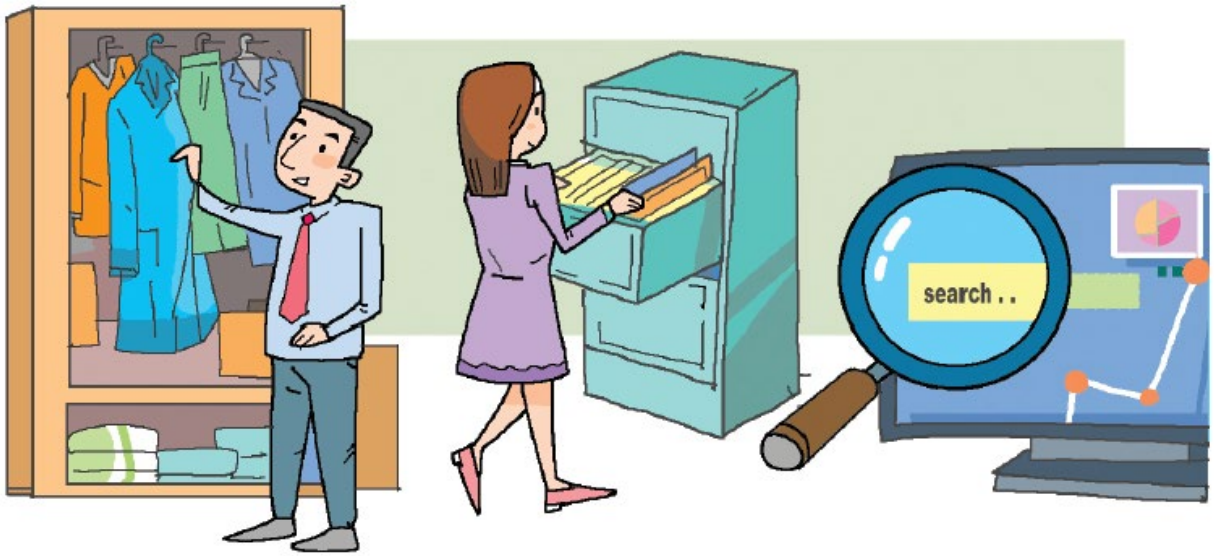
탐색

Search

탐색

탐색이란?

- 여러 자료 중 원하는 자료를 찾는 작업
- 컴퓨터가 가장 많이 하는 작업 중 하나
- 탐색을 효율적으로 수행하는 것은 매우 중요



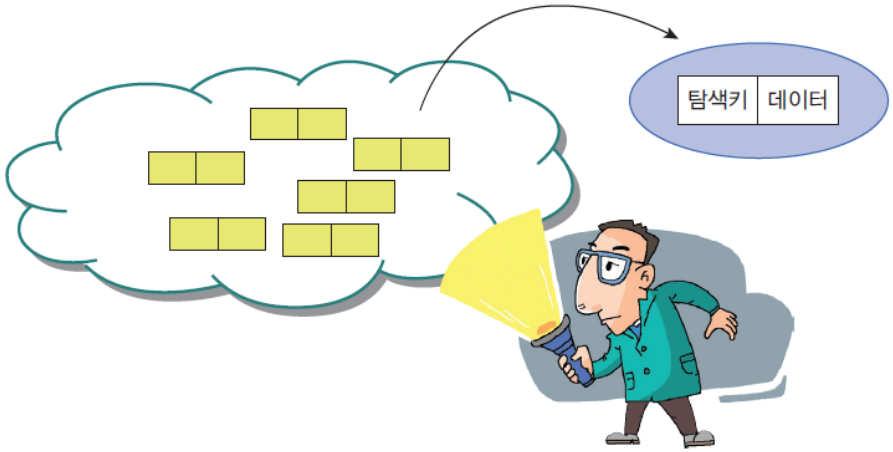
탐색

탐색을 위해?

- 구분할 수 있는 항목 필요 (단일 검색을 위해선)

- 탐색 키(search key)
  - 항목과 항목을 구별해주는 키(key)

- 탐색을 위하여 사용되는 자료 구조
  - 배열, 연결 리스트, 트리, 그래프 등 현재까지 배운 자료 구조는 거의 모두 사용



탐색

순차 탐색(sequential search)

- 탐색 방법 중에서 가장 간단하고 직접적인 탐색 방법
- 정렬되지 않은 배열을 처음부터 마지막까지 하나씩 검사하는 방법
- 평균 비교 횟수
  - 탐색 성공: (n + 1)/2번 비교
  - 탐색 실패: n번 비교
- 시간 복잡도: O(n)

```
int seq_search(int key, int low, int high){
int i;
for(i=low; i<=high; i++)
   if(list[i]==key)
            return i; // 탐색 성공
return -1;                  // 탐색 실패
}
```

순차 탐색(sequential search)

• 8을 찾는 경우

(1) 9≠8이므로 탐색 계속

(2) 5≠8이므로 탐색 계속

(3) 8=8이므로 탐색 성공

• 2를 찾는 경우

(1) 9≠2이므로 탐색 계속

(2) 5≠2이므로 탐색 계속

(3) 8≠2이므로 탐색 계속

(4) 3≠2이므로 탐색 계속

(5) 7≠2이므로 탐색 계속

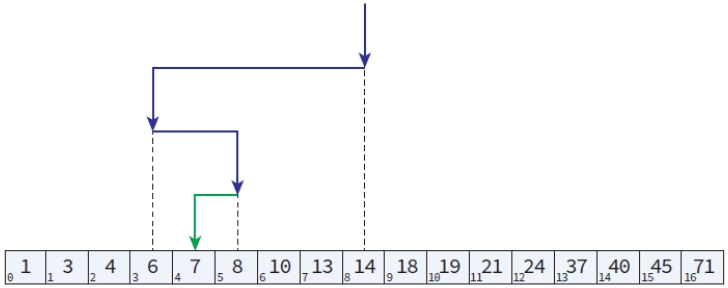
(6) 더 이상 항목이 없으므로 탐색 실패

순차 탐색(sequential search)

```
#include <stdio.h>
#define MAX_SIZE 1000
int list[MAX_SIZE] = { 3, 9,15, 22, 31, 55, 67, 88, 91 };
int n = 9;
int seq_search(int key, int low, int high){
  int i;
  for (i = low; i <= high; i++)
    if (list[i] == key)
      return i;
  return -1;
}
void main(){
  int i;
  i = seq_search(67, 0, n);
  if (i >= 0) {
    printf("탐색 성공 i=%d₩n", i);
  }
  else {
    printf("탐색 실패₩n");
  }
}
```

ex1.c

탐색



이진 탐색(binary search)

- 정렬된 배열의 중앙에 있는 값을 조사하여 찾고자 하는 항목이 왼쪽 또는 오른쪽 부분 배열에 있는지 알아내어 탐색 범위를 반으로 줄여가며 탐색 진행

• 5를 탐색하는 경우

7과 비교

1	3	5	6	7	9	11	20	30

5<7이므로 앞부분만을 다시 탐색

1	3	5	6

5를 3과 비교

1	3	5	6

5>3이므로 뒷부분만을 다시 탐색

5	6

5==5이므로 탐색 성공

5	6

• 2를 탐색하는 경우

7과 비교

1	3	5	6	7	9	11	20	30

2<7이므로 앞부분만을 다시 탐색

1	3	5	6

2를 3과 비교

1	3	5	6

2<3이므로 앞부분만을 다시 탐색

1

2>1이므로 뒷부분만을 다시 검색

1

더 이상 남은 항목이 없으므로 탐색 실패

이진 탐색(binary search)

```
search_binary(list, low, high)
        middle ← low에서 high사이의 중간 위치
        if( 탐색값 = list[middle] ) return TRUE;
        else if (탐색값 < list[middle] )
                return list[0]부터 list[middle-1]에서의 탐색;
        else if (탐색값 > list[middle] )
                return list[middle+1]부터 list[high]에서의 탐색;
```

```
int binsearch(int list[], int n, int searchnum){
  int left = 0;   int right = n - 1;   int middle;
  count = 0;
  while (left <= right) {
    count++;
    middle = (left + right) / 2;
    if (searchnum == list[middle])
      return middle;
    else if (searchnum < list[middle])
      right = middle - 1;
    else
      left = middle + 1;
  }
  return -1;
}
```

ex2.c

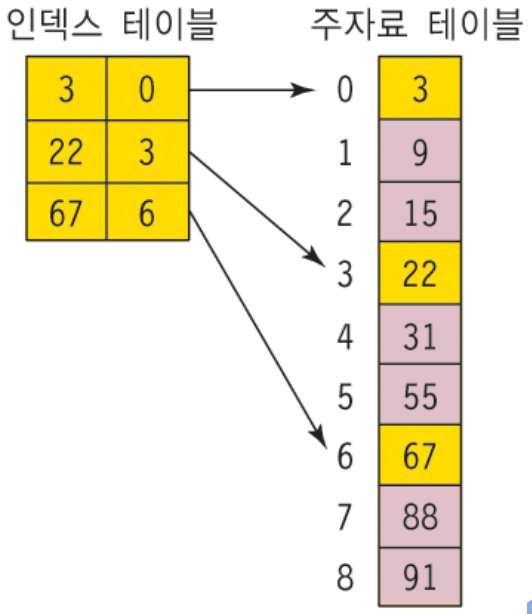
탐색

색인 순차 탐색(indexed sequential search)

- 인덱스(index) 테이블을 사용하여 탐색 효율 증대
  - 주 자료 리스트에서 일정 간격으로 발췌한 자료 저장

- 주 자료와 인덱스 테이블 모두 정리되어 있어야 함

- 복잡도: O(m+n/m)
  - 인덱스 테이블 크기=m
  - 주 자료 리스트 크기=n



탐색

색인 순차 탐색(indexed sequential search)

```c
#include <stdio.h>
#define MAX_SIZE 1000
#define INDEX_SIZE 10

int list[MAX_SIZE] = { 3, 9,15, 22, 31, 55, 67, 88, 91 };
int n = 9;
typedef struct {
  int key;
  int index;
} itable;
itable index_list[INDEX_SIZE] = { {3,0}, {15,3}, {67,6} };

int seq_search(int key, int low, int high){
  int i;
  for (i = low; i <= high; i++)
    if (list[i] == key)
      return i;  /* 탐색에 성공하면 키 값의 인덱스 반환 */
  return -1;    /* 탐색에 실패하면 -1 반환 */
}
```

ex3.c

탐색

색인 순차 탐색(indexed sequential search)

```
int index_search(int key){
  int i, low, high;
  if (key<list[0] || key>list[n - 1])
    return -1;
  for (i = 0; i < INDEX_SIZE; i++)
    if (index_list[i].key <= key && index_list[i + 1].key > key)
      break;
  if (i == INDEX_SIZE) {
    low = index_list[i - 1].index;
    high = n;
  }
  else {
    low = index_list[i].index;
    high = index_list[i + 1].index;
  }
  return seq_search(key, low, high);
}
```

```
void main(){
  int i;
  i = index_search(67);
  if (i >= 0) {
    printf("탐색 성공 i=%d₩n", i);
  }
  else {
    printf("탐색 실패₩n");
  }
}
```

ex3.c

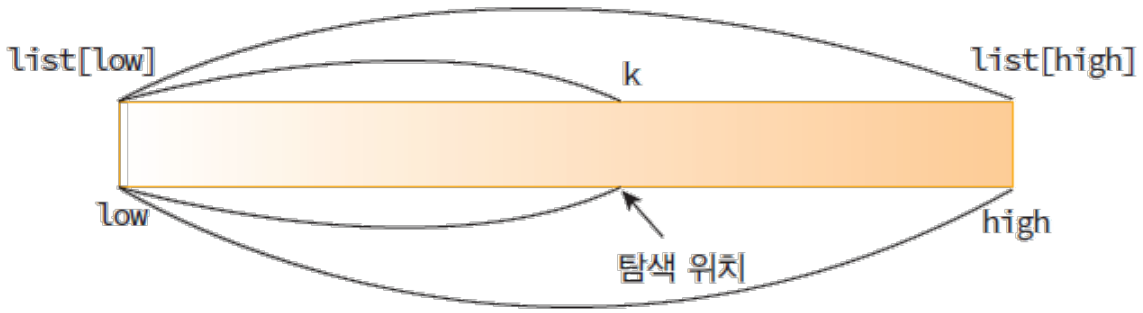
탐색

보간 탐색(interpolation search)

- 사전이나 전화번호부를 탐색하는 방법
- 탐색키가 존재할 위치를 예측하여 탐색
  - 복잡도: O(logn)
- 보간 탐색은 이진 탐색과 유사하나 리스트를 불균등 분할하여 탐색

$$\text{탐색 위치} = \frac{(k - list[low])}{list[high] - list[low]} * (high - low) + low$$

$$(list[high] - list[low]) : (k - list[low]) = (high - low) : (\text{탐색 위치} - low)$$



보간 탐색(indexed sequential search)

```
...
int search_interpolation(int key, int n){
  int low, high, j;
  low = 0;
  high = n - 1;
  while ((list[high] >= key) && (key > list[low])) {
    j = ((float)(key - list[low]) / (list[high] - list[low]) * (high - low))
+ low;
    if (key > list[j]) low = j + 1;
    else if (key < list[j]) high = j - 1;
    else low = j;
  }
  if (list[low] == key)
    return(low);
  else
    return -1;
}
...
```

ex4.c

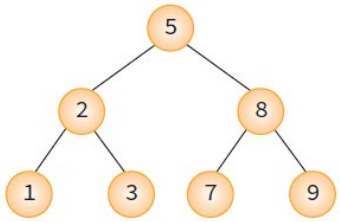
탐색

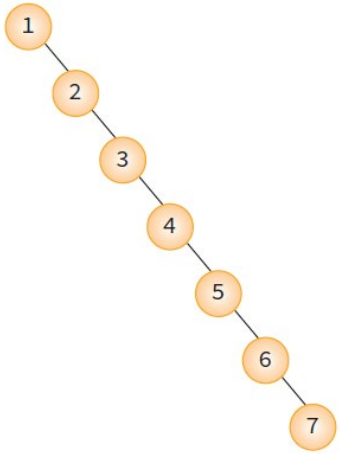
균형 이진 탐색 트리

- 이진 탐색(binary search)과 이진 탐색 트리(binary search tree)은 근본적으로 같은 원리에 의한 탐색 구조

- 이진 탐색은 자료들이 배열에 저장되어 있으므로 삽입/삭제가 매우 비효율

- 이진 탐색 트리는 매우 빠르게 삽입/삭제 수행





탐색

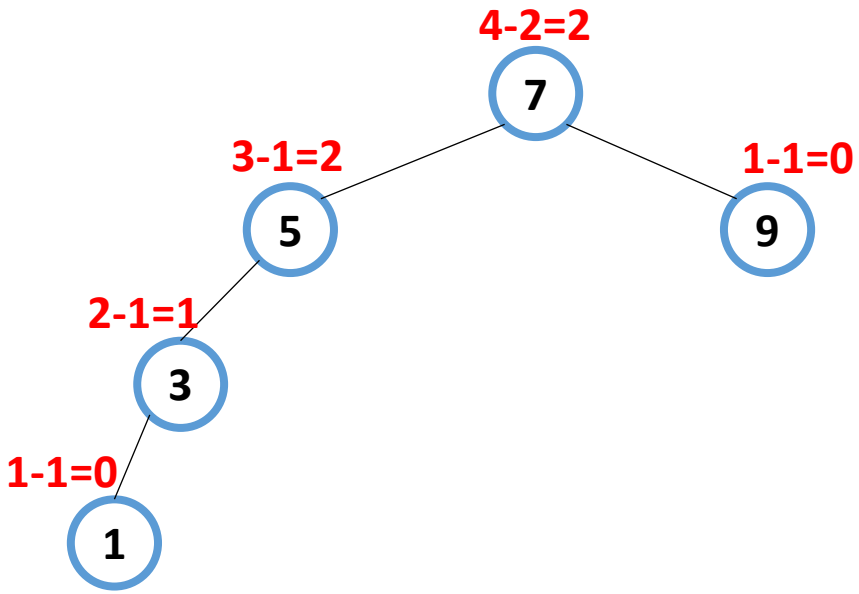
균형 이진 탐색 트리 (AVL 트리)

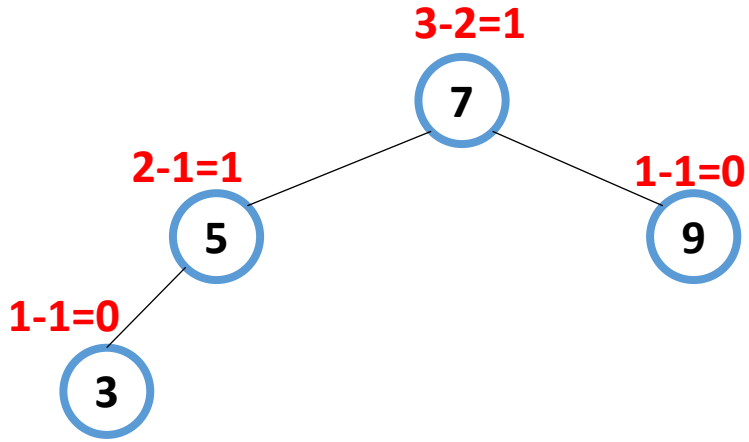
- Adelson-Velskii와 Landis에 의해 1962년 제안된 트리
- 모든 노드의 왼쪽과 오른쪽 서브 트리 높이 차가 1이하인 이진 탐색 트리
- 트리가 비균형 상태로 되면 스스로 노드들을 재배치하여 균형 상태로 유지

탐색

균형 이진 탐색 트리 (AVL 트리)

- 평균, 최선, 최악 시간 복잡도 = O(log(n))
- 균형 인수(balance factor) 필요
  - 왼쪽 서브 트리 높이 - 오른쪽 서브 트리 높이 값
- 모든 노드의 균형 인수가 ±1 이하면 AVL 트리





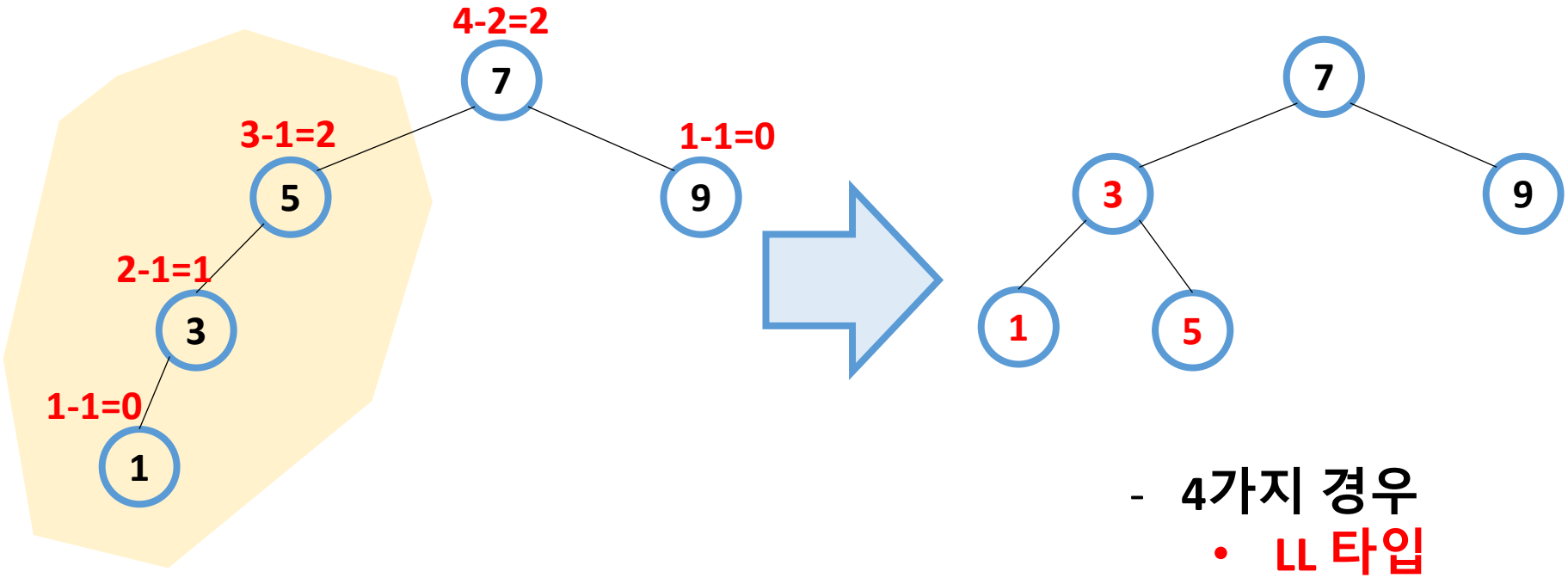
탐색

균형 이진 탐색 트리 (AVL 트리)

- 탐색 연산: 이진 탐색 트리와 동일
- 삽입 연산과 삭제 연산 시 균형 상태가 깨질 수 있음
- 삽입 연산
  - 삽입 위치에서 루트까지 경로에 있는 조상 노드들의 균형 인수 영향
  - 삽입 후 불균형 상태로 변한 가장 가까운 조상 노드(균형 인수가 ±2가 된 가장 가까운 조상 노드)의 서브 트리들에 대하여 다시 재 균형
  - 삽입 노드부터 균형 인수가 ±2가 된 가장 가까운 조상 노드까지 회전

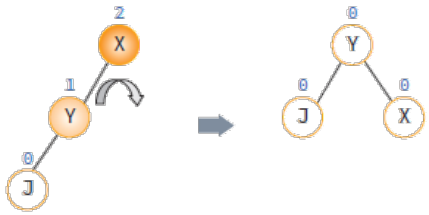
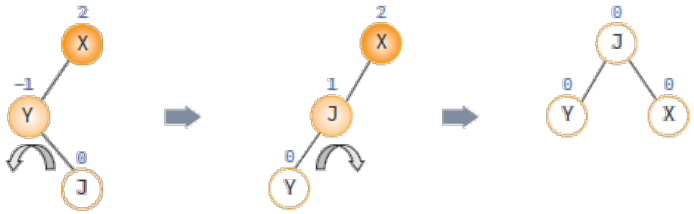
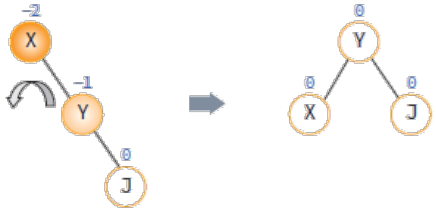
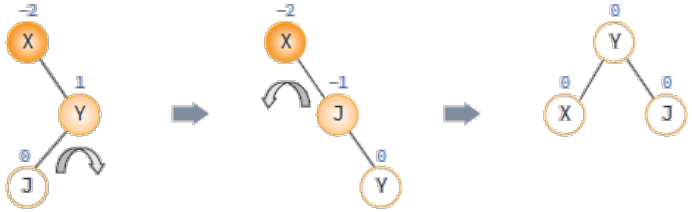
탐색

균형 이진 탐색 트리 (AVL 트리)



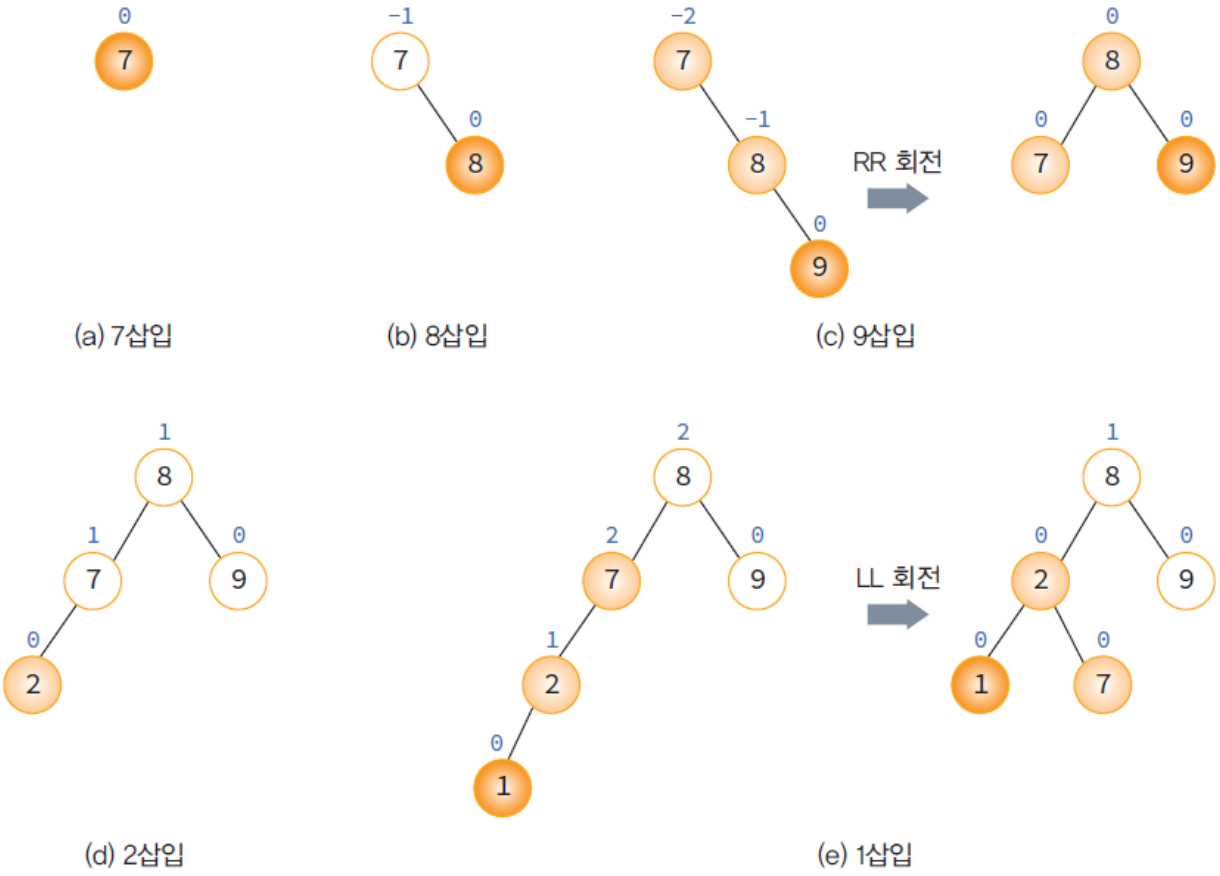
- 4가지 경우
  - LL 타입
  - LR 타입
  - RR 타입
  - RL 타입

균형 이진 탐색 트리 (AVL 트리)

4가지의 경우	해결방법	설명
LL 타입		LL 회전: 오른쪽 회전
LR 타입		LR 회전: 왼쪽 회전 → 오른쪽 회전
RR 타입		RR 회전: 왼쪽 회전
RL 타입		RL 회전: 왼쪽 회전 → 오른쪽 회전

균형 이진 탐색 트리 (AVL 트리)

(7, 8, 9, 2, 1)



균형 이진 탐색 트리 (AVL 트리)

```
typedef struct AVLNode{
  int key;
  struct AVLNode* left;
  struct AVLNode* right;
} AVLNode;
```

```
AVLNode* insert(AVLNode* node, int key){
  if (node == NULL)
    return(create_node(key));
  if (key < node->key)
    node->left = insert(node->left, key);
  else if (key > node->key)
    node->right = insert(node->right, key);
  else
    return node;
  int balance = get_balance(node);
  if (balance > 1 && key < node->left->key)
    return rotate_right(node);
  if (balance < -1 && key > node->right->key)
    return rotate_left(node);
  if (balance > 1 && key > node->left->key){
    node->left = rotate_right(node->left);
    return rotate_right(node);
  }
  if (balance < -1 && key < node->right->key){
    node->right = rotate_right(node->right);
    return rotate_left(node);
  }
  return node;
}
```

```
AVLNode* rotate_left(AVLNode* parent){
  AVLNode* child = parent->right;
  parent->right = child->left;
  child->left = parent;
  return child;
}
```

```
AVLNode* rotate_right(AVLNode* parent){
  AVLNode* child = parent->left;
  parent->left = child->right;
  child->right = parent;
  return child;
}
```

ex5.c