



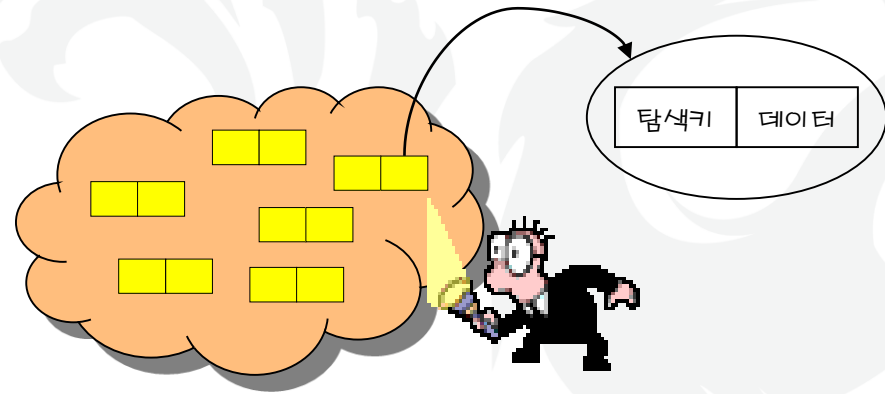
CSE2010 자료구조론

Week 15: Searching Revisited

ICT융합학부 한진영

탐색(Search)?

- 여러 개의 자료 중에서 원하는 자료를 찾는 작업
- 컴퓨터가 가장 많이 하는 작업 중의 하나
- 탐색을 효율적으로 수행하는 것은 매우 중요
- 탐색키(search key)
 - 항목과 항목을 구별해주는 키(key)
- 탐색을 위하여 사용되는 자료 구조
 - 배열, 연결 리스트, 트리, 그래프 등



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: $(n + 1)/2$ 번 비교
 - 탐색 실패: n 번 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
}
```

• 8을 찾는 경우

(1) $9 \neq 8$ 이므로 탐색 계속

9	5	8	3	7
---	---	---	---	---

(2) $5 \neq 8$ 이므로 탐색 계속

9	5	8	3	7
---	---	---	---	---

(3) $8 = 8$ 이므로 탐색 성공

9	5	8	3	7
---	---	---	---	---

(a) 탐색 성공의 경우

• 2를 찾는 경우

(1) $9 \neq 2$ 이므로 탐색 계속

9	5	8	3	7
---	---	---	---	---

(2) $5 \neq 2$ 이므로 탐색 계속

9	5	8	3	7
---	---	---	---	---

(3) $8 \neq 2$ 이므로 탐색 계속

9	5	8	3	7
---	---	---	---	---

(4) $3 \neq 2$ 이므로 탐색 계속

9	5	8	3	7
---	---	---	---	---

(5) $7 \neq 2$ 이므로 탐색 계속

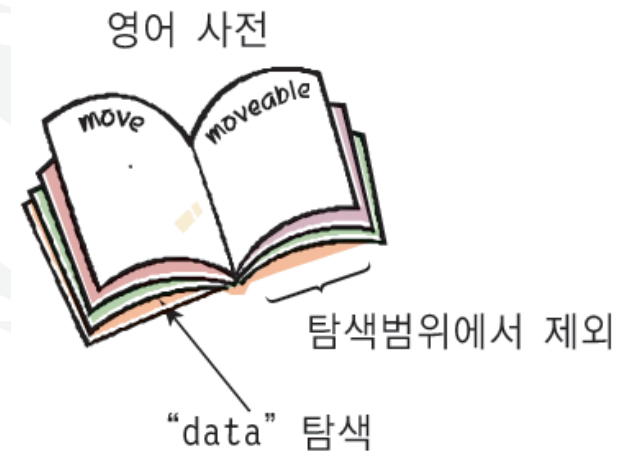
9	5	8	3	7
---	---	---	---	---

(6) 더 이상 항목이 없으므로
탐색 실패

(b) 탐색 실패의 경우

이진탐색(Binary Search)

- 정렬된 배열의 탐색에 적합
 - 배열의 중앙에 있는 값을 조사하여 찾고자 하는 항목이 왼쪽 또는 오른쪽 부분 배열에 있는지를 알아내어 탐색의 범위를 반으로 줄여가며 탐색 진행
- 예: 10억 명중에서 특정한 이름 탐색
 - 이진탐색 : 단지 30번의 비교 필요
 - 순차 탐색 : 평균 5억 번의 비교 필요



이진 탐색 동작과정

- 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
---	---

(a) 탐색이 성공하는 경우

- 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

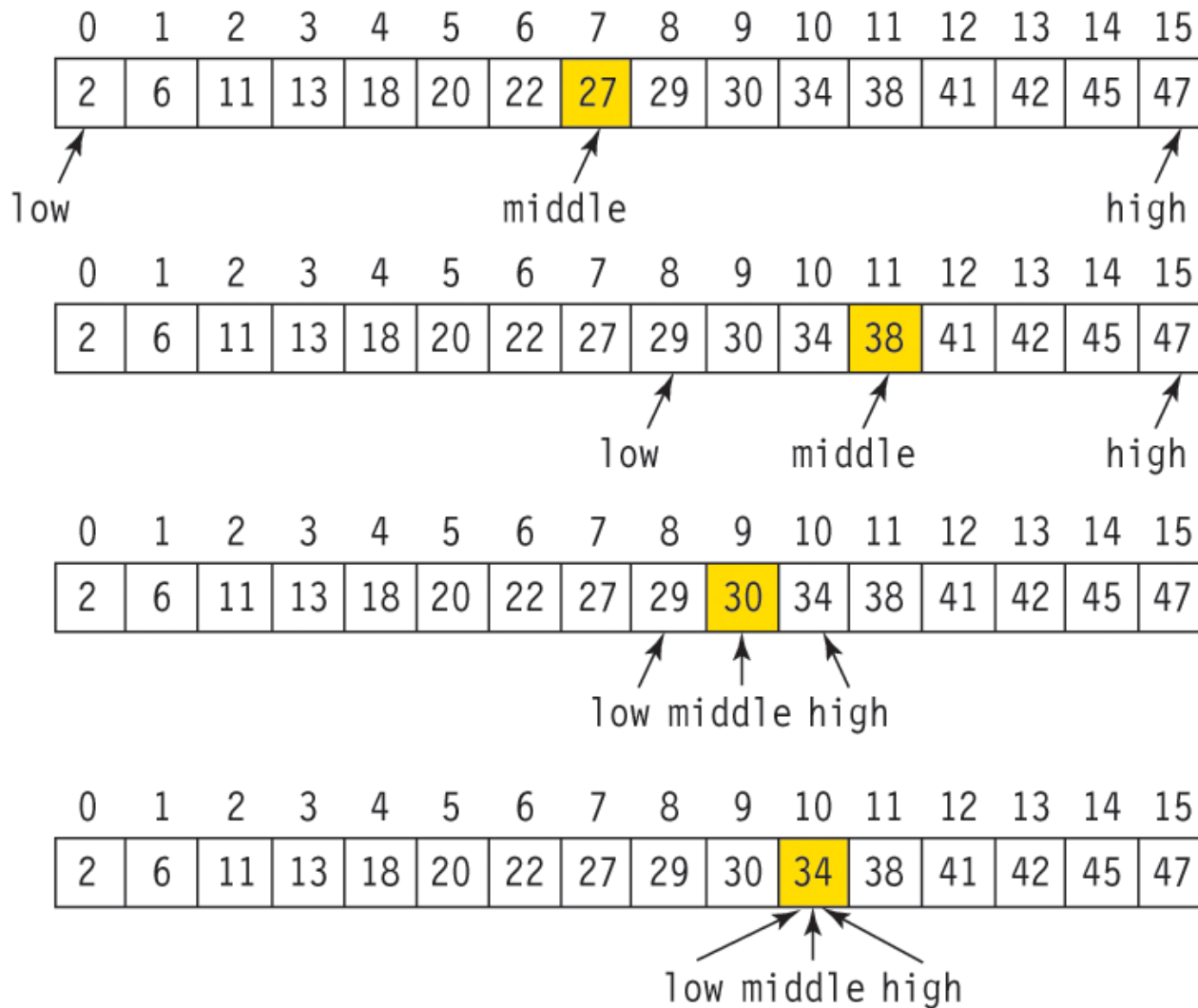
더 이상 남은 항목이 없으므로 탐색 실패

(b) 탐색이 실패하는 경우

이진탐색 알고리즘

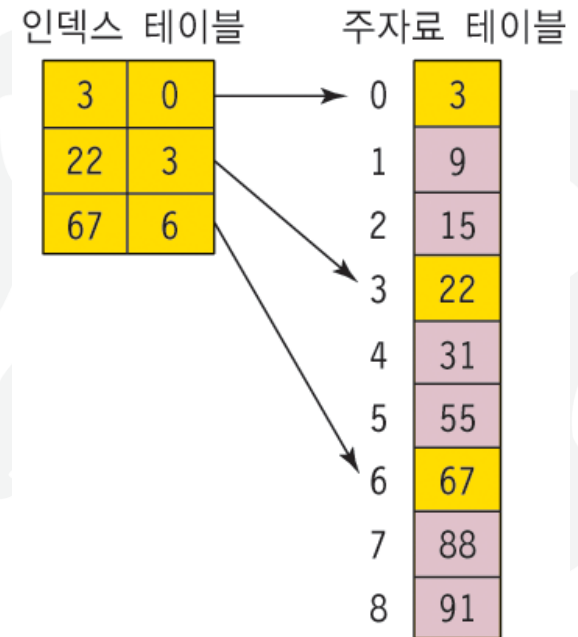
```
search_binary(list, low, high)
    middle ← low에서 high사이의 중간 위치
    if( 탐색값 ≠ list[middle] ) return TRUE;
    else if (탐색값 < list[middle] )
        return list[0]부터 list[middle-1]에서의 탐색;
    else if (탐색값 > list[middle] )
        return list[middle+1]부터 list[high]에서의 탐색;
```

이진탐색 예



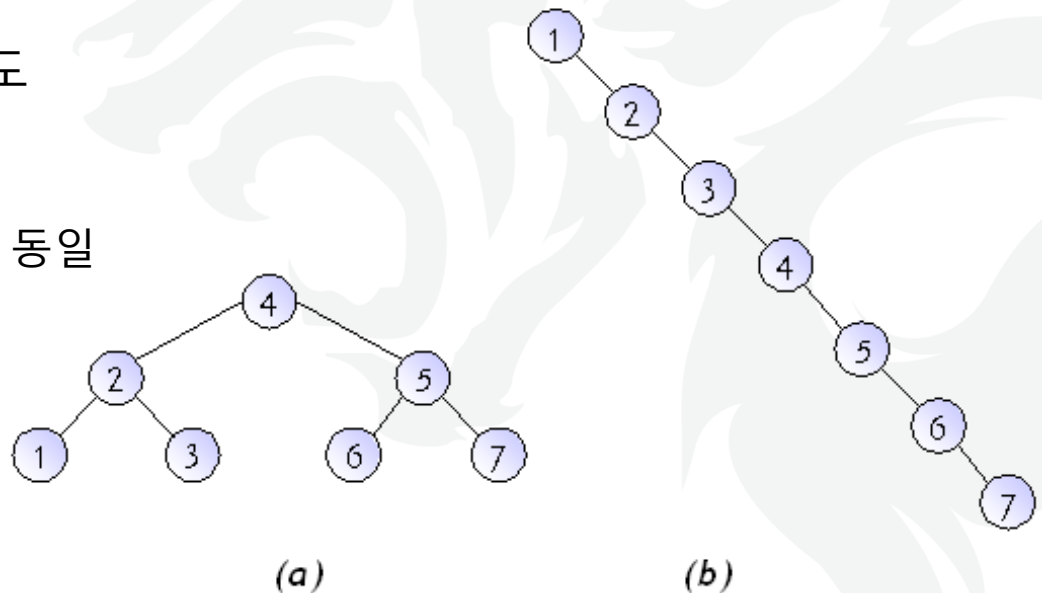
색인 순차탐색(Indexed Sequential Search)

- 인덱스(index) 테이블을 사용하여 탐색의 효율 증대
 - 주 자료 리스트에서 일정 간격으로 발췌한 자료 저장
- 주 자료 리스트와 인덱스 테이블은 모두 정렬되어 있어야 함
- 복잡도: $O(m+n/m)$
 - 인덱스 테이블의 크기= m , 주자료 리스트의 크기= n



균형 이진탐색트리

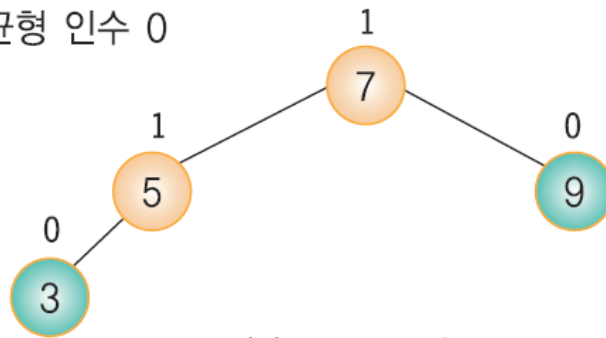
- 이진 탐색(binary search)과 이진 탐색 트리(binary search tree)의 차이점
 - 이진 탐색과 이진 탐색 트리는 근본적으로 같은 원리에 의한 탐색 구조
 - 이진 탐색은 자료들이 배열에 저장되어 있으므로 삽입/삭제가 매우 비효율
 - 자료의 삽입/삭제 시 원소들을 모두 이동시켜야 함
 - 이진 탐색 트리는 매우 빠르게 삽입/삭제 수행
 - 삽입, 삭제가 빈번히 이루어진다면 이진탐색트리가 유리함
- 이진탐색트리에서의 시간복잡도
 - 균형트리: $O(\log(n))$
 - 불균형트리: $O(n)$, 순차탐색과 동일



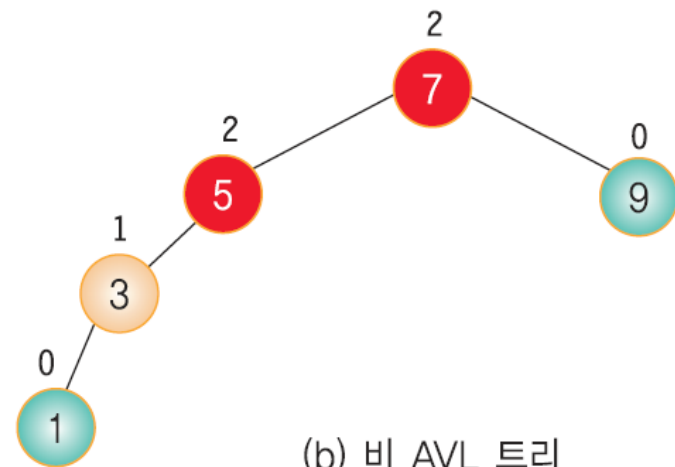
AVL 트리

- Adelson-Velskii와 Landis에 의해 1962년에 제안된 트리
- 모든 노드의 왼쪽과 오른쪽 서브트리의 높이 차이가 1이하인 이진 탐색트리
- 트리가 비 균형 상태로 되면 스스로 노드들을 재배치하여 균형 상태 유지
- 평균, 최선, 최악 시간복잡도: $O(\log(n))$
- 균형 인수(balance factor) = (왼쪽 서브 트리의 높이 - 오른쪽 서브 트리의 높이)
- 모든 노드의 균형 인수가 ± 1 이하이면 AVL 트리

- 균형 인수 +2, -2
- 균형 인수 +1, -1
- 균형 인수 0



(a) AVL 트리



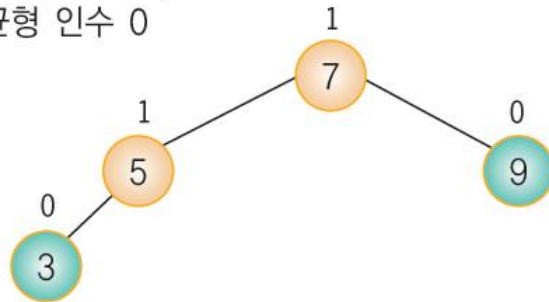
(b) 비 AVL 트리

AVL 트리의 연산

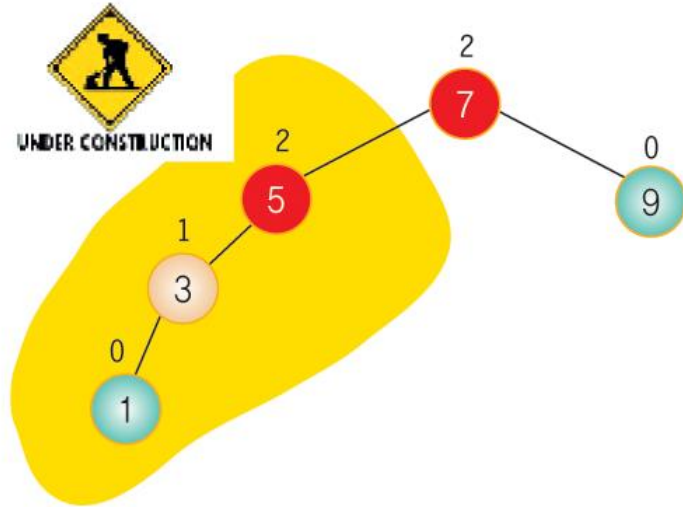
- 탐색연산: 이진탐색트리와 동일
- 삽입 연산과 삭제 연산 시 균형 상태가 깨질 수 있음
- 삽입 연산
 - 삽입 위치에서 루트까지의 경로에 있는 조상 노드들의 균형 인수 영향
 - 삽입 후에 불균형 상태로 변한 가장 가까운 조상 노드(균형 인수가 ± 2 가 된 가장 가까운 조상 노드)의 서브 트리들에 대하여 다시 재균형
 - 삽입 노드부터 균형 인수가 ± 2 가 된 가장 가까운 조상 노드까지 회전

AVL 트리의 삽입 연산(1)

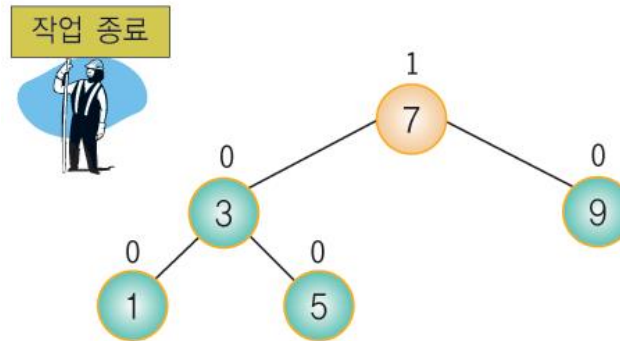
- 균형 인수 +2, -2
- 균형 인수 +1, -1
- 균형 인수 0



(a) 삽입 연산 전의 AVL 트리



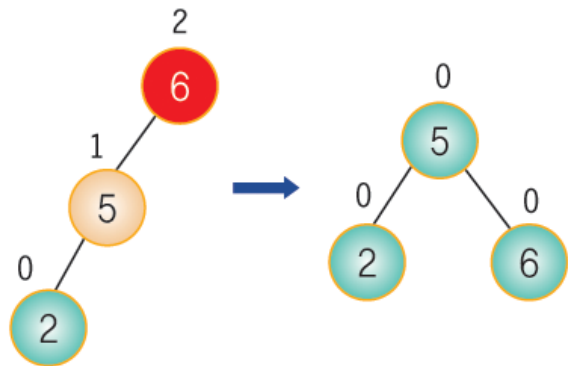
(b) 삽입 연산 후의 AVL 트리



AVL 트리의 삽입 연산(2)

- AVL 트리의 균형이 깨지는 4가지 경우(삽입된 노드 N으로부터 가장 가까우면서 균형 인수가 ± 2 가 된 조상 노드가 A라면)
 - LL 타입: N이 A의 왼쪽 서브 트리의 왼쪽 서브 트리에 삽입
 - LR 타입: N이 A의 왼쪽 서브 트리의 오른쪽 서브 트리에 삽입
 - RR 타입: N이 A의 오른쪽 서브 트리의 오른쪽 서브 트리에 삽입
 - RL 타입: N이 A의 오른쪽 서브 트리의 왼쪽 서브 트리에 삽입
- 각 타입별 재균형 방법
 - LL 회전: A부터 N까지의 경로상 노드의 오른쪽 회전
 - LR 회전: A부터 N까지의 경로상 노드의 왼쪽-오른쪽 회전
 - RR 회전: A부터 N까지의 경로상 노드의 왼쪽 회전
 - RL 회전: A부터 N까지의 경로상 노드의 오른쪽-왼쪽 회전

LL 회전방법

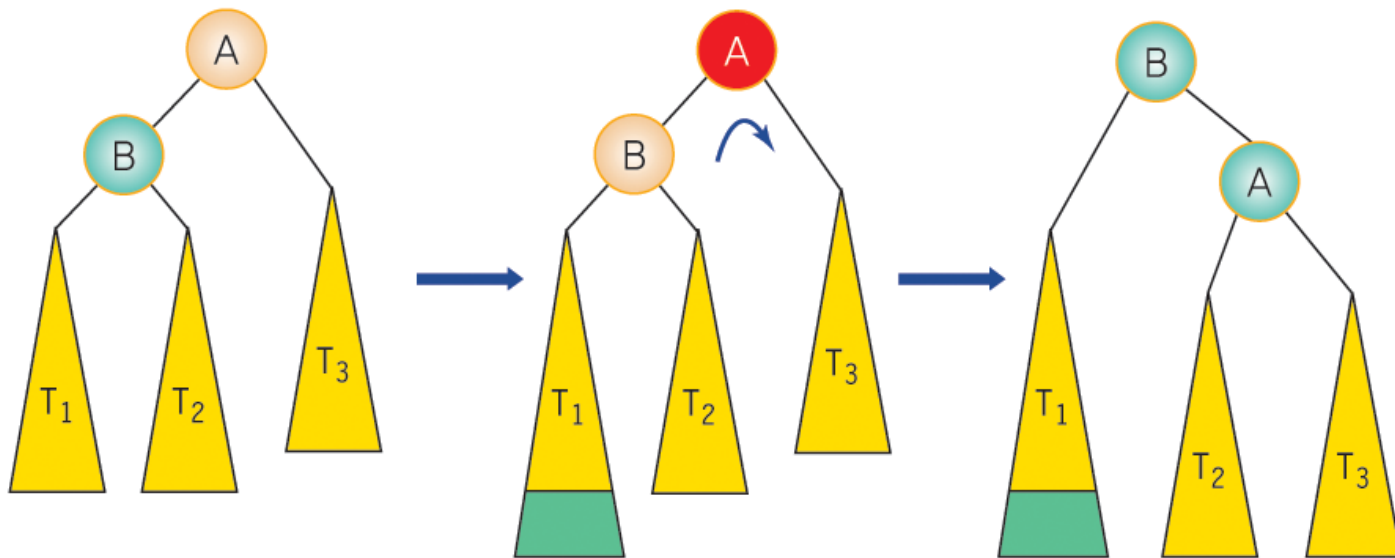


(a) LL 타입

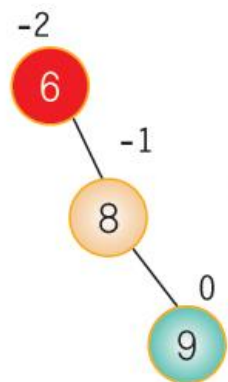
(b) LL 회전 결과

rotate_LL(A)

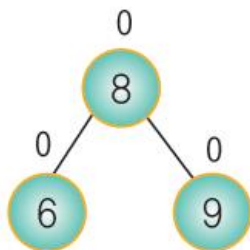
B의 오른쪽 자식을 A의 왼쪽 자식으로 만든다
A를 B의 오른쪽 자식 노드로 만든다.



RR 회전방법



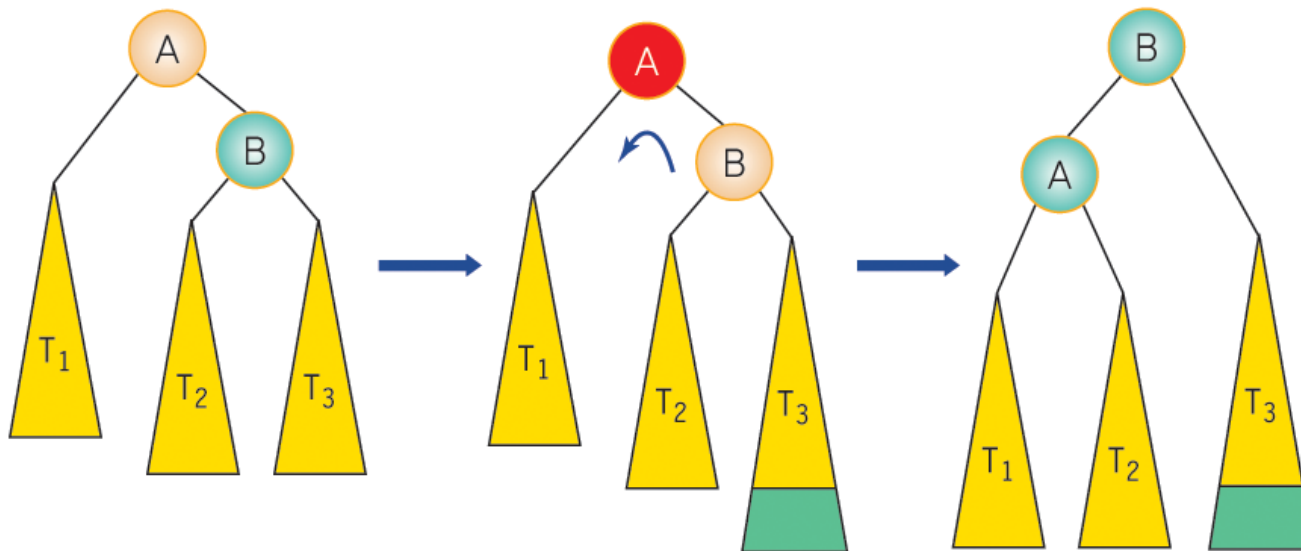
(a) RR 타입



(b) RR 회전 결과

rotate_RR(A)

B의 왼쪽 자식을 A의 오른쪽 자식으로 만든다
A를 B의 왼쪽 자식 노드로 만든다.

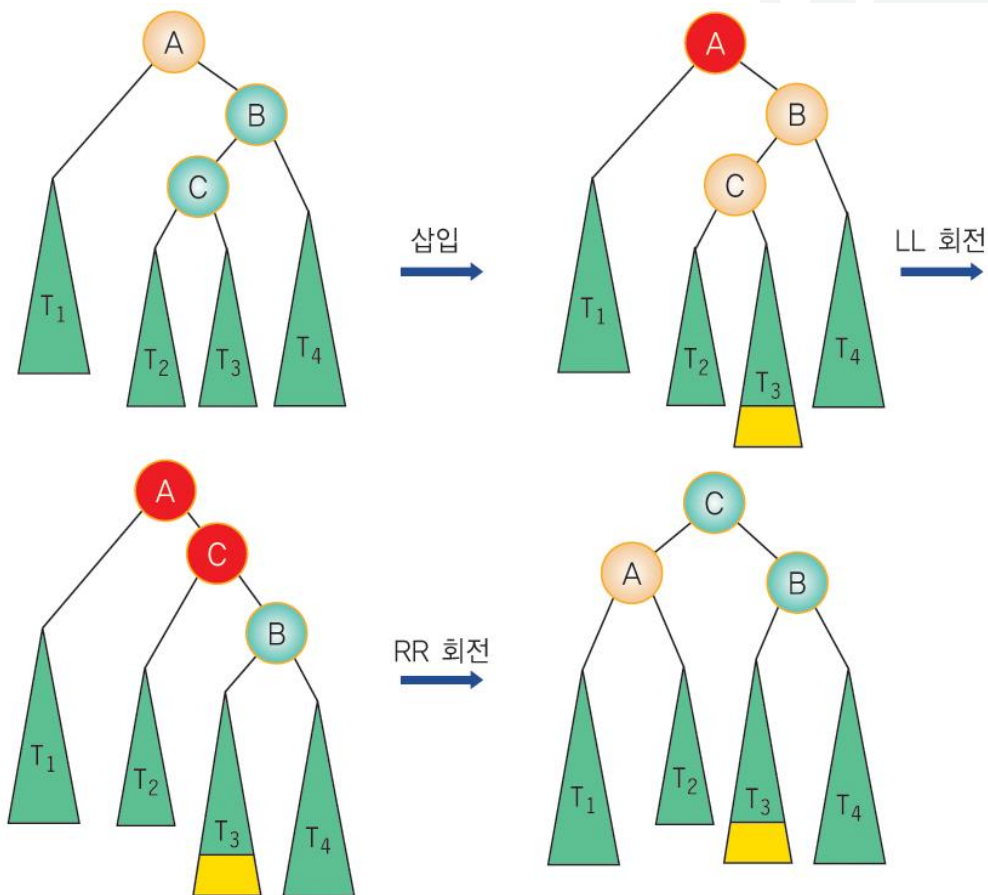


RL 회전방법

rotate_RL(A)

rotate_LL(B)가 반환하는 노드를 A의 오른쪽 자식으로 만든다

rotate_RR(A)

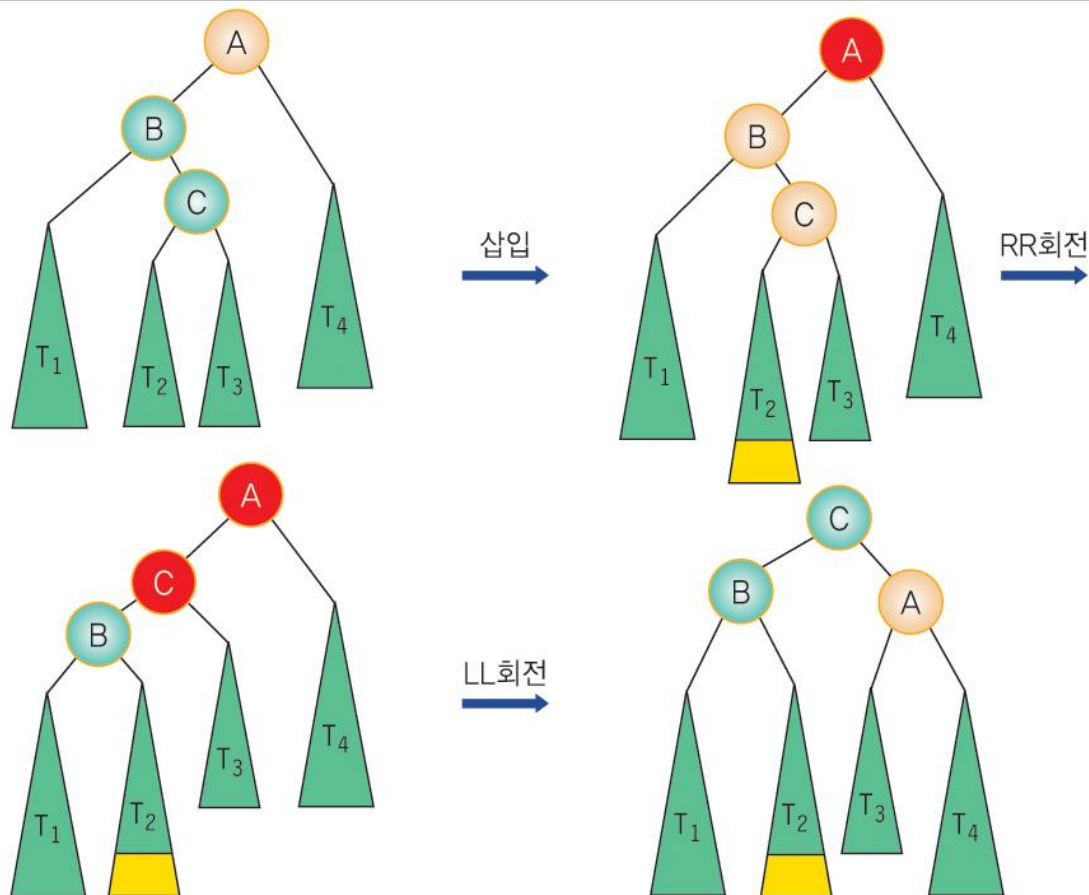


LR 회전방법

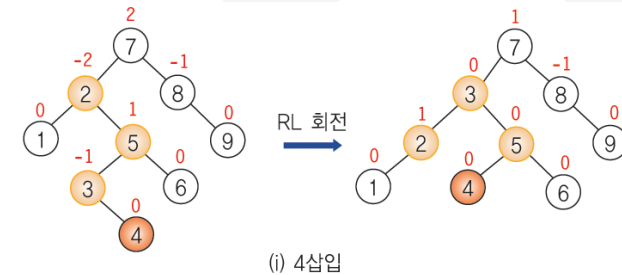
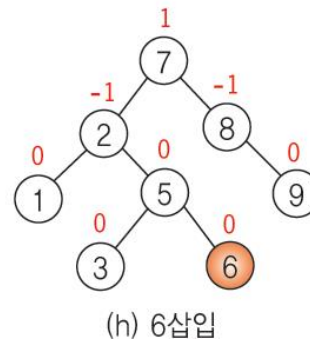
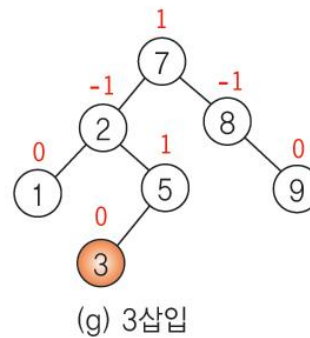
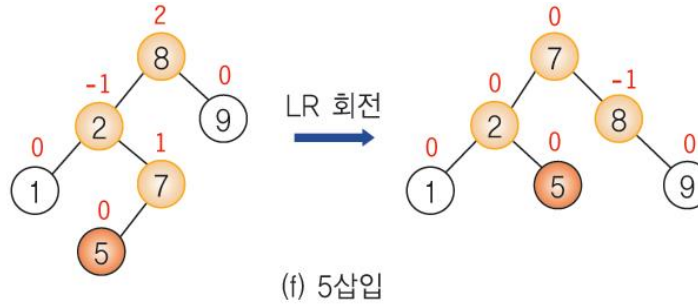
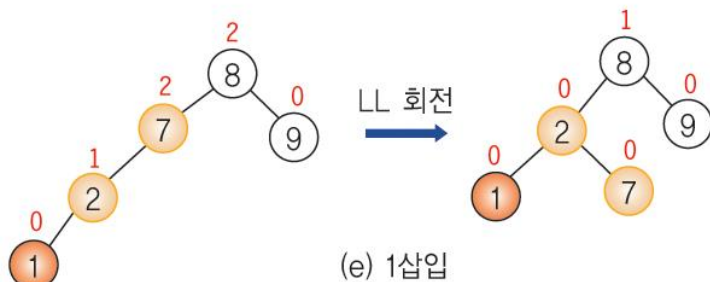
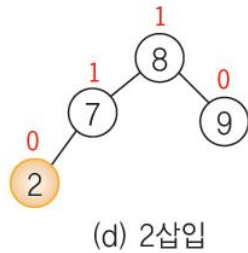
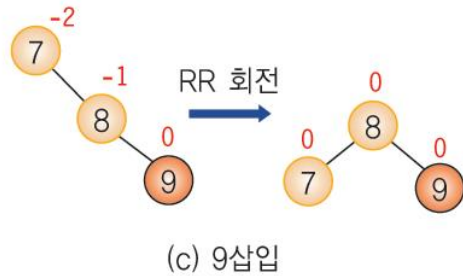
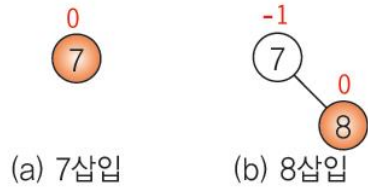
rotate_LR(A)

rotate_RR(B)가 반환하는 노드를 A의 왼쪽 자식으로 만든다

rotate_LL(A)



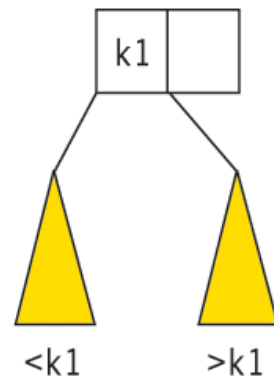
AVL 트리의 삽입연산



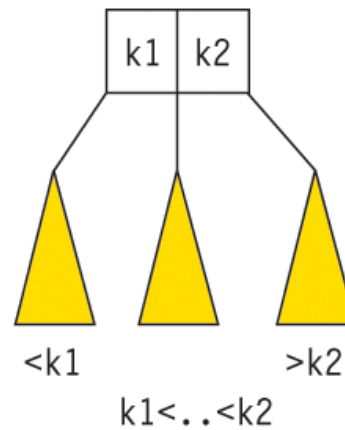
2-3 트리

- 차수가 2 또는 3인 노드를 가지는 트리
- 2-노드
 - 이진탐색트리 처럼 하나의 데이터 k_1 과 두 개의 자식 노드를 가짐
- 3-노드
 - 2개의 데이터 k_1, k_2 와 3개의 자식노드를 가짐
- 왼쪽 서브 트리에 있는 데이터들은 모두 k_1 보다 작은 값
- 중간 서브 트리에 있는 값들은 모두 k_1 보다 크고 k_2 보다 작음
- 오른쪽에 있는 데이터들은 모두 k_2 보다 큼

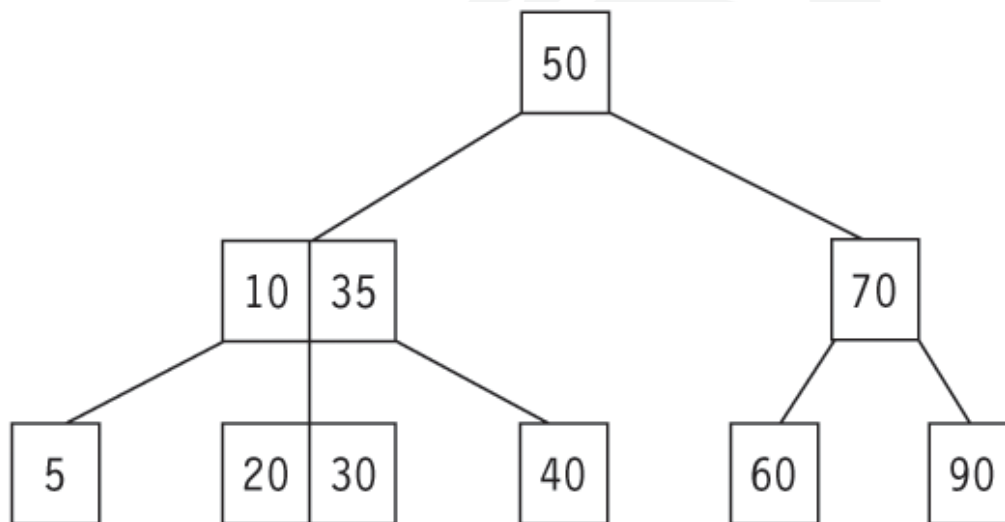
2-3 트리 예



(a) 2 - 노드



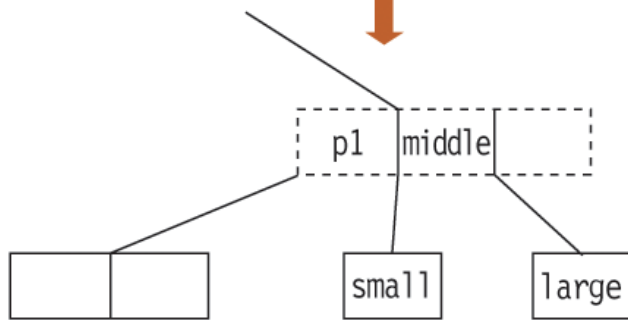
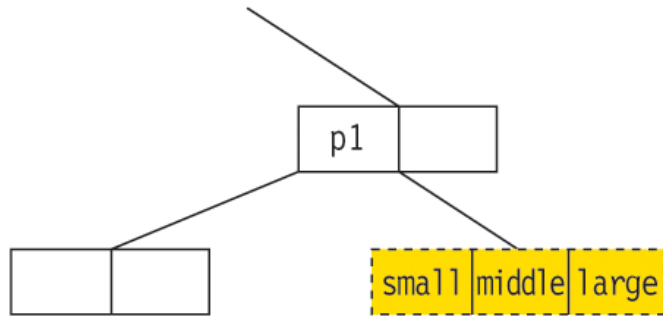
(b) 3 - 노드



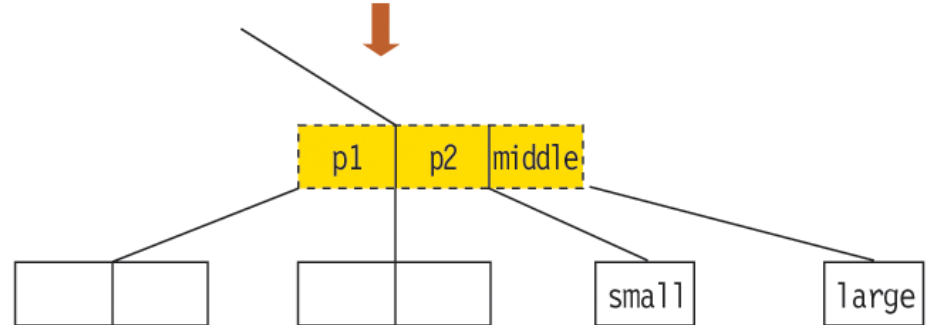
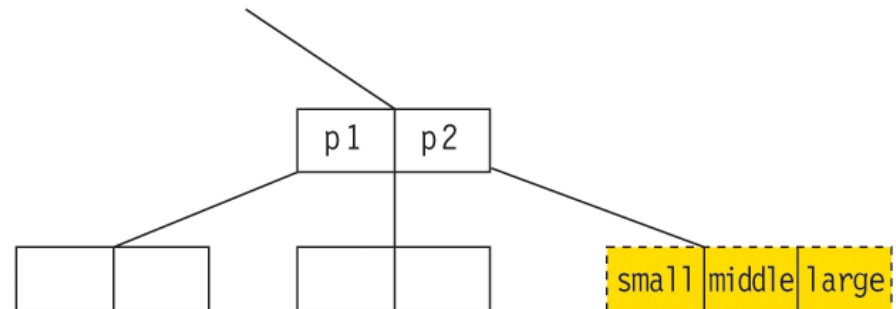
2-3 트리 삽입



2-3 트리 단말노드 분리

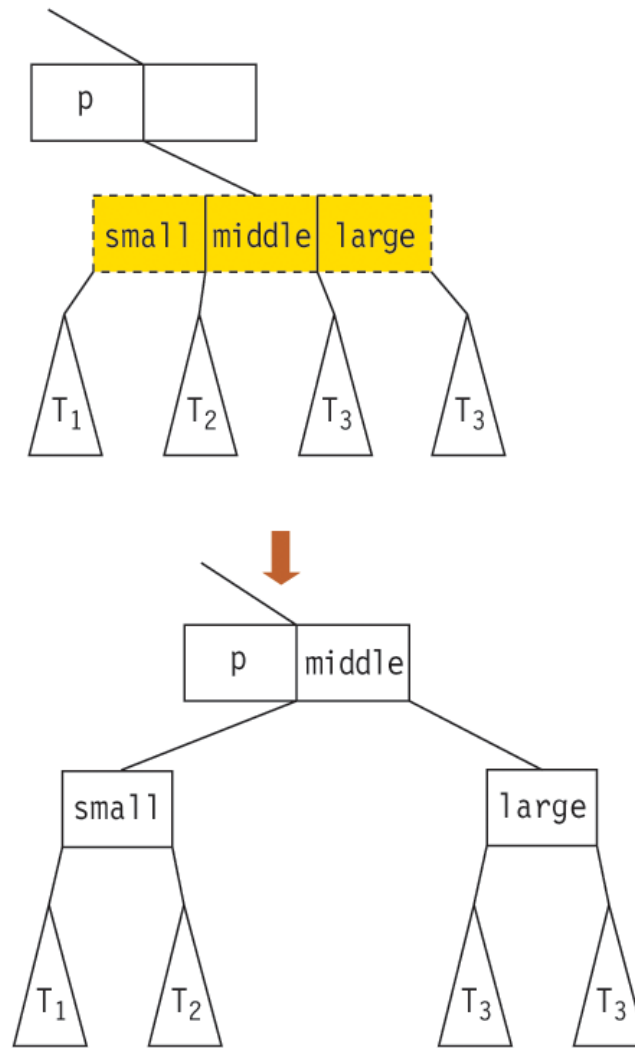


(a) 부모 노드가 2-노드인 경우

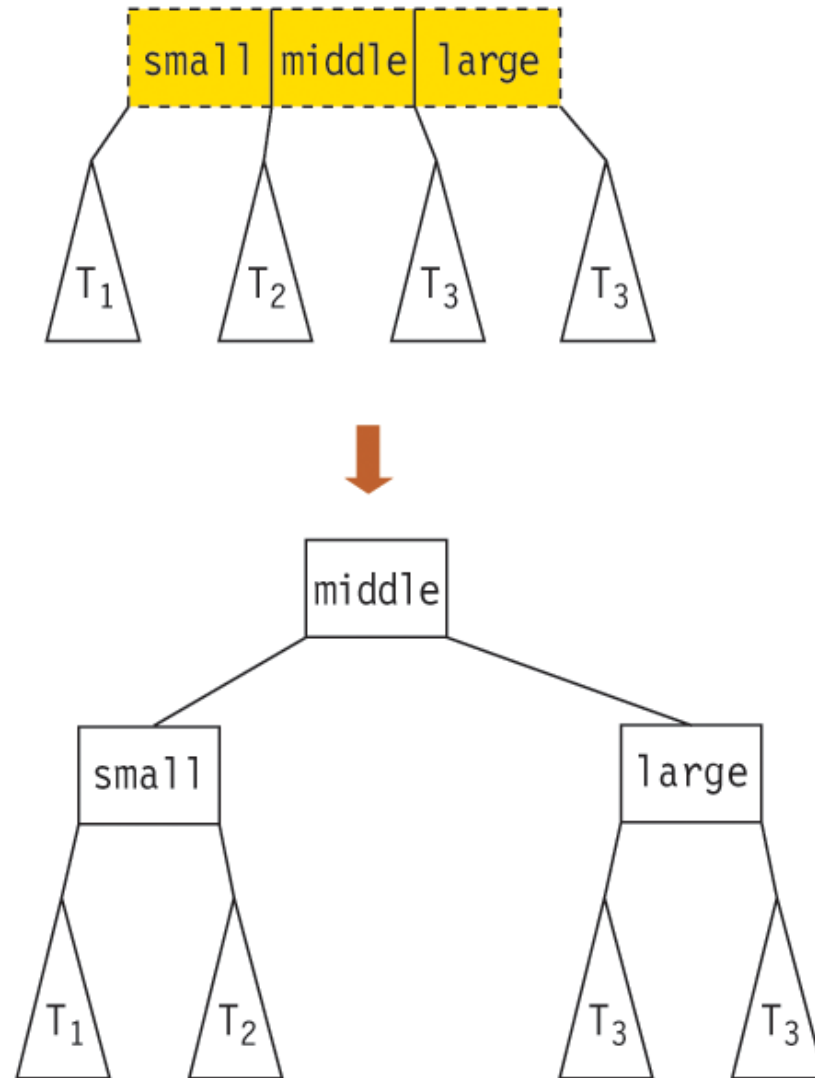


(b) 부모 노드가 3-노드인 경우

2-3 트리 비단말노드 분리



2-3 트리 루트노드 분리



Week 15: Searching Revisited

