CHAP 13

탐색(searching, retrieval)

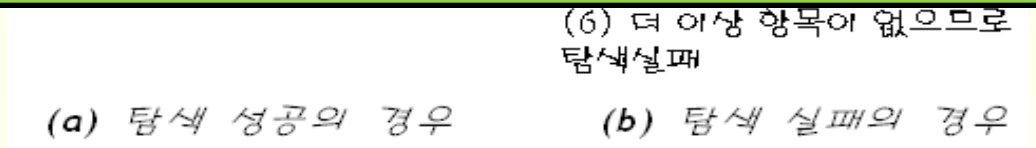
탐색이란?

- **탐색**(search) : 기본적으로 여러 개의 자료 중에서 원하는 자료를 찾는 작업
  - 컴퓨터가 가장 많이 하는 작업 중의 하나
  - 탐색을 효율적으로 수행하는 것은 매우 중요
- **탐색키**(search key) : 각 항목을 구별해주는 키(key)
  - 예 : 주민등록번호, 학번, 이름+전화번호
- **탐색의 전제 조건** : 없음 vs. 정렬
  - 없음 : 순차검색, 해싱(특수한 검색방법)
  - 정렬 : 이진검색, 색인순차검색, 보간검색 등
- **탐색에 사용되는 자료 구조**
  - 배열, 연결 리스트, 트리, 그래프 등

1. 순차 탐색

- 탐색 방법 중에서 가장 간단하고, 직접 비교에 의한 탐색 방법
- 전제 조건 : 없음(정렬 또는 비정렬 상태에 무관)
- 방법 : 항목들을 처음부터 찾을 때(또는 마지막)까지 하나씩 비교검사

```
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
}
```



- 성능 : n개의 데이터에서 키 검색을 위한 비교문(if문) 횟수
  - 최선 경우 : 1번(맨 앞에서)
  - 최악 경우 : n번(맨 끝에서)
  - 평균 경우 : 1/n * (1+2+ … +n) = 1/n * n(n+1)/2 = (1+n)/2

- 실제 상황에서의 평균 성능은 위와 다르다?

  키값이 데이터 내에 있을 확률 50% = ½, 없을 확률 50% = ½

  ➔ 0.5*(1+n)/2 + 0.5*n = (1+n)/4 + 2n/4 = (1+3n)/4

  - 결론 : 실제의 평균 검색시간은 데이터의 약 75%는 비교해야 함

2. 이진탐색

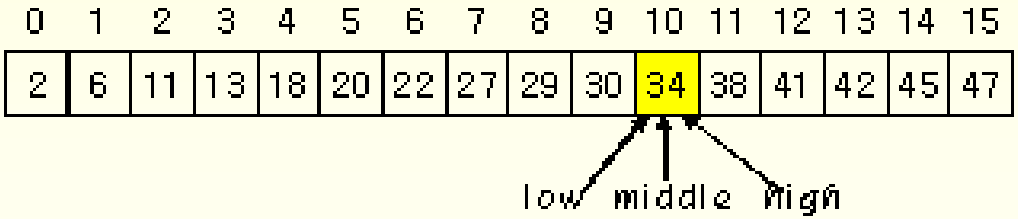
- 정렬된 데이터에 대한 탐색은 **성능 면**에서 이진 탐색(binary search)이 적합
- 조건 : **정렬**
- 방법 : 중앙 값을 비교 ➔ 맞으면 검색 종료, 아니면 왼쪽 또는 오른쪽 부분을 선택 ➔ 탐색 범위를 반으로 줄여감

(예) 10억(약 2^{30}) 명중에서 이진탐색을 이용하여 특정인을 탐색한다면?

- 이진탐색 사용 : 최대 31번, 평균 30번 비교
- 순차탐색 사용 : 평균 5억번 비교

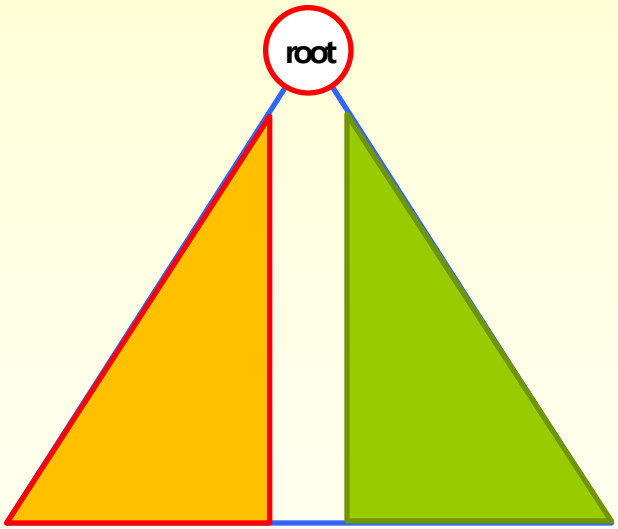
이진탐색 알고리즘

```
int  search_binary(key, low, high)
{
    int  result;
    mid ← low에서 high사이의 중간 위치 첨자;
    if (key == list[mid]) return  mid;
    else  if (key < list[mid]) result = search_binary(key, low, mid-1);
          else                 result = search_binary(key, mid+1, high);
    return  result; //위치 반환
}
```



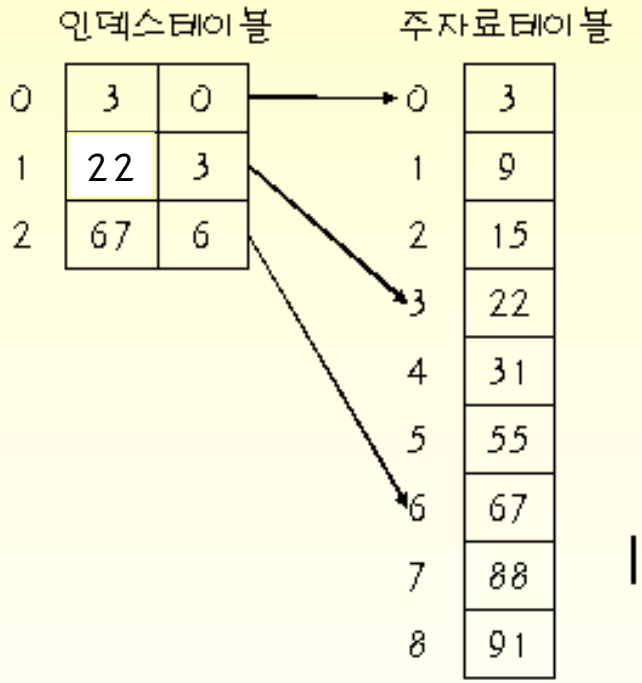
- 성능 : n개의 데이터에서 키 검색을 위한 비교문(if문) 횟수
  - 최선 경우 : 1번(맨 앞에서)
  - 최악 경우 : $\log_2 n + 1$ 번(맨 끝에서)
  - 평균 경우 : $\log_2 n$ 번

- 최악, 평균 경우의 성능이 위와 같이 계산되는 이유?



3. 색인 순차탐색

- 인덱스(index)를 사용하여 구성된 테이블에서 검색 ➔ 탐색효율 향상
- 조건 : 인덱스 테이블, 주 자료 테이블 모두 정렬
  (인덱스 테이블은 주 자료 테이블에서 일정간격으로 추출한 자료 저장)
- 방법 : 인덱스 테이블에서 크기 비교 ➔ 주 자료 테이블에서는 순차검색



4. 보간 탐색

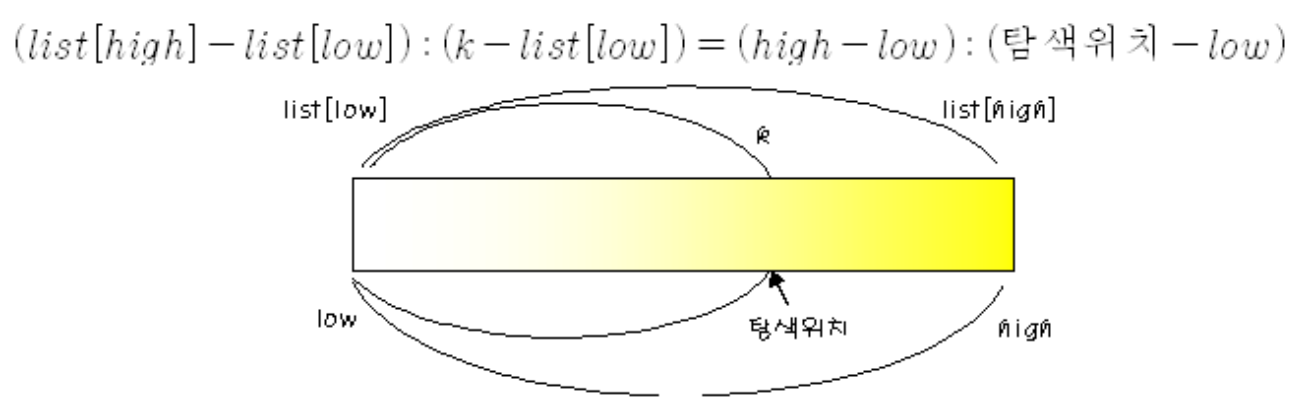
- 사전, 전화번호부 탐색방법과 같이, 키가 존재할 위치를 예측하여 탐색
- 조건 : 정렬
- 방법 : 이진 탐색과 유사하나, 리스트를 반으로 분할하지 않고 불균등하게 분할하여 키가 존재할 구간을 탐색

- 탐색 위치 = ((key – list[low]) / (list[high] - list[low])) * (high-low) + low

(예) key = 100, low = 1, high = 50, list[low] = 15, list[high] = 215 일때,

탐색위치 = ((100 – 15)/(215 – 15)) * (50-1) + 1 = 85/200*49 + 1

= 21.815 ➔ 21

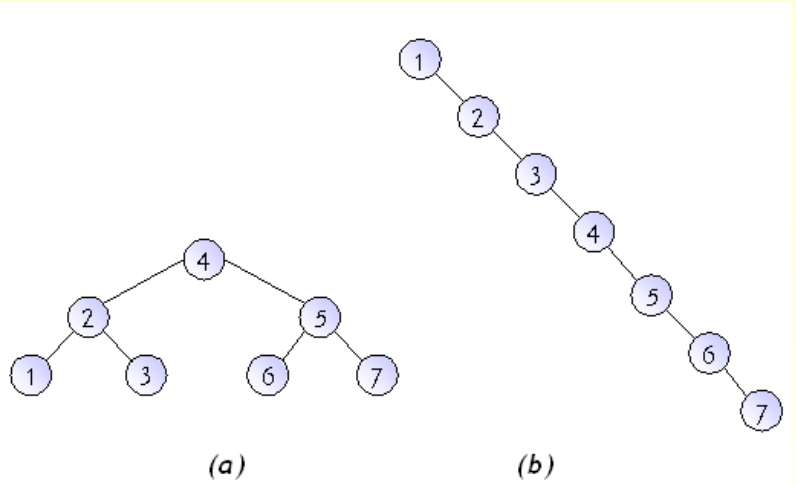


5. 균형잡힌 이진탐색 트리

- 이진 검색(binary search)과 이진탐색 트리(binary search tree) 검색
  - 근본적으로 같은 원리에 의한 탐색 방법(검색시간 : $O(\log_2 n)$)
  - 이진 검색 : 자료들이 정렬된 상태로 저장 ➔ 삽입, 삭제 시간이 O(n)
  - 이진탐색트리 검색 : 트리 구성해서 검색 ➔ 삽입, 삭제 시간이 $O(\log_2 n)$
    ➔ 삽입, 삭제가 빈번한 데이터라면 **반드시 이진탐색트리**로 구성 후 사용
    but, 이진탐색트리 구축 복잡도, 메모리 사용이 많아짐

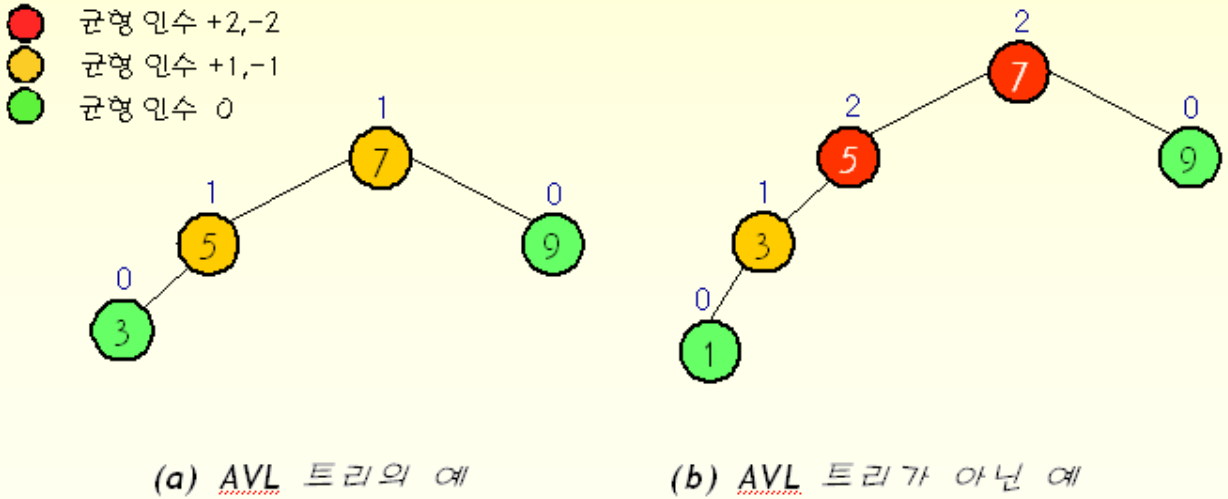
- 이진탐색트리의 구조별 시간복잡도
  (a) 균형 트리 : O(logn)
  (b) 불균형 트리 : O(n)

➔ **빠른 검색을 위해서**
이진검색보다는 이진탐색트리 검색,
일반적 이진탐색트리보다는
균형된 이진탐색트리 활용이 필요...



실시간 재구성 이진 검색트리 : AVL 트리

- 1962년 Adelson-Velskii, Landis에 의해 제안된 이진탐색 트리
- **구성 : 왼쪽 서브트리와 오른쪽 서브트리의 높이를 1이하로 유지하는 이진탐색 트리**
- 삽입/삭제로 인한 실시간 재구성
  - 삽입/삭제에 의해 균형 인수가 2이상인 서브트리가 발생 ➔ 스스로 노드 위치를 재배치하여 균형되게 재구성
  - 균형 트리가 항상 보장되기 때문에 탐색시간은 O(logn) 이내

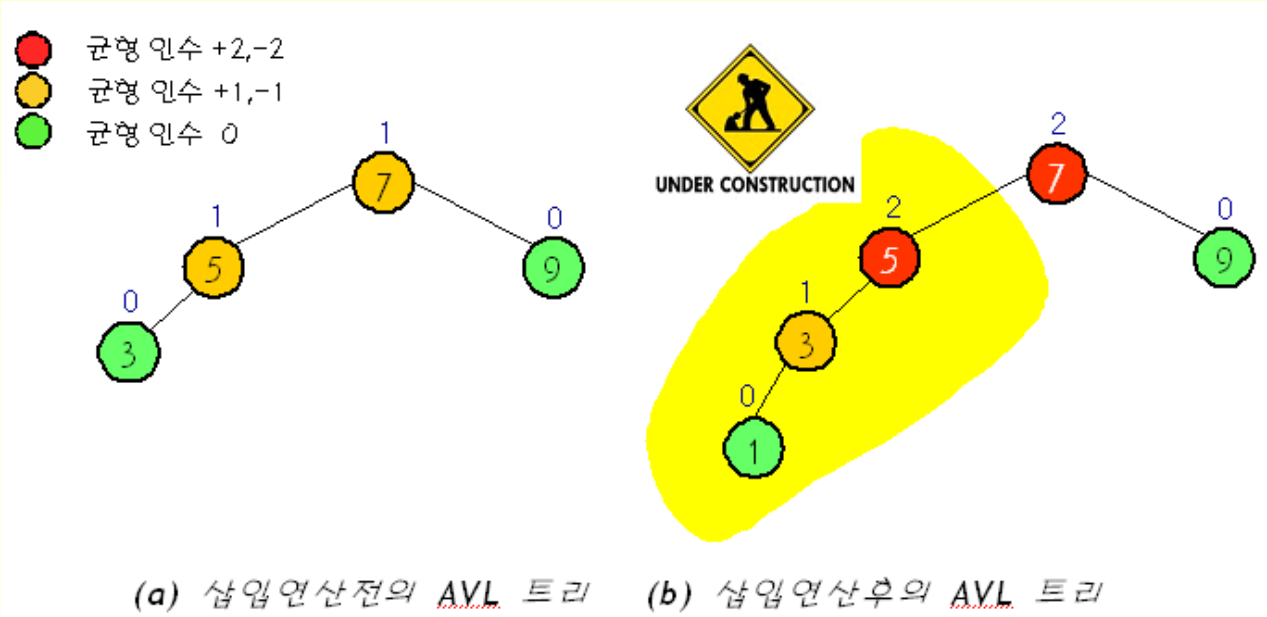


(a) AVL 트리의 예 (b) AVL 트리가 아닌 예

AVL 트리의 재구성 연산

- 불균형 상태 원인 : 삽입, 삭제 연산에 의한 좌우 서브트리의 높이 변동

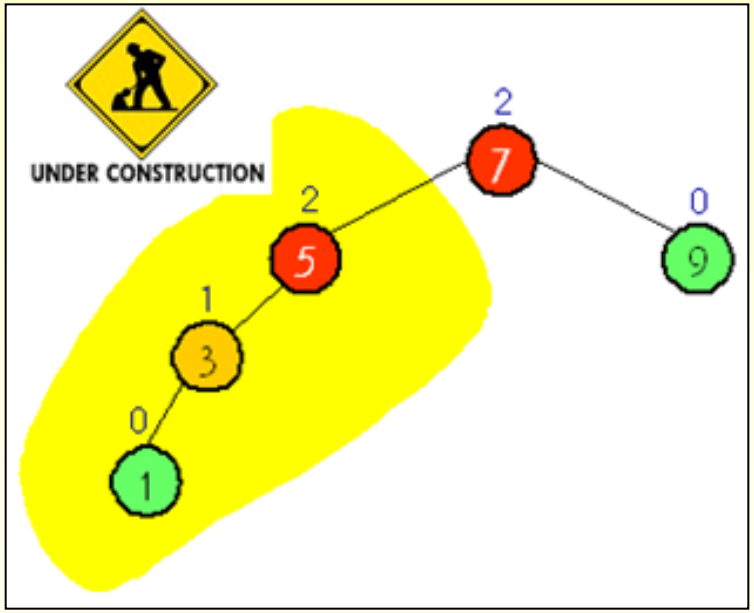
  예) 삽입연산 : 삽입된 위치에서 루트까지 경로 상에 있는 노드의 균형 인수는 변동
  ➔ 균형 인수가 ±2 이상 변동된 노드에 대하여 재균형 작업을 시도

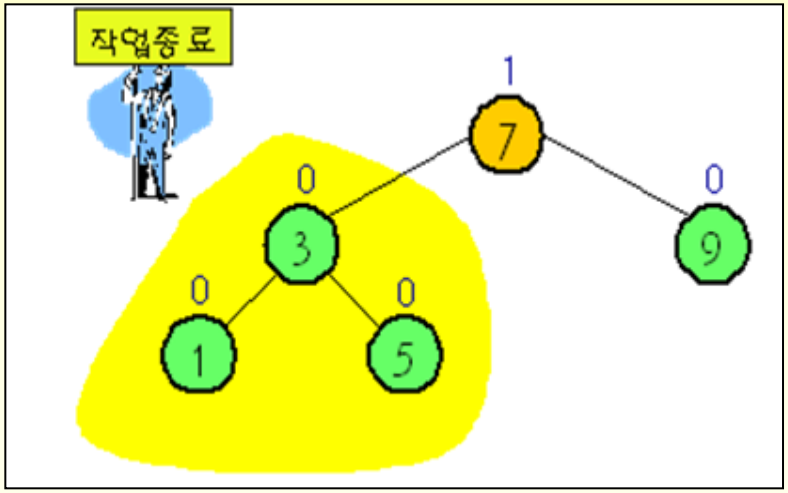


(a) 삽입연산전의 AVL 트리 (b) 삽입연산후의 AVL 트리

AVL트리의 삽입연산

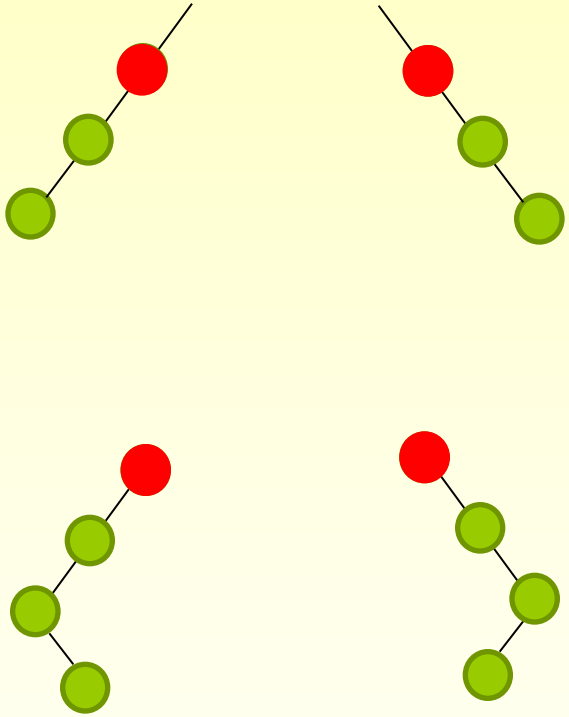
- 균형트리로 재구성하는 방법 : **회전(rotation)**
  - 불균형 인수를 가진 부모노드를 균형 인수를 가진 자식노드의 왼쪽 또는 오른쪽 자식 노드로 꺾어내리는(변형) 과정



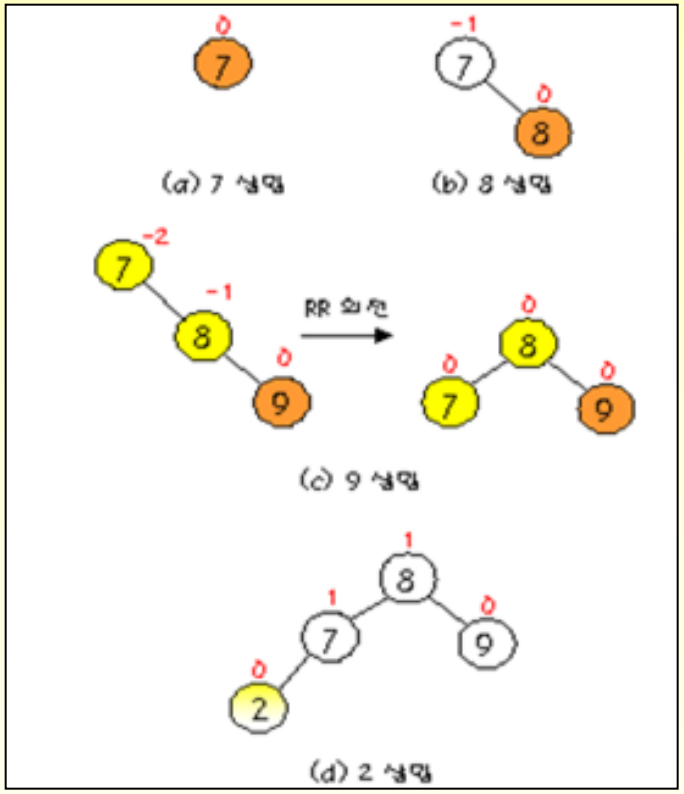


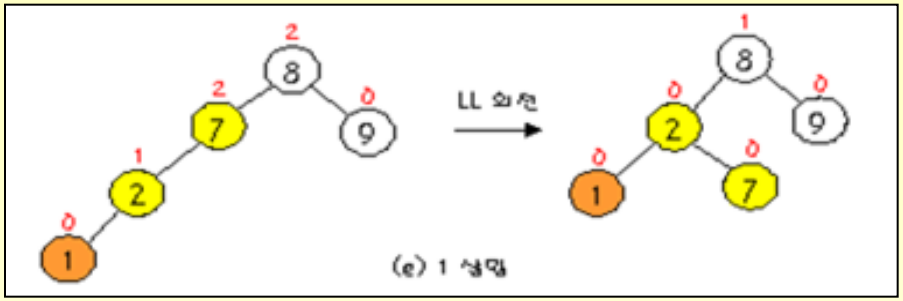
AVL트리의 삽입연산

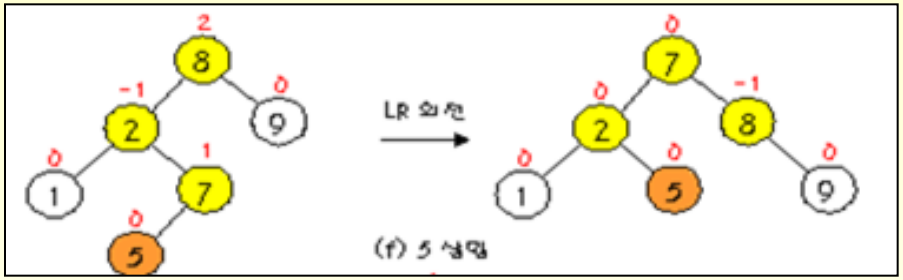
- AVL 트리의 불균형 상태 발생은 4가지 경우(LL, RR, LR, RL 유형)
  - LL 타입 : N이 A의 왼쪽 서브트리의 왼쪽 서브트리에 삽입되면
    ➔ **LL 회전 적용**
  - RR 타입 : N이 A의 오른쪽 서브트리의 오른쪽 서브트리에 삽입되면
    ➔ **RR 회전 적용**
  - LR 타입 : N이 A의 왼쪽 서브트리의 오른쪽 서브트리에 삽입되면
    ➔ **LR 회전 적용**
  - RL 타입 : N이 A의 오른쪽 서브트리의 왼쪽 서브트리에 삽입되면
    ➔ **RL 회전 적용**

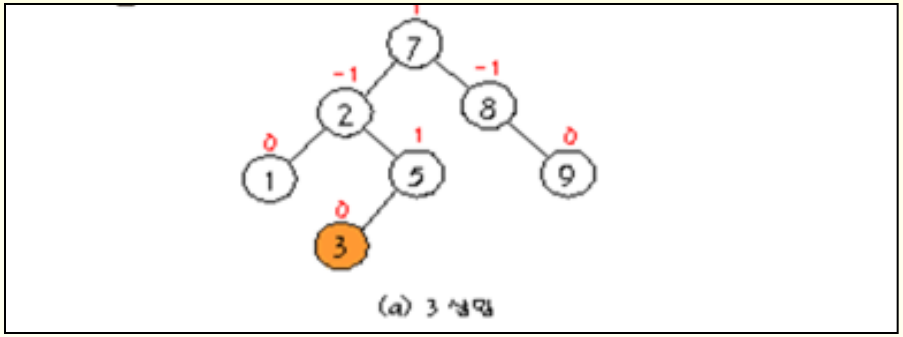


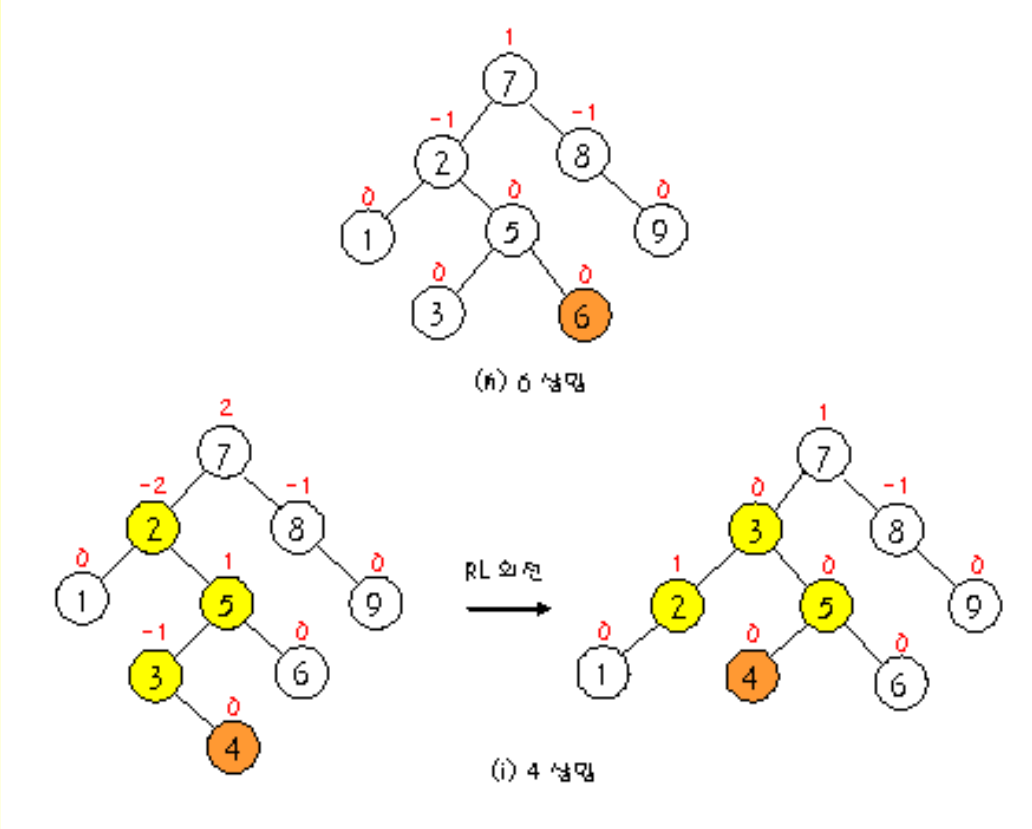
종합적인 예제







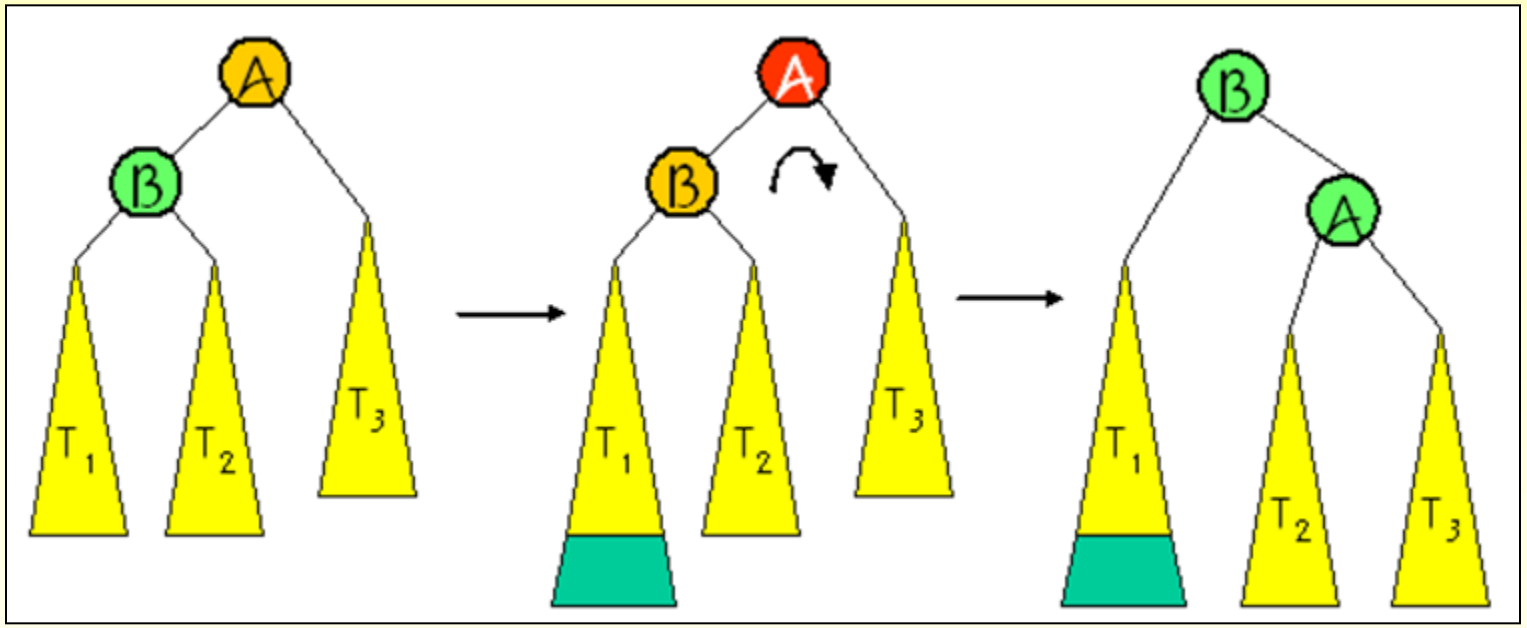




- LL 회전 : A부터 N까지의 경로상의 노드들을 오른쪽으로 회전
- LR 회전 : A부터 N까지의 경로상의 노드들을 왼쪽-오른쪽으로 회전
- RR 회전 : A부터 N까지의 경로상의 노드들을 왼쪽으로 회전
- RL 회전 : A부터 N까지의 경로상의 노드들을 오른쪽-왼쪽으로 회전

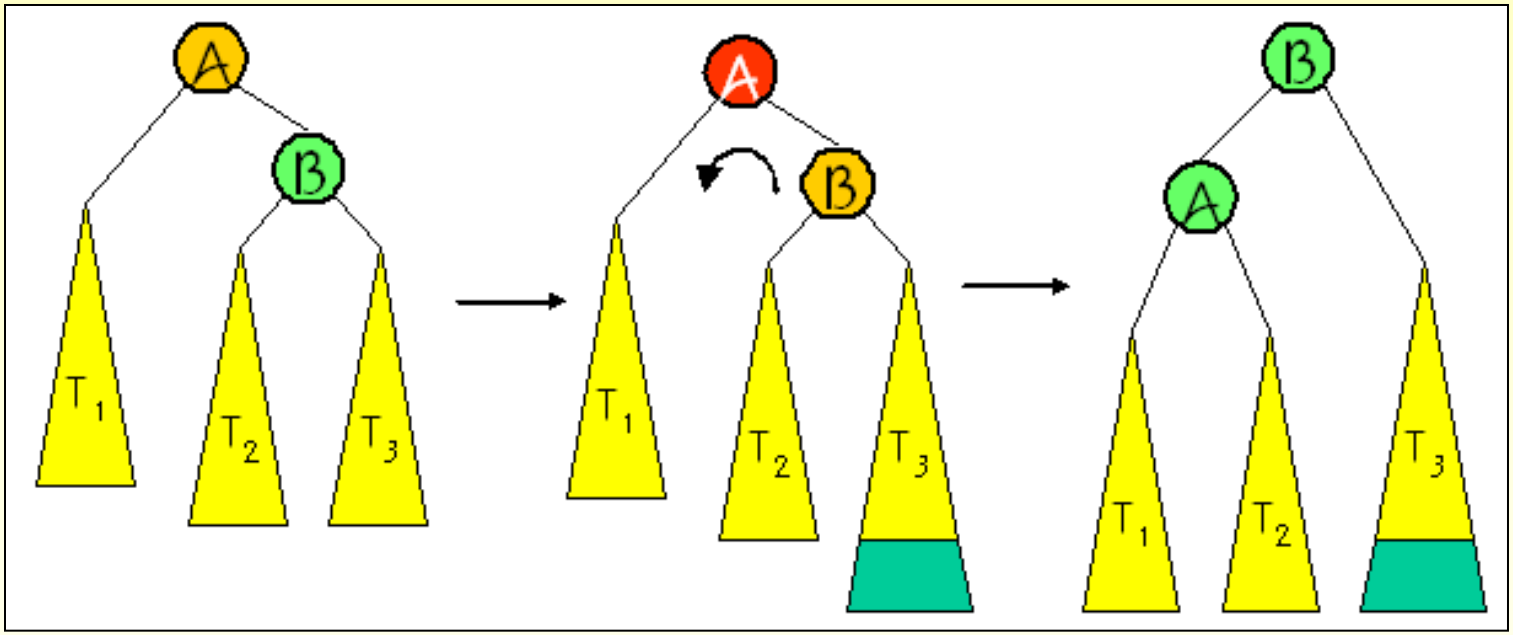
AVL 트리의 회전방법

LL 회전 RR 회전 RL 회전 LR 회전



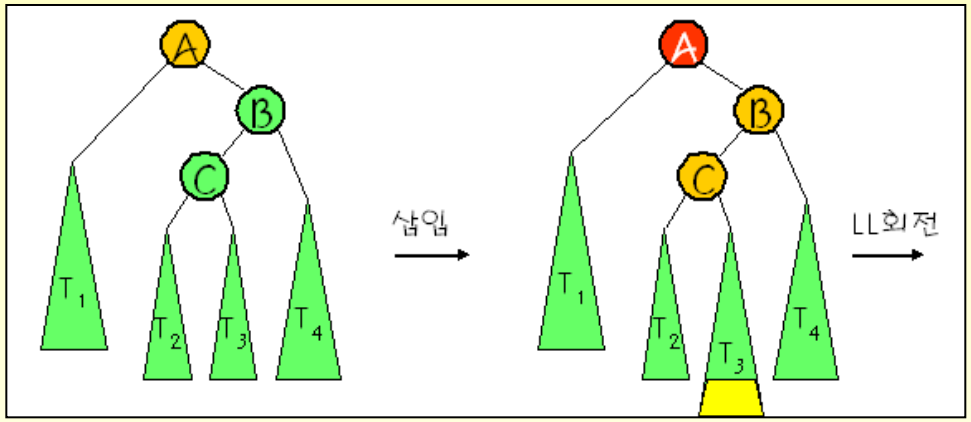
AVL 트리의 회전방법

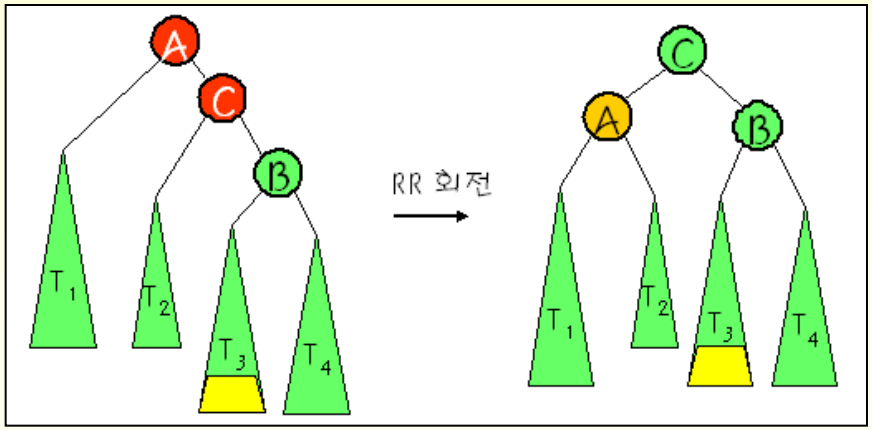
LL 회전 RR 회전 RL 회전 LR 회전



AVL 트리의 회전방법

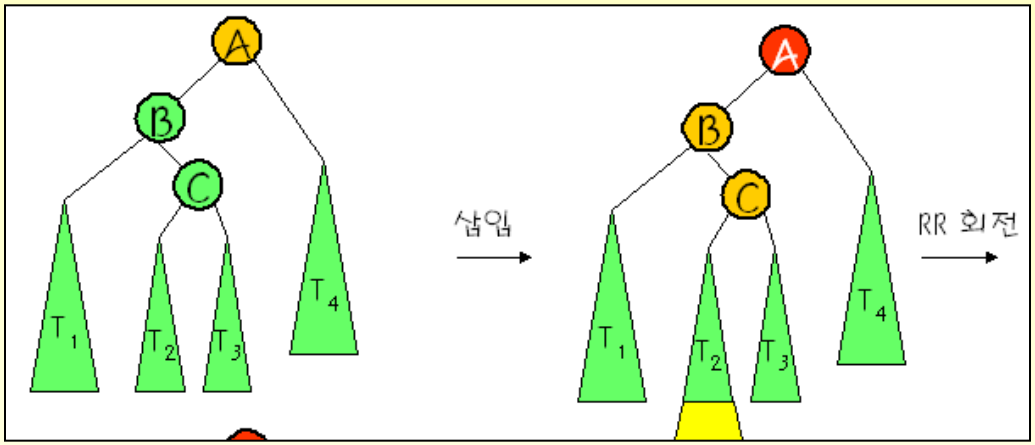
LL 회전 RR 회전 RL 회전 LR 회전

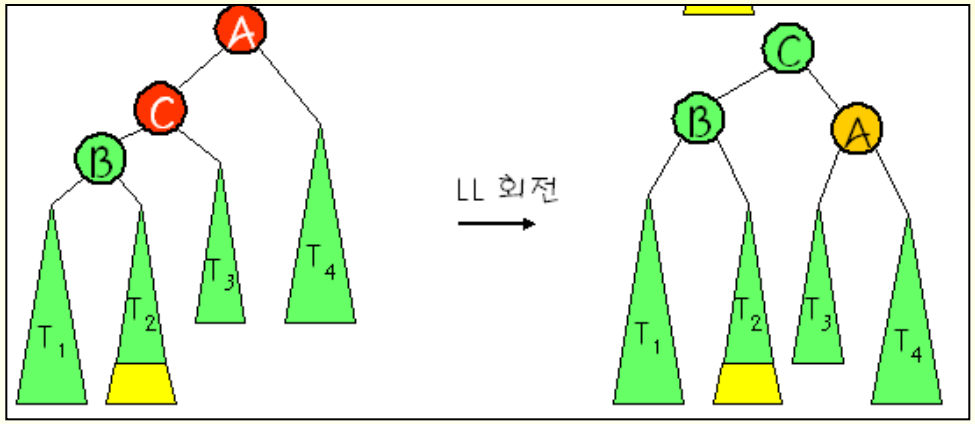




AVL 트리의 회전방법

LL 회전 RR 회전 RL 회전 LR 회전





AVL 트리 알고리즘 구성

```
struct  node  { int  data; struct  node *llink, *rlink; };
struct  node *root;
struct  node *rotate_right(struct  node *parent) //오른쪽 회전
{
        struct  node *child = parent->llink;
        parent->llink = child->rlink;
        child->rlink = parent;
        return child;
}
struct  node *rotate_left(struct  node *parent) //왼쪽 회전
{
        struct  node *child = parent->rlink;
        parent->rlink = child->llink;
        child->llink = parent;
        return child;
}
```

AVL 트리 알고리즘 구성

```
struct  node *rotate_right_left(struct  node *parent) //오른쪽-왼쪽
{
        struct  node *child = parent->rlink;
        parent->rlink = rotate_right(child);
        return rotate_left(parent);
}
struct  node *rotate_left_right(struct  node *parent) //왼쪽-오른쪽
{
        struct  node *child = parent->llink;
        parent->llink = rotate_left(child);
        return rotate_right(parent);
}
```

AVL 트리 알고리즘 구성

```
int  get_height(struct  node *node) // 트리의 높이를 반환
{
      int height=0;
      if( node != NULL )  height = 1 + max(get_height(node->llink),
                                           get_height(node->rlink));
      return height;
}

int  get_height_diff(struct node *node) // 노드의 균형인수를 반환
{
      if( node == NULL ) return 0;
      return  get_height(node->llink) - get_height(node->rlink);
}
```

AVL 트리 알고리즘 구성

```
struct  node *rebalance(struct  node **node)
{   //불균형 노드를 회전하여 균형된 이진트리로 재구성하는 문장 }

struct  node *avl_add(struct  node **root, int  new_key)
{   // AVL 트리에 새로운 노드를 삽입하는 문장                    }

struct avl_node *avl_search(struct avl_node *node, int key)
{   // AVL 트리의 탐색함수                                      }

void main()
{   //삽입 데이터 : 8,9,10,2,1,5,3,6,4,7,11,12
    avl_add(&root, 8);   avl_add(&root, 9);    avl_add(&root, 10);
    avl_add(&root, 2);   avl_add(&root, 1);    avl_add(&root, 5);
    avl_add(&root, 3);   avl_add(&root, 6);    avl_add(&root, 4);
    avl_add(&root, 7);   avl_add(&root, 11);   avl_add(&root, 12);
    avl_search(root, 12);
}
```