자료구조

Chap 8. Search

2018년 1학기

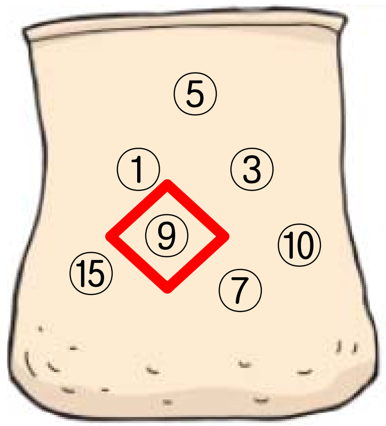
컴퓨터과학과
민경하

Contents

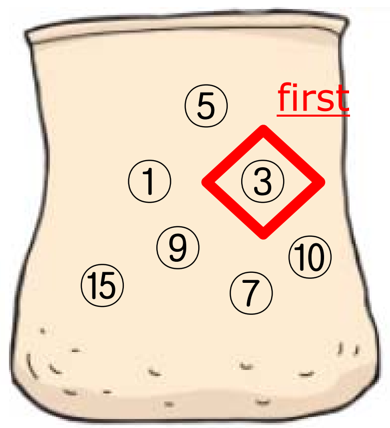
Contents

7.0 Introduction

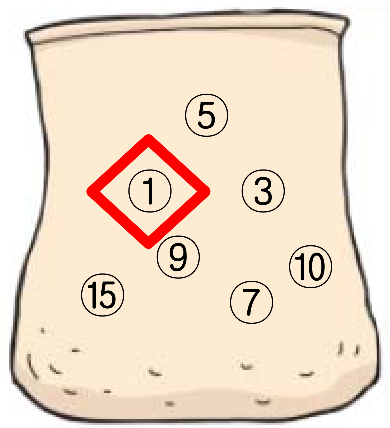
- 3 types of search
  - Find an **arbitrary** element in a given set
  - Find the **first / last** element in a give set
  - Find the **top** (maximum/minimum) element in a given set



Find **arbitrary**



Find **first / last**



Find **top**

7.0 Introduction

- Traversal
  - Find all the elements **reachable** from an element in a given set
  - Used in tree and graph
    - Depth first search = depth first traversal
    - Breadth first search = breadth first traversal
    - Inorder search = inorder traversal
    - Preorder search = preorder traversal
    - Postorder search = postorder traversal

7.0 Introduction

- All about search (1)

search	operation	Data structure	algorithm	Time complexity		
				search	insert	delete
Arbitrary	Find x	Unsorted array	Linear search	O(n)	O(1)	O(n)
		Sorted array	Linear search	O(n)	O(n)	
			Binary search	**O(log n)**		
			Interpolation search	**O(log n)**		
		Binary search tree	Search	**O(log n) / O(n)**	**O(log n) / O(n)**	**O(log n) / O(n)**
		Hash	Hashing	O(1)	O(1)	O(1)
Top	Find max/min	Unsorted array	Find max/min	O(n)	O(1)	O(n)
		Sorted array	Find max/min	O(1)	O(n)	
		Heap	Find max/min	**O(1)**	**O(log n)**	**O(log n)**
Arrival	Find first/ last	Stack	Pop	O(1)	O(1)	O(1)
		Queue	DeleteQ			

7.0 Introduction

- All about search (2)

Type	Data structure	Type of search	Algorithm	Time complexity		
				search	insert	delete
Linear	Unsorted array	Arbitrary	Linear search	O(n)	O(1)	O(n)
		Top	Find max/min			
	Sorted array	Arbitrary	Linear search	O(n)	O(n)	
			Binary search	**O(log n)**		
			Interpolation search	**O(log n)**		
		Top	Find max/min	**O(1)**		
	Stack	Arrival	Pop	O(1)	O(1)	O(1)
	Queue		Push			
Hiearchical	Binary search tree	Arbitrary	Search	**O(log n) / O(n)**	**O(log n) / O(n)**	**O(log n) / O(n)**
	Heap	Top	Find max/min	**O(1)**	**O(log n)**	**O(log n)**
Hash	Hash	Arbitrary	Hashing	**O(1)**	**O(1)**	**O(1)**

7.1 Search on linear data structure

- Linear data structure
  - data structure
    - A data structure whose elements are mapped by indices

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

    - List
    - Implementation
      - Consecutive VS separate
      - Array VS linked list
      - Index VS pointer

7.1 Search on linear data structure

(1) Linear search

- Definition:
  - Search an element by visiting the elements from the first one to the last one
- Next element to visit: (i+1)

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

- Working on unsorted list
- O(n)

7.1 Search on linear data structure

(2) Binary search

– Definition:

- Search an element by visiting the element in the center index (mid)

$$mid = \frac{(high + low)}{2} = low + \frac{(high - low)}{2}$$

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

low mid high

low: 1
high: 12

mid = 1 + (12-1)/2 = 6.5 ≈ 6

7.1 Search on linear data structure

(2) Binary search

- Example:
  - Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

  - Step 1. Search 150 from List[1..12]
    - mid = 1 + (12-1)/2 = 6.5 = 6
    - Compare List[6] & 150
    - If 150 > List[6] ➔ Search 150 from List[7..12]
    - Else ➔ Search 150 from List[1..5]

7.1 Search on linear data structure

(2) Binary search

– Example:

- Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

- Step 2. Search 150 from List[7..12]
  – mid = 7 + (12-7)/2 = 9.5 = 9
  – Compare List[9] & 150
  – If 150 > List[9] ➔ Search 150 from List[10..12]
  – Else ➔ Search 150 from List[7..8]

7.1 Search on linear data structure

- Binary search
  - Example:
    - Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

    - Step 3. Search 150 from List[10..12]
      - mid = 10 + (12-10)/2 = 11
      - Compare List[11] & 150
      - If 150 > List[11] ➔ Search 150 from List[12]
      - Else ➔ Search 150 from List[10]
      - If List[11] == 150 ➔ Bingo!!

7.1 Search on linear data structure

(2) Binary search

– Example:

- Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

(1) (2) (3)

– Problem of binary search?

7.1 Search on linear data structure

(2) Binary search

– Example: Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

(1) (2) (3)

– Problem of binary search?

- 150 (key to find) is closer to 170 (high) than to 20 (low)
  (|150 – 170| < |150 – 20|)
- Searching from high is more efficient than searching from mid or low

7.1 Search on linear data structure

(3) Interpolation search

– Search an element by visiting the element in the center index (mid)

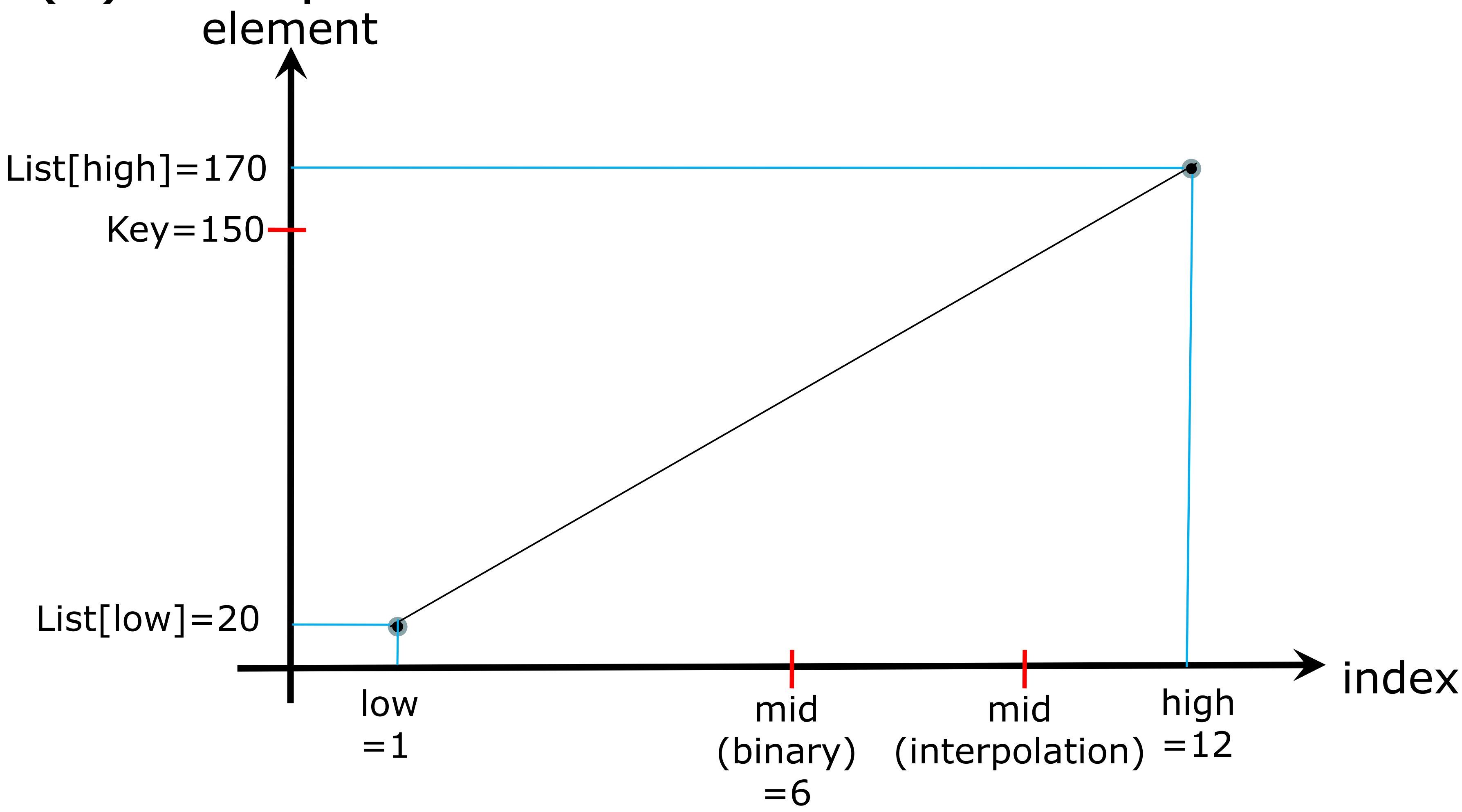
$$\text{mid} = \text{low} + \frac{(\text{Key} - \text{List[low]})}{\left(\text{List[high]} - \text{List[low]}\right)}(\text{high} - \text{low})$$

– In binary search,

$$\text{mid} = \text{low} + \frac{1}{2}(\text{high} - \text{low})$$

7.1 Search on linear data structure

(3) Interpolation search



7.1 Search on linear data structure

(3) Interpolation search

– Example: Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

$$\mathrm{mid} = \mathrm{low} + \frac{(\mathrm{Key} - \mathrm{List}[\mathrm{low}])}{(\mathrm{List}[\mathrm{high}] - \mathrm{List}[\mathrm{low}])}(\mathrm{high} - \mathrm{low})$$

- Step 1. Search 150 from List[1..12]
  - mid = 1+((150-20)/(170-20))*(12-1) = 10.5 = 10
  - Compare List[10] & 150
  - If 150 > List[10] ➔ Search 150 from List[11..12]
  - Else ➔ Search 150 from List[1..9]

7.1 Search on linear data structure

(3) Interpolation search

– Example: Search 150 from List[1..12]

index	1	2	3	4	5	6	7	8	9	10	11	12
element	20	42	55	62	78	92	112	132	140	145	150	170

$$\text{mid} = \text{low} + \frac{(\text{Key} - \text{List[low]})}{(\text{List[high]} - \text{List[low]})}(\text{high} - \text{low})$$

- Step 2. Search 150 from List[11..12]
  - mid = 11+((150-150)/(170-150))*(12-11) = 11
  - Compare List[11] & 150
  - If 150 > List[11] ➔ Search 150 from List[12]
  - Else ➔ None
  - If 150 == List[11] ➔ Bingo!!

7.1 Search on linear data structure

- Summary

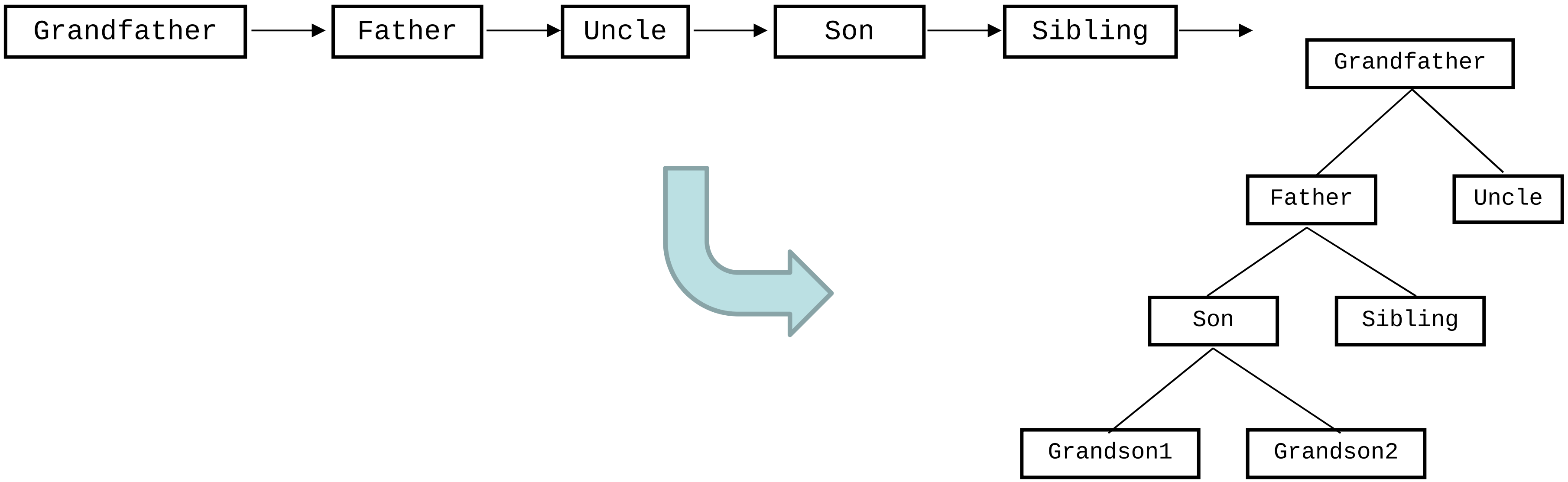
	Linear search	Binary search	Interpolation search
Next one to visit	i + 1	$\text{low} + \frac{1}{2}(\text{high} - \text{low})$	$\text{low} + \frac{(\text{Key} - \text{List}[\text{low}])}{(\text{List}[\text{high}] - \text{List}[\text{low}])}(\text{high} - \text{low})$
Time complexity	O(n)	O(log n)	Improved O(log n)
On sorted list?	No	Yes	Yes
Depends on implementation?	No	Yes	Yes

7.2 Search on hierarchical data structure

- Hierarchical data structure
  - Binary search tree
  - Heap
  - AVL tree
  - B+ tree

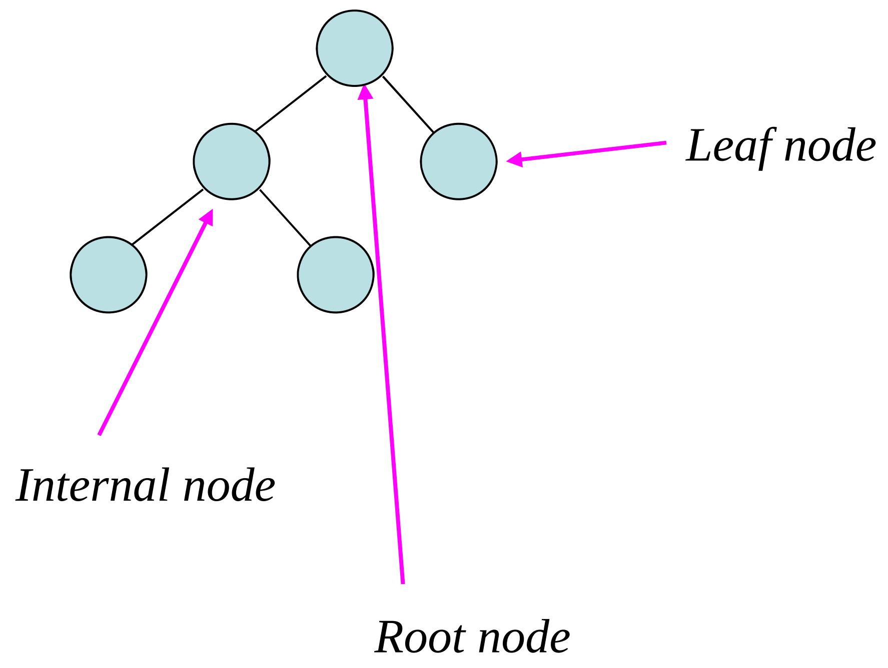
7.2 Search on hierarchical data structure

- Hierarchical data structure
  - Limitations of linear data structure?
    - Representation of “family record”
      - grandfather
      - father, uncle
      - son, sibling
      - grandson1, grandson2



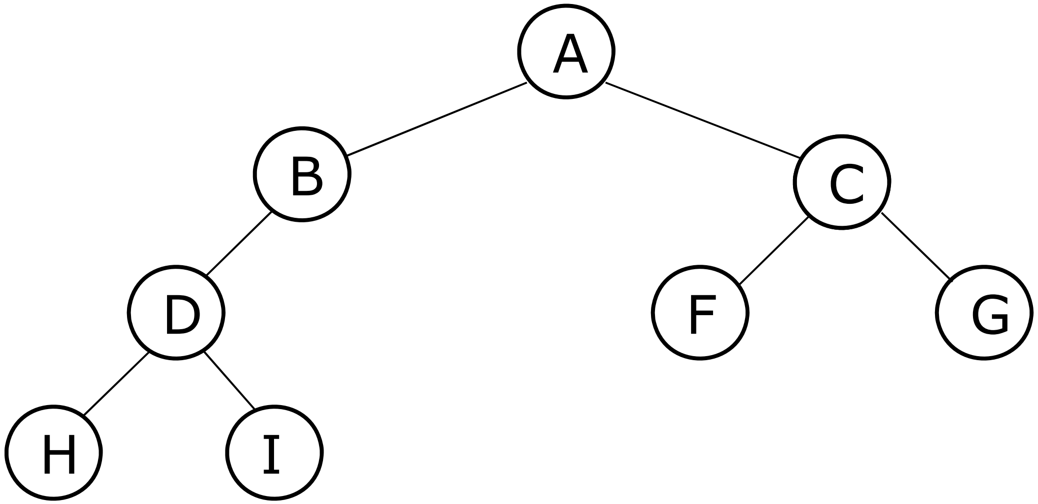
7.2 Search on hierarchical data structure

- Hierarchical data structure
  - Tree



7.2 Search on hierarchical data structure

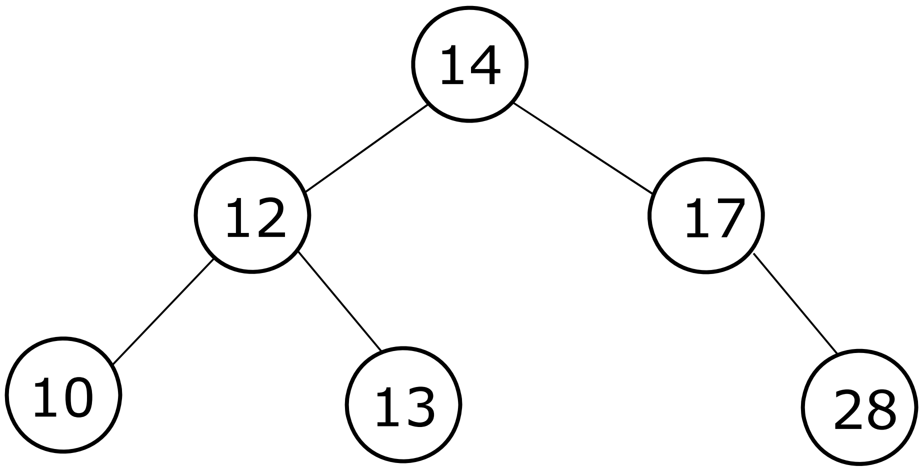
- Hierarchical data structure
  - Binary tree
    - A tree whose nodes have at most two child nodes



- Tree traversal algorithm: O(n)
  - Pre-order search
  - In-order search
  - Post-order search

7.2 Search on hierarchical data structure

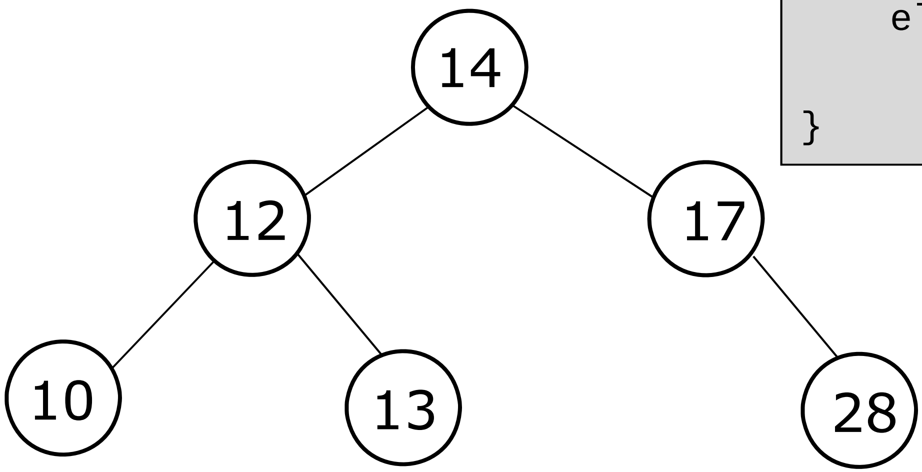
- Binary search tree (BST)
  - A binary tree (may be empty)
  - Satisfies the following properties
    - Each node has exactly one key, which is distinct
    - The keys in the left subtree < the key in root
    - The keys in the right subtree > the key in root
    - The left and right subtrees are also binary search tree



7.2 Search on hierarchical data structure

- Binary search tree (BST)
  - Search algorithm

```
element search ( BST root, KEY key )
{
    if ( !root )
        return NULL;
    if ( key == root->key )
        return root->data;
    if ( key < root->key )
        return search ( root->lchild, key );
    else
        return search ( root->rchild, key );
}
```



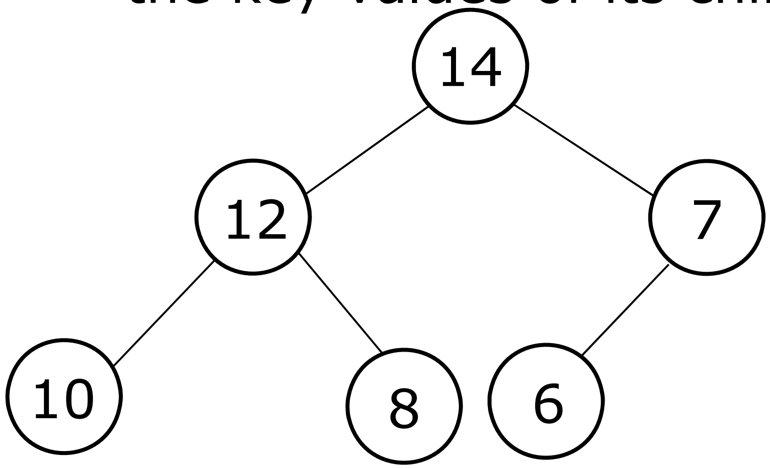
7.2 Search on hierarchical data structure

- Binary search tree (BST)
  - Time complexity?
    - O ($\log_2 n$)
  - Is it better than binary search on list?
  - Why binary search tree is better than sorted list?

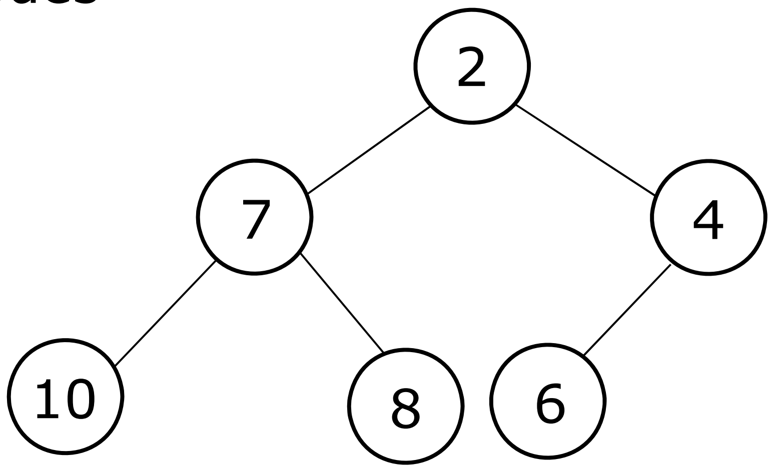
```
element search ( BST root, KEY key )
{
    if ( !root )
         return NULL;
    if ( key == root->key )
         return root->data;
    if ( key < root->key )
         return search ( root->lchild, key );
    else
         return search ( root->rchild, key );
}
```

7.2 Search on hierarchical data structure

- Heap
  - Priority queue
    - The element to be deleted is the one with the highest (or lowest) priority
  - A complete binary tree
  - Max heap (Min heap)
    - The key value in each node is no ***smaller (greater)*** than the key values of its child nodes



Max heap



Min heap

7.2 Search on hierarchical data structure

- Comparison of the data structures

	Sorted list	**BST**	**Heap**
What is it?	A list whose elements are sorted (array VS linked list)	A binary tree…	A complete binary tree…
Arbitrary	Sorted array: O (log n) Linked list: O (n)	**O (log n)**/O(n)	-
Top	Sorted array: O (1) Linked list: O(1)	O (log n)/O(n)	**O (1)**
Insert/Delete	Array: O (n) Linked list: O (n)	**O (log n)**/O(n)	**O (log n)**

7.2 Search on hierarchical data structure

- Balanced Binary Search Tree (BBST)
  - Self-balancing search tree
  - Height-balanced search tree

  - a search tree that attempts to keep its *height* as small as possible at all times, automatically

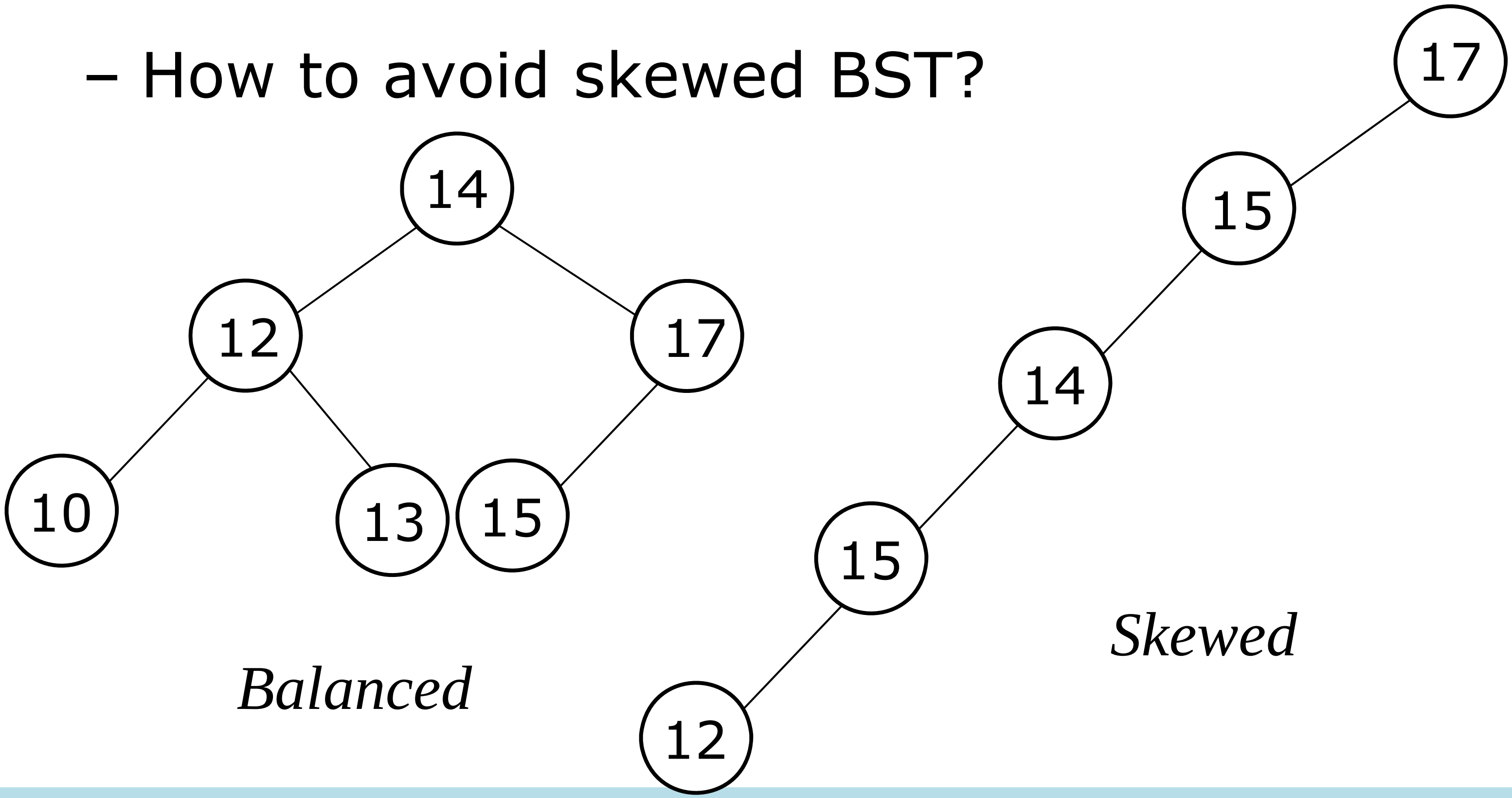
  (1) AVL tree
  (2) Red-black tree
  (3) 2-3 tree
  (4) B+ tree

7.2 Search on hierarchical data structure

(1) AVL tree

- Weakness of binary search tree
  - On a skewed BST, all operations take O (n)

- How to avoid skewed BST?



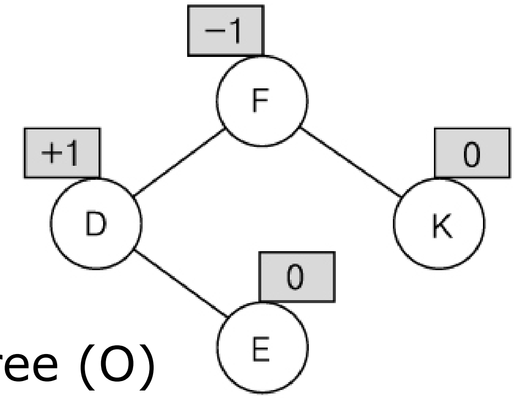
7.2 Search on hierarchical data structure

(1) AVL tree

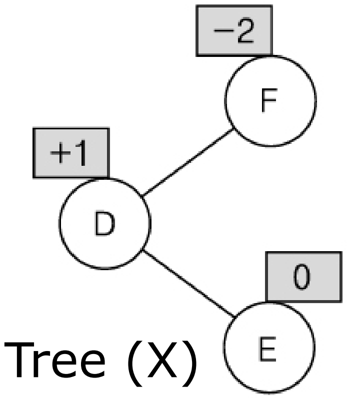
- G. M. Adelson-Velskii & E. M. Landis

- the heights of the two child subtrees of any node differ by at most one

- Balance factor
  - = Height of right subtree – Height of left subtree



AVL Tree (O)

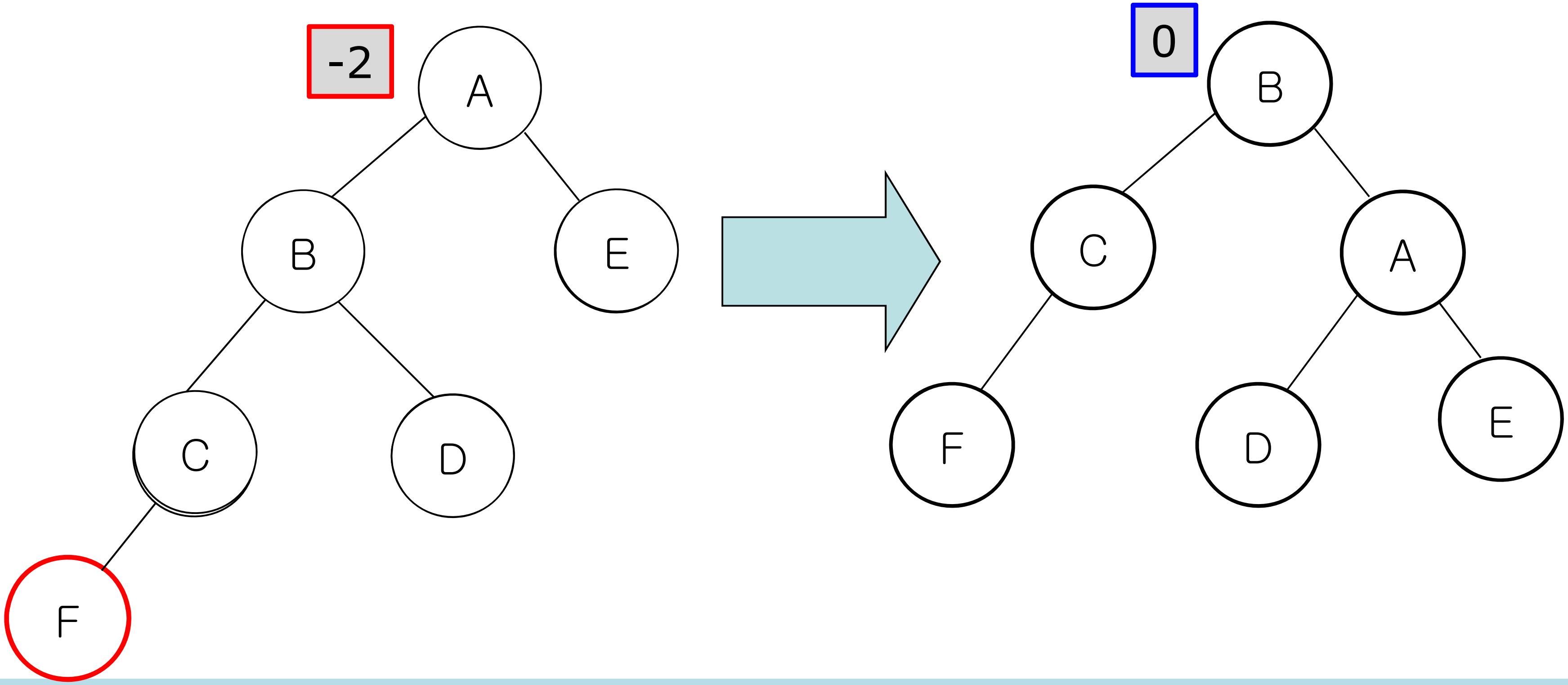


AVL Tree (X)

7.2 Search on hierarchical data structure

(1) AVL tree

- After insertion/deletion, balance can be broken
- Using balancing operations, the tree is modified to keep the balance



7.2 Search on hierarchical data structure

(1) AVL tree

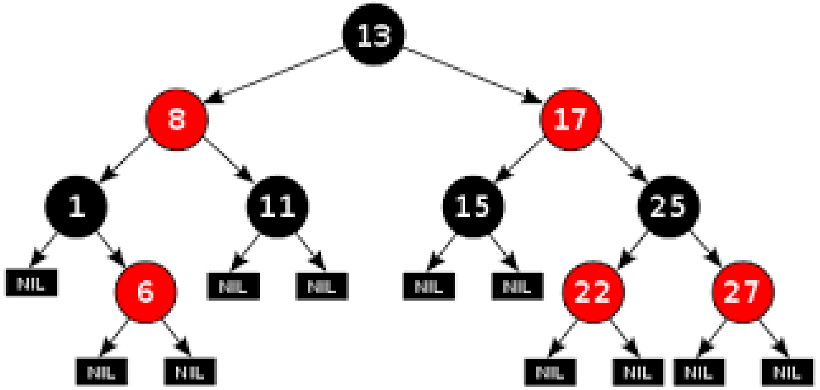
- Balancing operations
  - Single left rotate
  - Single right rotate

  - Double left rotate
  - Double right rotate

7.2 Search on hierarchical data structure

(2) Red-black tree

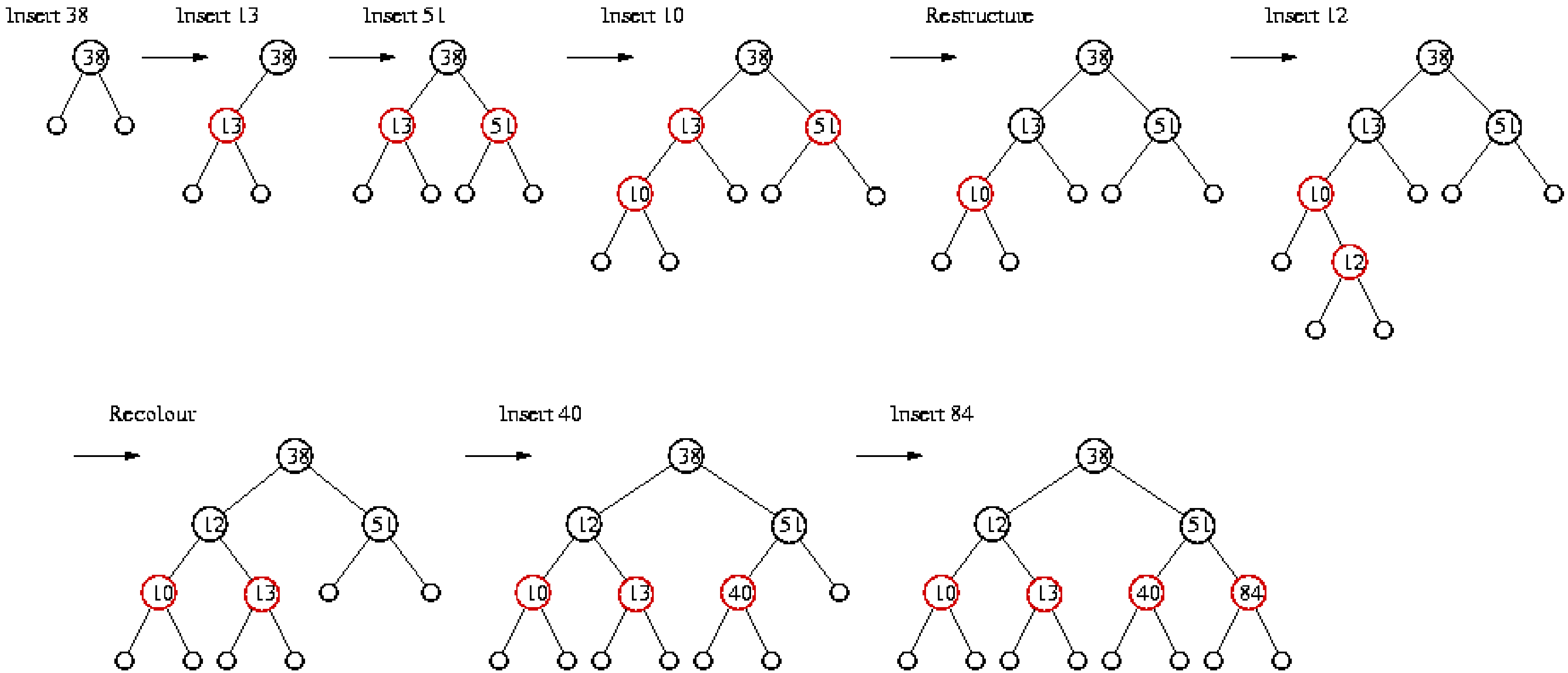
- L. J. Guibas and R. Sedgewick (1978)
- Five properties
  - Each node is either red or black
  - The root is black
  - All leaf nodes (NIL node) are black
  - If a node is red, then its both child nodes are black
  - Every path from a given node to all of its descendent NIL nodes contain the same number of black nodes



7.2 Search on hierarchical data structure

(2) Red-black tree

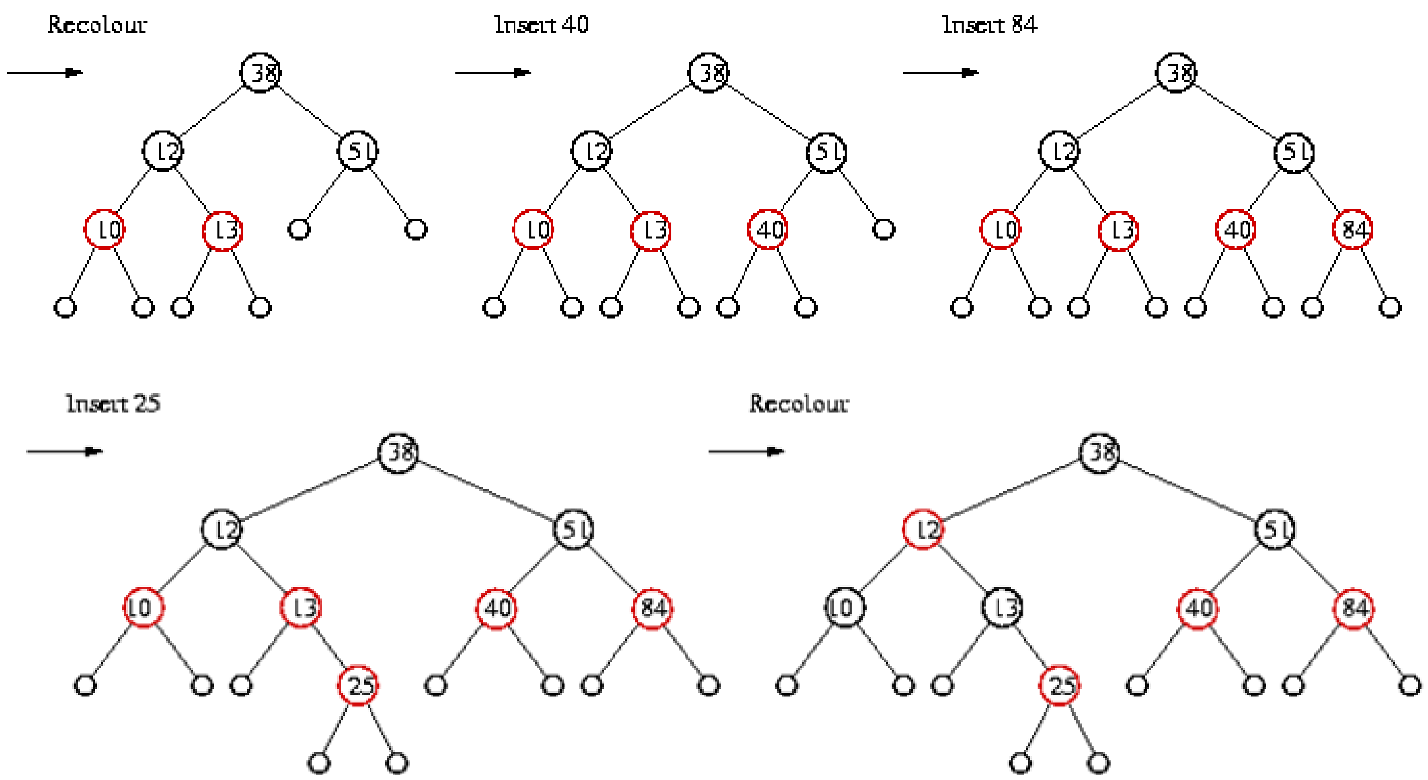
Insertions: 38, 13, 51, 10, 12, 40, 84, 25



7.2 Search on hierarchical data structure

(2) Red-black tree

Insertions: 38, 13, 51, 10, 12, 40, 84, 25

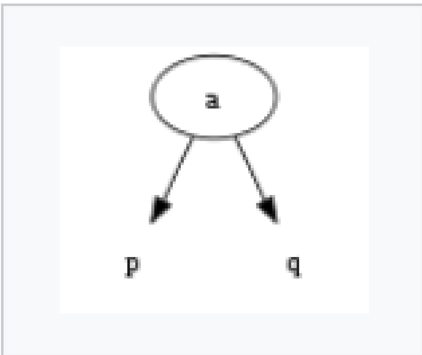


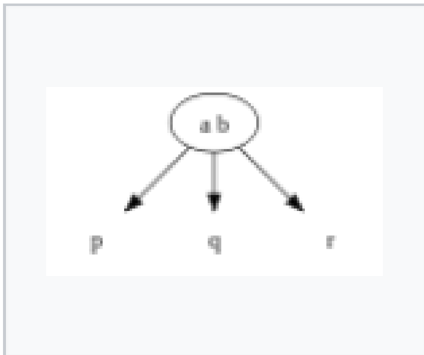
7.2 Search on hierarchical data structure

(3) 2-3 tree

- J. Hopcroft (1970)

- A search tree whose nodes are either 2-node or 3-node.
  - 2-node
    - One key value with two child nodes
  - 3-node
    - Two key values with three child nodes

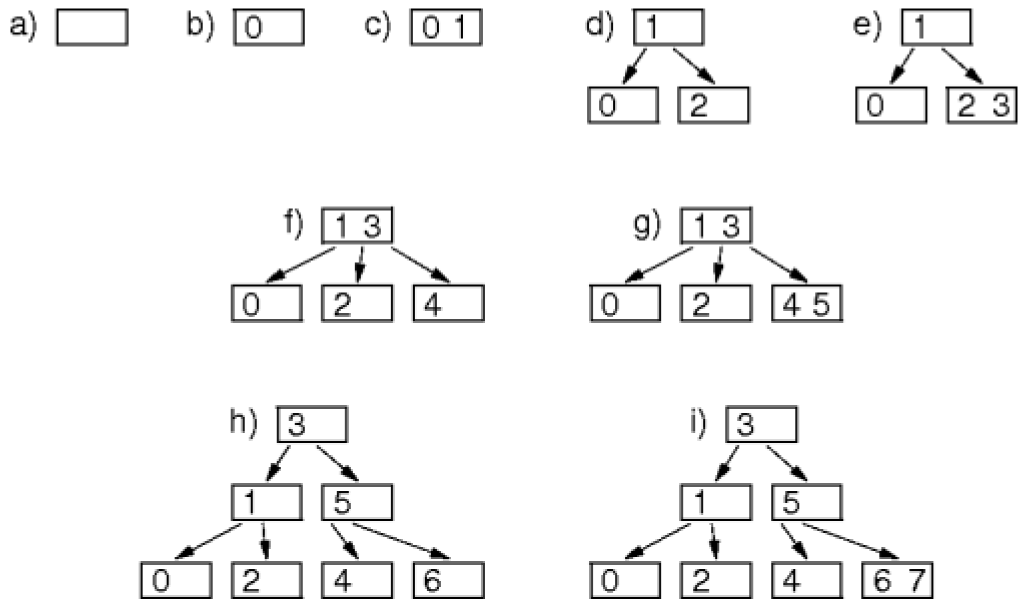




7.2 Search on hierarchical data structure

(3) 2-3 tree

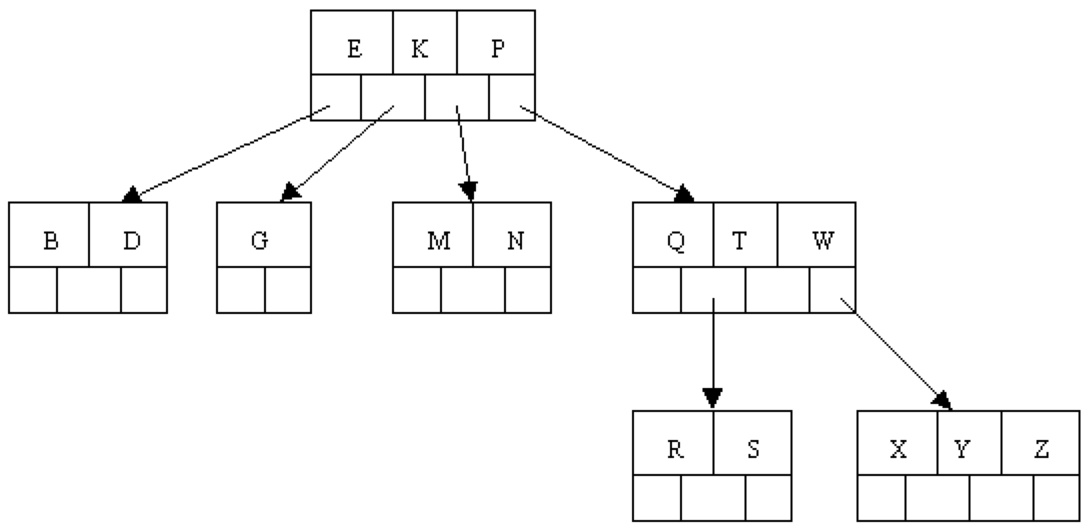
– Insert 0, 1, 2, 3, 4, 5, 6, 7



7.2 Search on hierarchical data structure

(4) B+ tree

- A multi-way tree
  - Each node stores
    - k keys
    - (k+1) subtrees
- Time complexity: O ($\log_k$ n)



7.3 Search on graph

- Travesal

- Find a vertex of a graph that contains the key element

  - Depth-first traversal

  - Breadth-first traversal

7.4 Hashing

- Why hashing?
  - 이진 탐색
    - O(log n)의 탐색 시간
    - 자료의 크기에 따라서 검색 시간이 결정됨
  - 자료의 크기에 상관없이 실시간에 탐색이 수행되어야 하는 경우가 있음
  - O(1)의 탐색 시간을 보장하는 탐색 알고리즘

7.4 Hashing

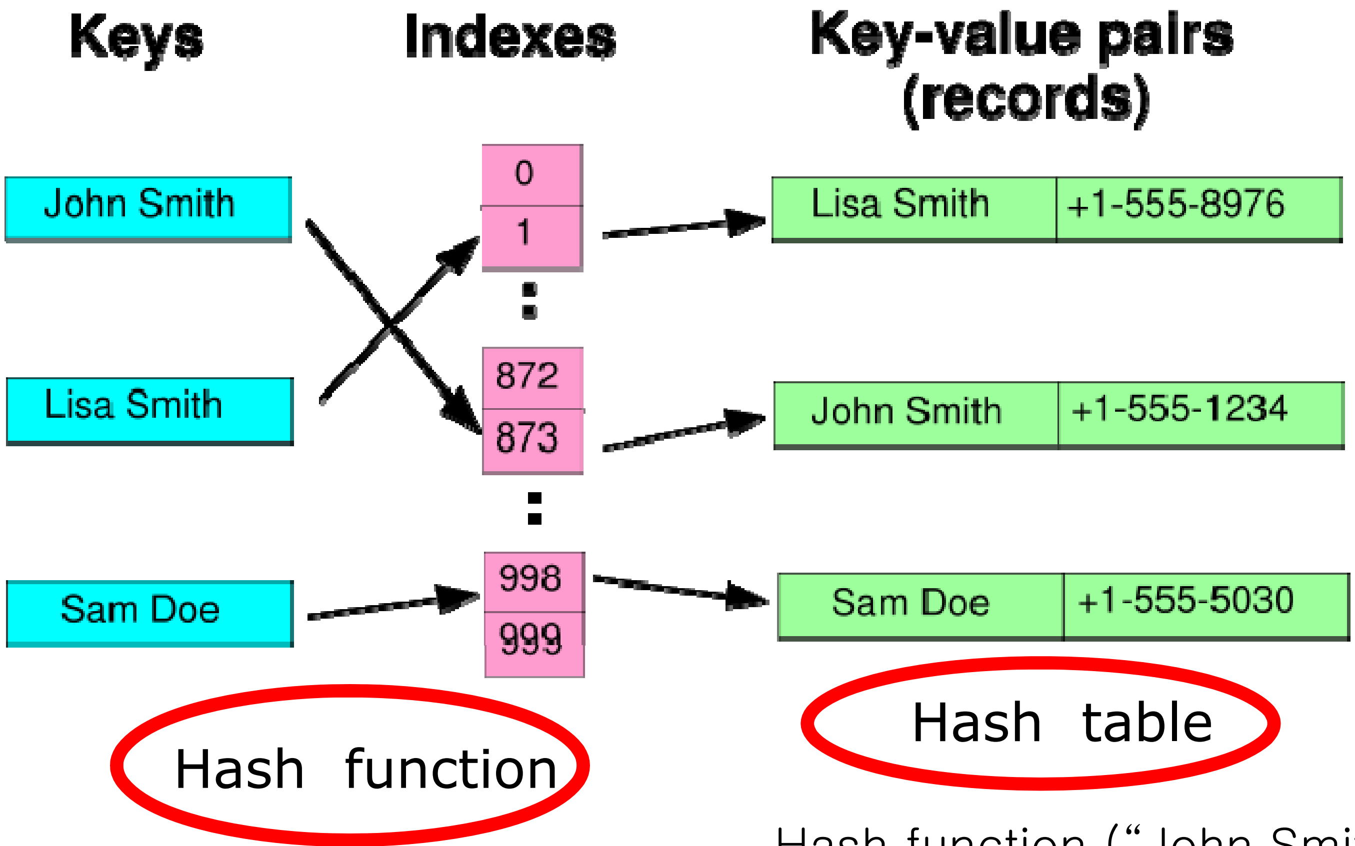
- Definition

“모든 키의 레코드를 산술 연산에 의해 한 번에 바로 접근할 수 있는 기법”

  - Hash function
  - Hash index
  - Hash table
  - Collision & collision resolution

7.4 Hashing

- Key concept of hashing



Hash function (“John Smith”) = 873
Hash function (“Sam Doe”) = 998

7.4 Hashing

- Hash function (1)
  - ① 자릿수 선택 (digit selection)
    - 키의 값 중에서 일부 자릿수만 골라내서 인덱스를 생성하는 함수
    - 예
      - 13자리 주민등록번호 중 홀수 자릿수만 선택

        h(8812152051218) = 8112528

    - 충돌이 발생하는 경우는?

        h(8812152051218) = 8112528

        h(8711142152238) = 8112528

7.4 Hashing

- Hash function (2)
  ② 자릿수 접기 (digit folding)
    - 키의 각각의 자릿수를 더해서 인덱스를 생성하는 함수
    - 예

      h(8812152051218) = 8 + 8 + 1 + ... + 8 = 44

    - 충돌이 발생하는 경우는?

      h(8812152051218) =
      8+8+1+2+1+5+2+0+5+1+2+1+8=44

      h(8713142152208) =
      8+7+1+3+1+4+2+1+5+2+2+0+8=44

7.4 Hashing

- Hash function (3)
  ③ 모듈로 연산 (modulo function)
  - 키를 해쉬 테이블의 크기로 나눈 나머지를 인덱스로 생성하는 함수
  - h(KEY) = KEY mod TableSize
  - 충돌이 발생하는 경우는?
    h(8812152051218) = 8812152051218%100 = 18

    h(8713142152218) = 8713142152218%100 = 18

7.4 Hashing

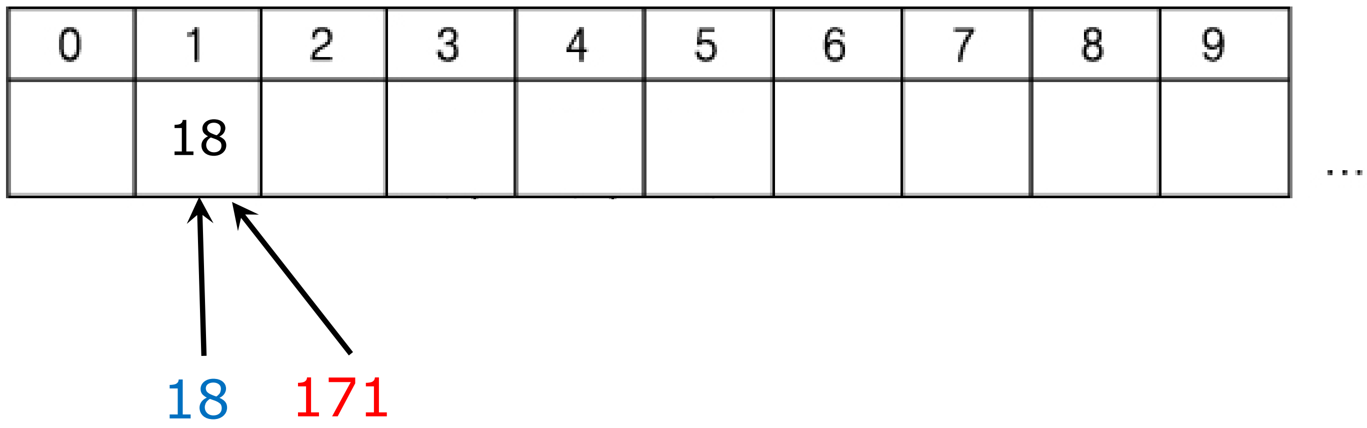
- 충돌 (collision)
  - 서로 다른 키를 가진 레코드가 해쉬 함수에 의해서 동일한 인덱스로 대응되는 현상
  - 예
    - Hash function:

      $H(K) = K \bmod m$, where $m = 17$

    - For $K_1 = 18$ & $K_2 = 171$,
      - $H(K_1) = 18 \bmod 17 = 1$
      - $H(K_2) = 171 \bmod 17 = 1$
    - Collision !!!

7.4 Hashing

- 충돌
  - 예
    - K_1 = 18 & K_2 = 171,
      - $H(K_1)$ = 18 mod 17 = 1
      - $H(K_2)$ = 171 mod 17 = 1



7.4 Hashing

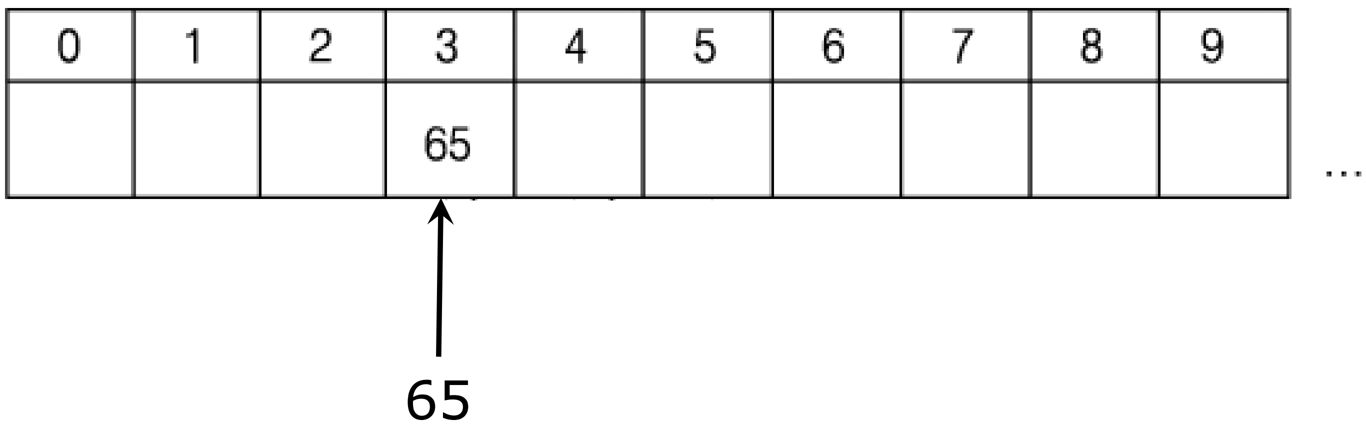
- 충돌 해소
  - 충돌이 발생한 레코드를 다른 주소에 저장하는 기법
  - 열린 어드레싱 (open addressing)
    - 충돌이 발생한 레코드를 저장할 다른 주소를 찾는 방법
    - 레코드의 주소 (address)가 변경될 수 있기 때문에 열린 (open) 어드레싱이라고 함
    - 선형 탐사 (linear probing)
    - 제곱 탐사 (quadratic probing)
    - 이중 해시 (double hash)
  - 닫힌 어드레싱 (closed addressing)
    - 충돌이 발생한 레코드를 동일한 주소에 저장하는 방법
    - 주소가 변경되지 않으므로 닫힌 (closed) 어드레싱이라고 함
    - 버켓 (bucket)
    - 별도 체인 (separate chain)

7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 충돌이 일어나면 그 다음 빈 곳에 원소를 저장함
    - T[h(KEY)]가 점유되어 있을 때,
      - T[h(KEY) + 1]
      - T[h(KEY) + 2]
      - ...
    - 배열의 끝을 만나면 다시 처음으로 되돌아와서 거기서부터 빈자리를 찾음
      - T[n-1]
      - T[0]

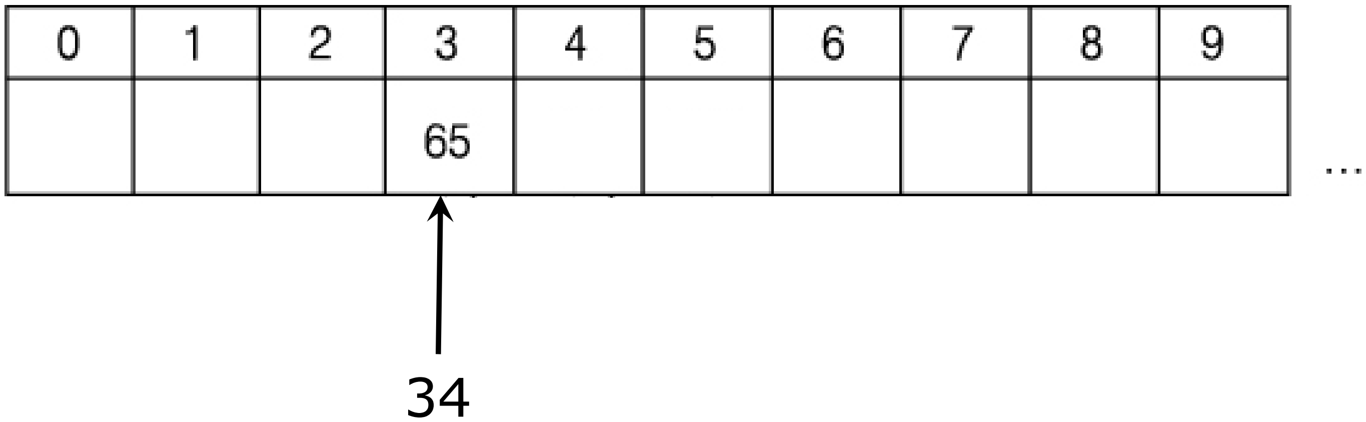
7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 예: h(x) = x % 31



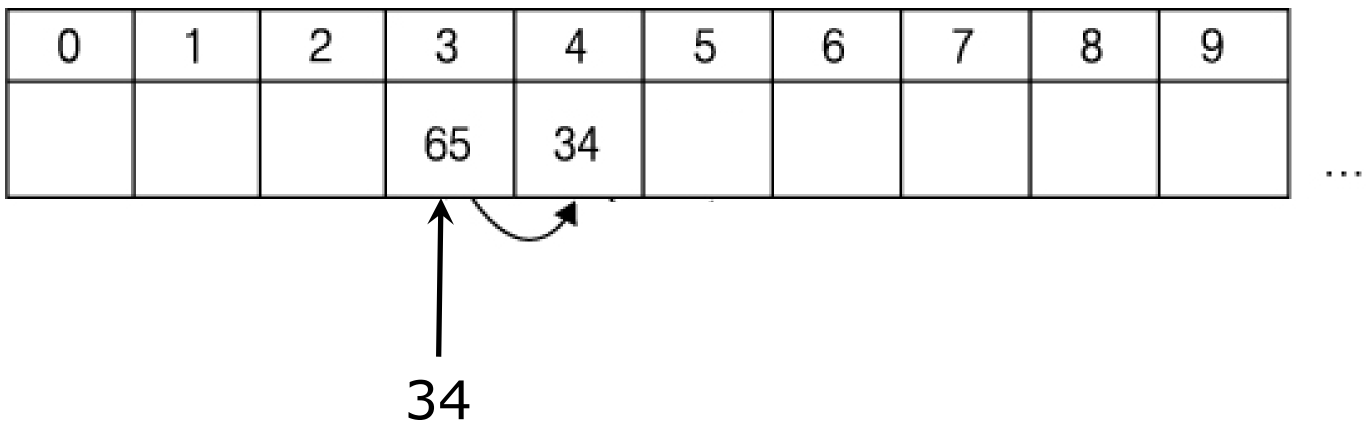
7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 예: h(x) = x % 31
    - 65를 삽입 ➔ 65%31 = 3
    - 34를 삽입 ➔ 34%31 = 3



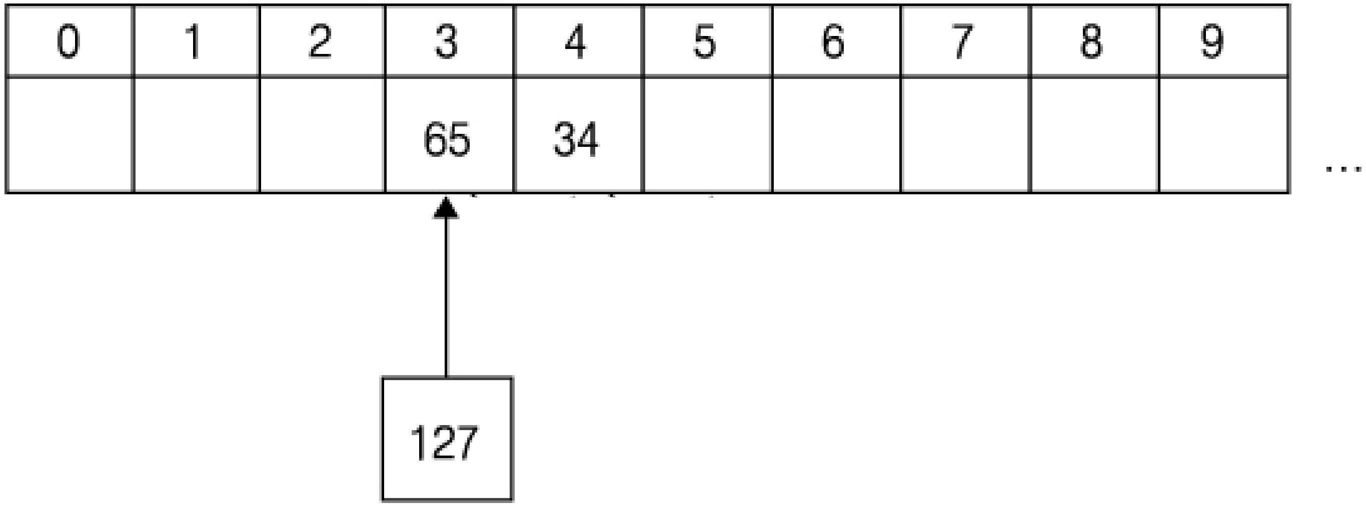
7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 예: h(x) = x % 31
    - 65를 삽입 ➔ 65%31 = 3
    - 34를 삽입 ➔ 34%31 = 3



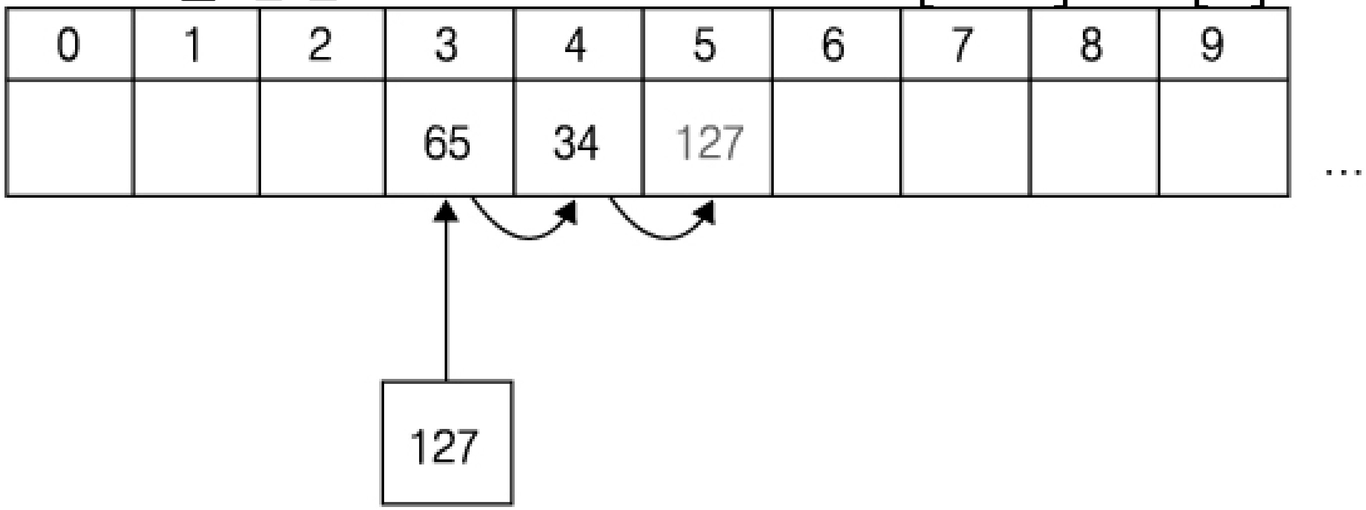
7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 예: h(x) = x % 31
    - 65를 삽입 ➔ 65%31 = 3
    - 34를 삽입 ➔ 34%31 = 3
    - 127을 삽입 ➔ 127%31 = 3



7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 예: h(x) = x % 31
    - 65를 삽입 ➔ 65%31 = 3
    - 34를 삽입 ➔ 34%31 = 3 ➔ T[3+1] = T[4]
    - 127을 삽입 ➔ 127%31 = 3 ➔ T[3+2] = T[5]



7.4 Hashing

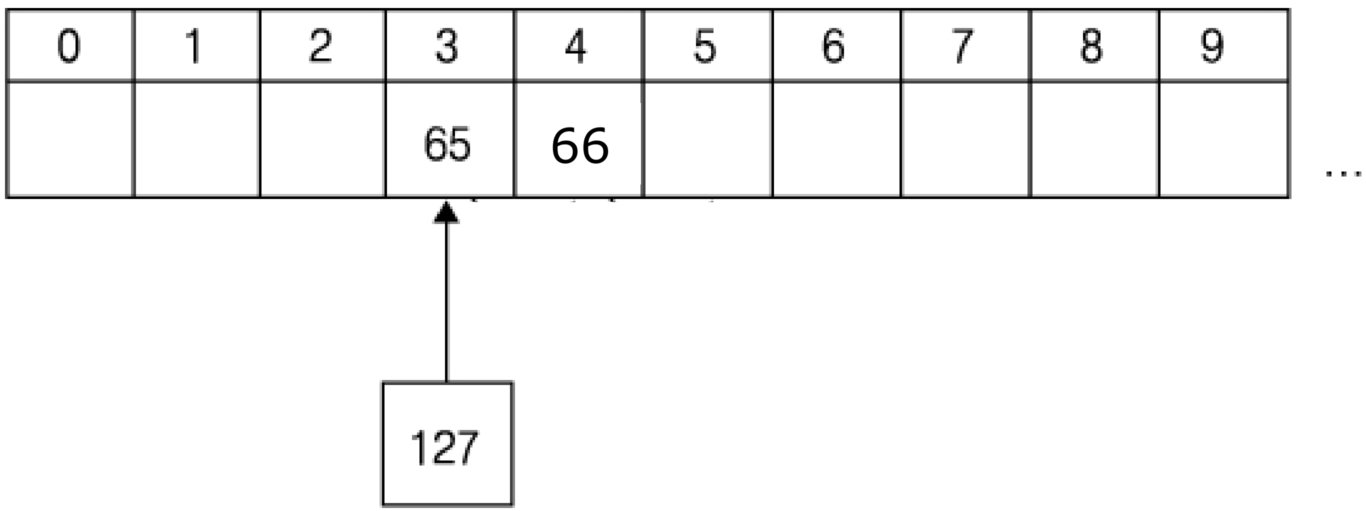
- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 태그 (tag)
      - – 선형 탐사에서 다음 원소를 가리키는 포인터

0	1	2	3	4	5	6	7	8	9
			65	34	127				

…

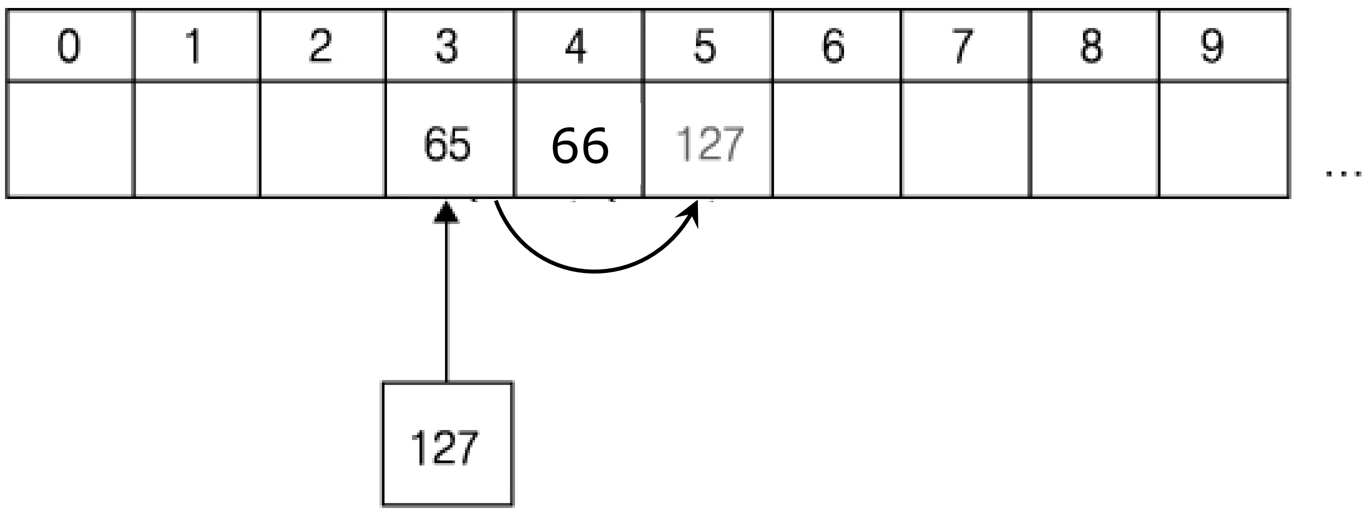
7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 태그 (tag)의 필요성 (1)
      - – 다음 자리 (h(key)+1)에 이미 다른 원소가 존재할 때



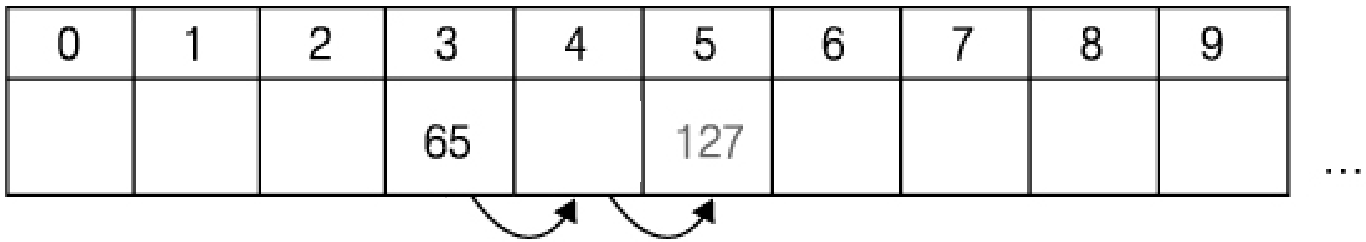
7.4 Hashing

- 열린 어드레싱 (1)
  ① 선형 탐사 (linear probing)
  - 태그 (tag)의 필요성 (1)
    - 다음 자리 (h(key)+1)에 이미 다른 원소가 존재할 때
    - 다음 원소의 위치를 가리킴



7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 태그 (tag)의 필요성 (2)
      - – 중간의 원소를 삭제할 때



7.4 Hashing

- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 태그 (tag)의 필요성 (2)
      - 중간의 원소를 삭제할 때
      - 태그를 수정함

0	1	2	3	4	5	6	7	8	9
			65		127				

…

7.4 Hashing

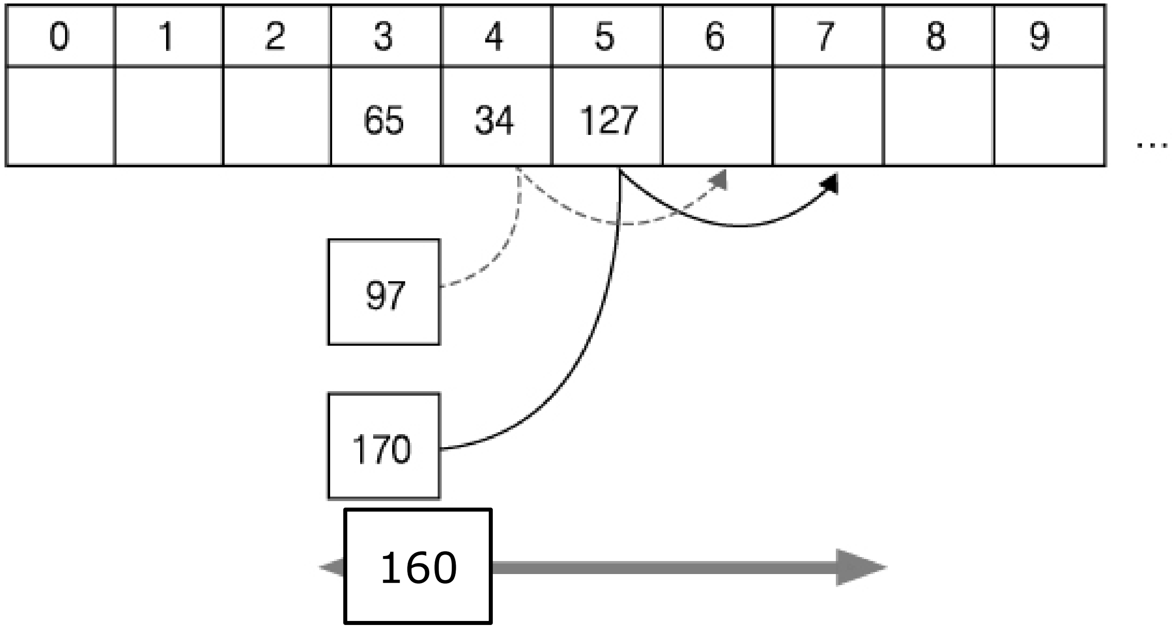
- 열린 어드레싱 (1)
  - ① 선형 탐사 (linear probing)
    - 단점: 클러스터링 (clustering)
      - – 레코드가 분산되지 않고 군집되는 현상

0	1	2	3	4	5	6	7	8	9
			65	34	127				

…

7.4 Hashing

- 열린 어드레싱 (1)
  ① 선형 탐사 (linear probing)
  - 단점: 클러스터링 (clustering)
    – 예: 97을 삽입: 97%31 = 4
    – 예: 160을 삽입: 160%31 = 5

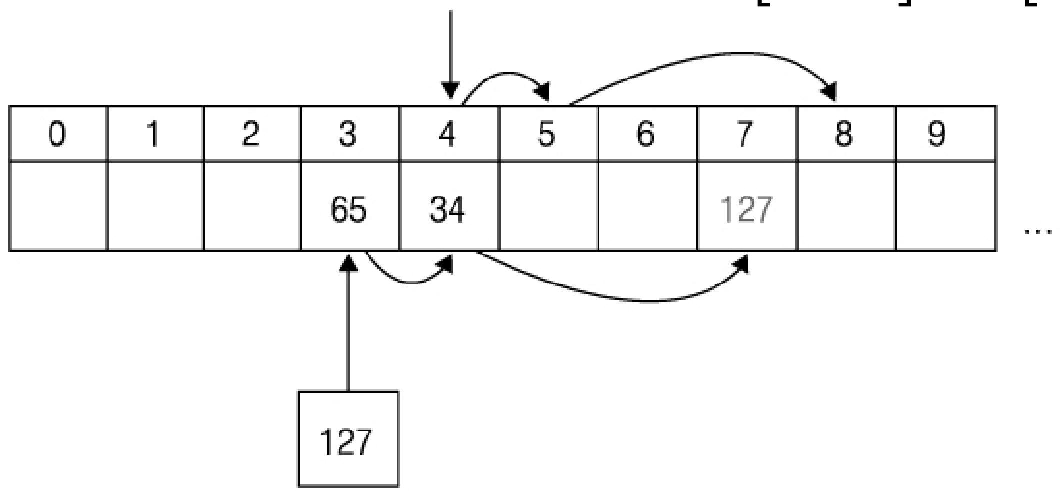


7.4 Hashing

- 열린 어드레싱 (2)
  - ② 제곱 탐사 (quadratic probing)
    - 충돌이 일어날 때 바로 그 뒤에 넣지 않고 조금 간격을 두고 삽입
    - T[h(KEY)]가 점유되어 있을 때,
      - T[h(KEY) + 1]
      - T[h(KEY) + 2^2]
      - ...

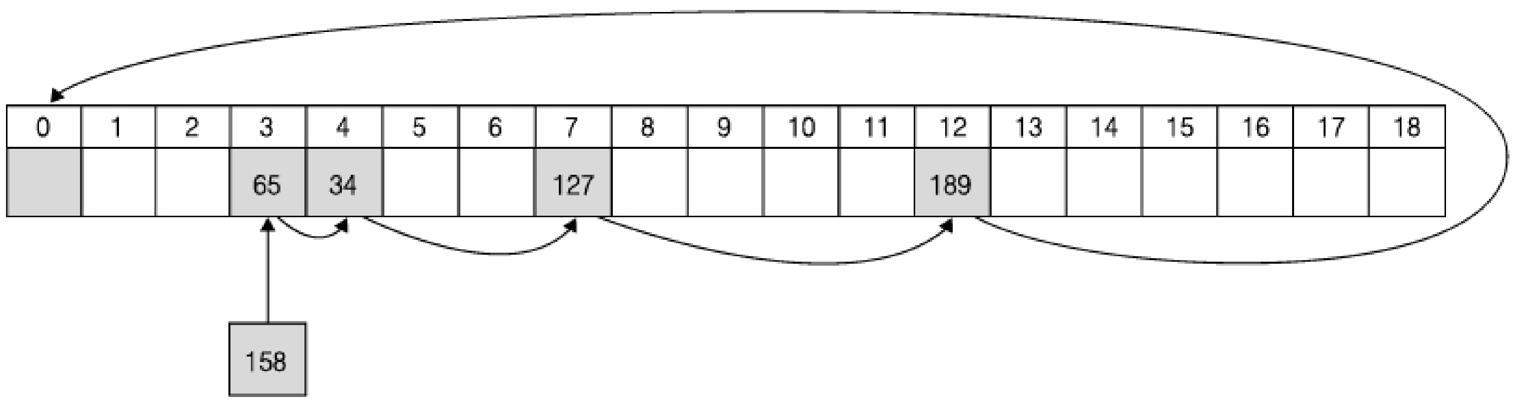
7.4 Hashing

- 열린 어드레싱 (2)
  - ② 제곱 탐사 (quadratic probing)
    - 충돌이 일어날 때 바로 그 뒤에 넣지 않고 조금 간격을 두고 삽입
      - 예: h(x) = x % 31
      - 65를 삽입 ➔ 65%31 = 3
      - 34를 삽입 ➔ 34%31 = 3 ➔ T[3+1] = T[4]
      - 127을 삽입 ➔ 127%31 = 3 ➔ T[3+2^2] = T[7]



7.4 Hashing

- 열린 어드레싱 (2)
  ② 제곱 탐사 (quadratic probing)
  - 클러스터링을 피할 수 있는가?
    - 연속적인 배열을 피할 수 있음
    - 동일한 키를 가진 레코드는 동일한 자리에 삽입됨
    - 예: 158의 경우, 65, 34, 127, 189 다음의 자리에 삽입
    - 궁극적으로는 클러스터링을 피할 수 없음
      ➔ 2차 클러스터링



7.4 Hashing

- 열린 어드레싱 (3)
  - ③ 이중 해쉬 (double hash)
    - 제곱탐사의 단점인 2차 클러스터를 방지
    - 두 개의 해쉬 함수 h_1, h_2를 사용
      - h_1은 주어진 키로부터 인덱스를 계산하는 해쉬 함수
      - h_2는 충돌 시 탐색할 인덱스의 간격(Step Size)을 의미

7.4 Hashing

- 열린 어드레싱 (3)
  - ③ 이중 해쉬 (double hash)
    - 예
      - h_1 = KEY % 13
      - h_2 = 1 + KEY % 11
    - 14를 삽입 (KEY = 14)
      - h_1 = 14 % 13 = 1 ➔Collision

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72		15		50	

14

7.4 Hashing

- 열린 어드레싱 (3)
  ③ 이중 해쉬 (double hash)
  - 예
    – h_1 = KEY % 13
    – h_2 = 1 + KEY % 11
  - 14를 삽입 (KEY = 14)
    – h_1 = 14 % 13 = 1 ➔Collision
    – h_2 = 1 + 14% 11 = 4 ➔ 4칸씩 건너뛰면서 삽입

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72		15		50	

14

7.4 Hashing

- 열린 어드레싱 (3)
  - ③ 이중 해쉬 (double hash)
    - 예
      - h_1 = KEY % 13
      - h_2 = 1 + KEY % 11
    - 14를 삽입 (KEY = 14)
      - h_1 = 14 % 13 = 1 ➔Collision
      - h_2 = 1 + 14% 11 = 4 ➔ 4칸씩 건너뛰면서 삽입

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72		15		50	

14

7.4 Hashing

- 열린 어드레싱 (3)

  ③ 이중 해쉬 (double hash)

  - 예
    - h_1 = KEY % 13
    - h_2 = 1 + KEY % 11
  - 14를 삽입 (KEY = 14)
    - h_1 = 14 % 13 = 1 ➔Collision
    - h_2 = 1 + 14% 11 = 4 ➔ 4칸씩 건너뛰면서 삽입

0	1	2	3	4	5	6	7	8	9	10	11	12
14	79			69	98		72		15		50	

7.4 Hashing

- 닫힌 어드레싱 (1)

  ① 버켓 (bucket)

  - 해쉬 테이블의 각 원소들이 다시 여러 개의 원소로 이루어짐
  - 2차원 배열
  - 충돌되는 레코드를 하나의 인덱스에 둠
  - 사용되지 않는 배열 공간이 낭비됨

3					4				
65	34	127	189						

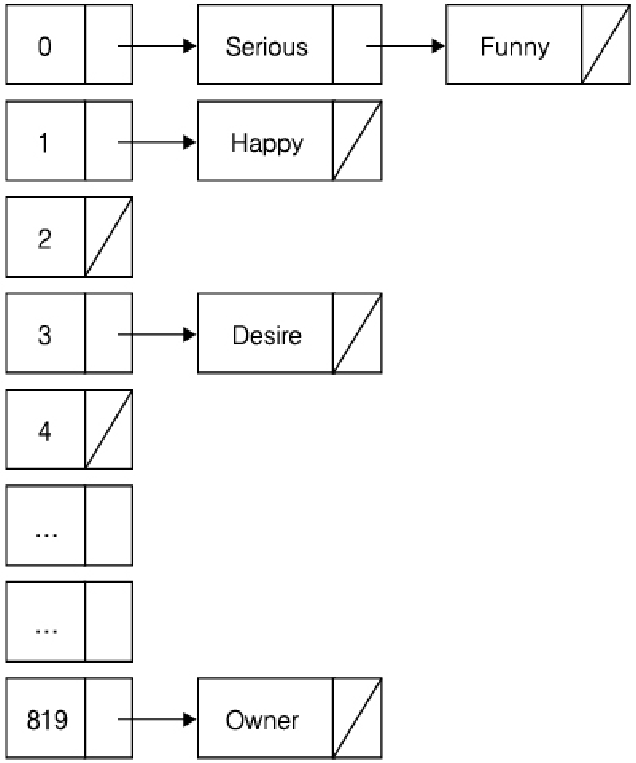
…

7.4 Hashing

- 닫힌 어드레싱 (2)
  - ② 별도 체인 (separate chain)
    - 해쉬 테이블의 각 원소들가 연결 리스트를 가리키는 헤드
    - 충돌이 일어날 때마다 해당 레코드를 연결 리스트의 첫 위치에 삽입
    - 동적 구조(Dynamic Structure)

7.4 Hashing

- 닫힌 어드레싱 (2)
  - ② 별도 체인 (separate chain)



키	해시 값
Funny	0
Happy	1
Serious	0
Owner	819
Desire	3
...	...

7.5 Heuristic search

- Graph search algorithm
  - Find a path from a source node to a target node by estimating the best route
  - Depends on the heuristic
    - Find the best path according to the heuristic
    - After a failure, it searches an alternative
    - An evaluation function quantifies the heuristic to find the best path

- A* search
  - $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$
- Game programming

Contents