b	o	a	t	etc
jump	3	2	1	4	4

오른쪽끝문자	s	h	o	o	l	etc
jump	4	3	2	1	5	5

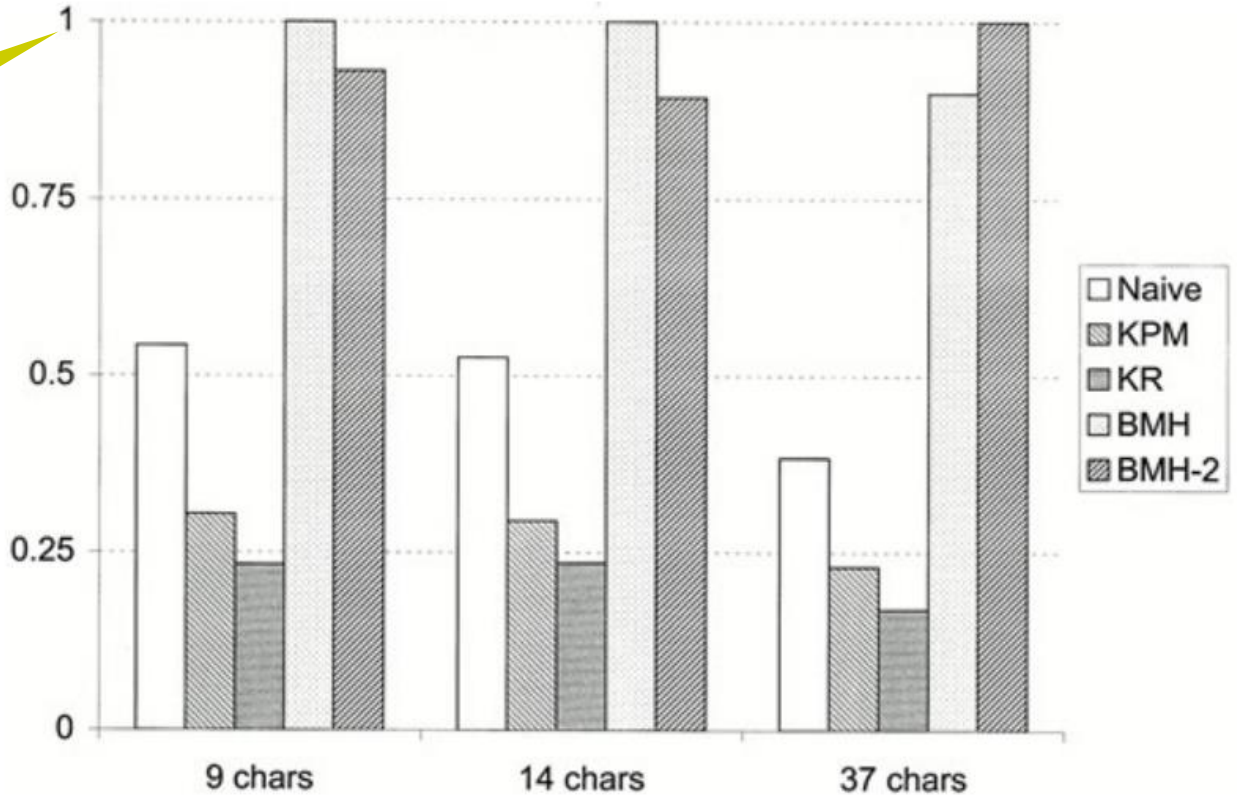
오른쪽끝문자	s	h	o	l	etc
jump	4	3	1	5	5

- ✓ 같은 문자가 있을 경우는 작은 jump 값으로 설정
- ✓ 총 O(mn)
- ✓ Boyer-Moore 방법은 이 방법보다 정교한 jump 판단 방식채용

- Naïve: 앞부터 하나씩 맞춤
- KPM: Knuth-Morris-Pratt(오토마타 이용)
- KR: Rabin Karp
- BMH: Boyer-Moore-Horspool
- BMH-2: BMH을 약간 변형
  - 보고 있는 문자열의 오른쪽끝, 가운데 문자로 판단

the number of characters
analyzed per millisecond

최대값을 1로
하였을 때



https://academic.oup.com/jamia/article/7/4/378/714050

목표: 좋은 알고리즘을 구축한다.

효과적인 알고리즘을 구축한다.

- 복잡도를 표현하는 방법
- 복잡도를 분석하는 방법

- divide and conquer algorithm
- dynamic programming
- greedy algorithm
- backtracking
- branch and bound algorithm
- computational complexity
  - ✓ sorting
  - ✓ searching


끝
수고하셨습니다.