10 장. 탐색

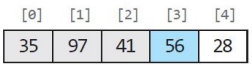
Jinseog Kim

학습 내용

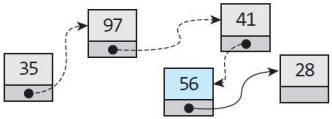
1. 탐색이란?
2. 순차 탐색 (sequential search)
3. 이진 탐색 (binary search)
4. 해싱 (hashing)
5. 심화 학습: 트리를 이용한 탐색

순차 탐색 (sequential search)

- 일렬로 늘어선 자료에서 원하는 레코드를 찾는 방법

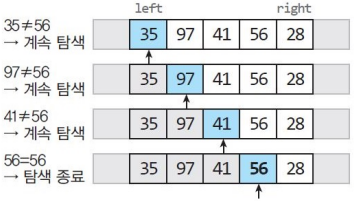


배열 구조의 테이블(56 탐색)

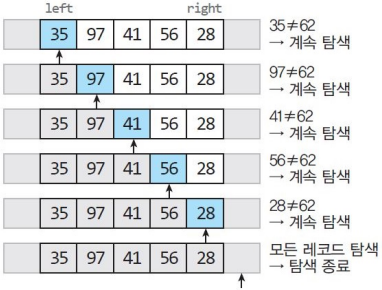


연결된 구조의 테이블(56 탐색)

- 순차 탐색의 예



(a) 56 탐색 → 탐색 성공



(b) 62 탐색 → 탐색 실패

순차 탐색 알고리즘

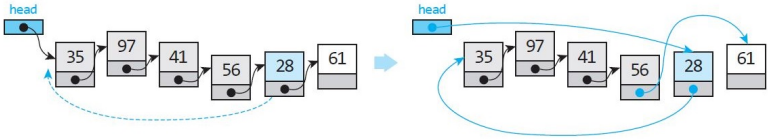
```
1  sequential_search(A[], key, left, right)
2      for i<-left to right :
3          if A[j] == key :
4              return i
5      return -1
```

- 분석
  - 최선의 경우 $O(1)$: 찾는 레코드가 맨 앞에
  - 최악의 경우 $O(n)$: 찾는 레코드가 없는 경우
  - 효율적이지는 않지만, 테이블이 정렬되어 있지 않다면 별다른 대안은 없음

순차 탐색 개선 방안?

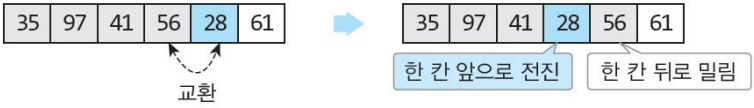
- 자기 구성 (self-organizing) 순차 탐색
  - 자주 탐색 되는 레코드를 테이블의 앞쪽으로 옮겨 탐색 효율을 높이려고 함
- 맨 앞으로 보내기 (move to front) 전략
  - 배열 구조의 테이블

  - 연결된 구조의 테이블



순차 탐색 개선 방안?

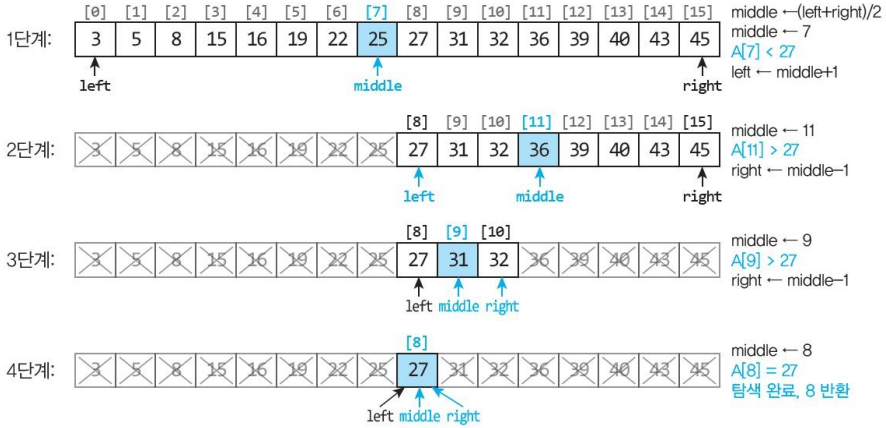
- 교환하기 (transpose) 전략



- 기타 전략
  - 레코드마다 탐색 된 횟수를 저장하고, 탐색 된 횟수가 많은 순으로 테이블을 재구성
- 사용 조건
  - **지금까지 더 많이 탐색 된 레코드가 앞으로도 더 많이 탐색 될 가능성이 큰** 응용에만 사용해야 함

이진 탐색 (binary search)

- 한 번 비교할 때마다 탐색 범위가 절반으로 줄어듦
  - 사전에서 단어를 찾는 과정과 비슷함
  - 테이블의 모든 레코드가 오름차순으로 정렬되어 있어야 적용
  - 정렬된 배열에서 27 을 탐색하는 예



이진 탐색 알고리즘

- 알고리즘

```
1  binary_search(A[], key, left, right)
2      if left <= right :
3          middle <- (left + right)/2
4          if key == A[middle] : return middle
5          else if key < A[middle] :
6            return binary_search(A, key, left, middle - 1)
7          else :
8            return binary_search(A, key, middle + 1, right)
9      return -1          // 탐색실패
10
```

- 순환 구조와 반복 구조로 구현 가능

이진 탐색 알고리즘 분석

- 탐색: 순환 호출을 할 때마다 탐색 범위가 절반이 됨

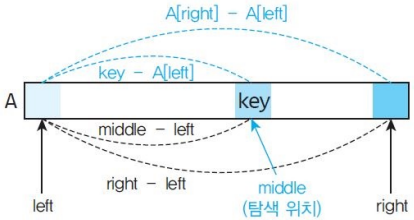
$$2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \ldots \rightarrow 2^1 \rightarrow 2^0$$

  - 탐색 연산의 성능: $O(\log n)$
  - 테이블이 정렬되어 있어야만 사용 가능
- 삽입/삭제 연산의 성능은? $O(n)$
  - 테이블이 배열이라면 $\Rightarrow$ 레코드의 많은 이동이 필요
- 응용분야에 따른 특징
  - 테이블이 한번 만들어지면 자주 변경되지 않고 탐색 연산만 주로 처리하는 응용이라면 $\Rightarrow$ 매우 효율적
  - 삽입/삭제 연산이 빈번한 응용이라면? $\Rightarrow$ 효율적이지 않음

    이진탐색트리와 같은 다른 방법이 좋을 수도 있음 (AVL 트리 등)

보간 탐색 (interpolation search)

- 탐색 키가 존재할 위치를 예측하여 탐색하는 방법
  - 이진 탐색을 개선
  - 레코드의 킷값과 탐색 키의 비율을 고려해 탐색 위치 계산: 찾는 값과 위치가 비례한다고 가정을 바탕으로 함

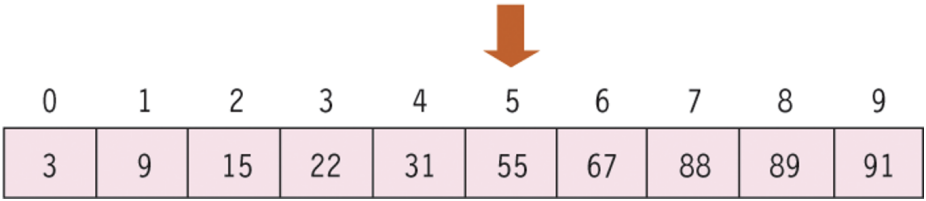


$$(middle) = left + (right - left)\frac{key - A[left]}{A[right] - A[left]}$$

- 데이터가 균등하게 분포된 자료에 훨씬 효율적

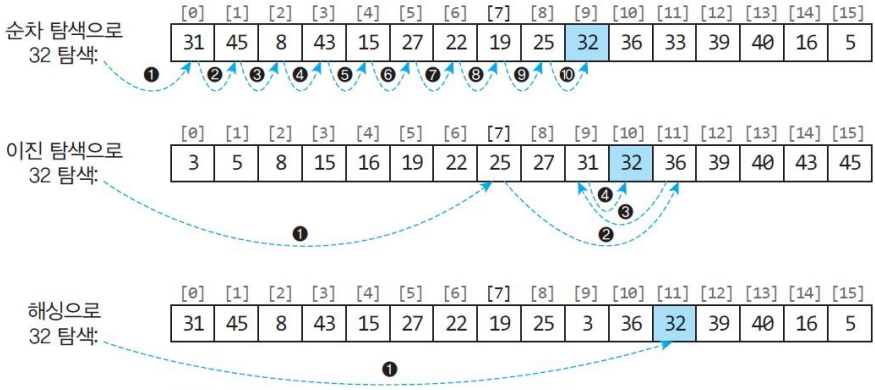
탐색 위치 계산 예

탐색 위치 = (55-3)/(91-3)*(9-0) + 0 = 5.31 ≒ 5



해싱 (hashing)

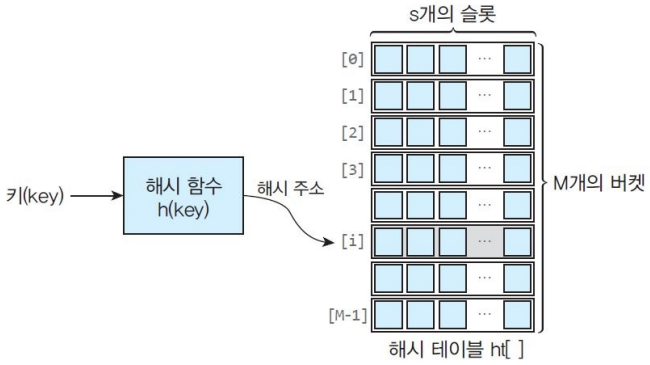
- 키에 산술적인 연산을 적용해 레코드가 있는 위치를 계산해서 바로 찾아가는 탐색 방법
  - 최강의 탐색 방법



32의 해시 주소 계산: 예) hash_function(32) = 11
해당 주소에서 탐색: 32가 테이블에 있다면 반드시 11번지에 있어야 함

해싱의 구조

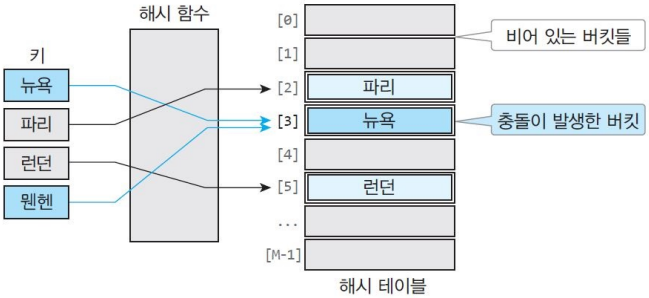
- 용어
  - 해시 테이블
  - 버킷
  - 슬롯
  - 해시 함수
  - 해시 주소



- 해시 함수
  - 키값에서부터 레코드의 위치인 해시 주소를 계산
  - 탐색, 삽입, 삭제 연산이 모두 이 주소에서 이루어짐

해싱의 구조

- 해시 충돌 (hash collision)
  - 두 개의 서로 다른 키가 같은 주소로 계산되는 상황
  - 예: **h(뉴욕)=3**, h(파리) = 2, h(런던) = 5, **h(뭰헨)=3**



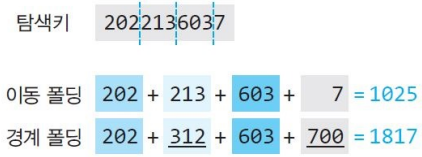
- 해시 오버플로 (overflow)
  - 충돌이 슬롯 수보다 더 많이 발생하는 경우

해싱의 구조

- 이상적인 해싱
  - 충돌이 절대 일어나지 않는 경우 ⇒ 지나치게 많은 메모리 필요
  - 현실적으로 불가능
- 실제의 해싱
  - 테이블의 크기를 적절히 줄이고,
  - 해시 함수를 이용해 주소를 계산
  - 충돌과 오버플로가 빈번하게 발생함
- 해싱에서는 **오버플로 처리 방안이 반드시 필요**
  - 개방 주소법
  - 체이닝

해시 함수

- 좋은 해시 함수
  - 계산이 빠르고, 충돌이 적게 발생하고,
  - 결과가 테이블의 주소 영역 내에서 고르게 분포
- 해시 함수들
  - 제산 함수: $h(k) = k \bmod M$
  - 폴딩 함수



  - 중간 제곱 함수
  - 비트 추출 방법
  - 숫자 분석 방법
  - 탐색 키가 문자열인 경우: 먼저 각 문자를 정수로 대응시켜야 함

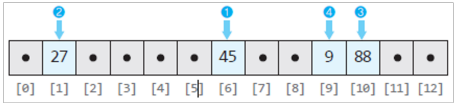
오버플로 처리: 개방 주소법

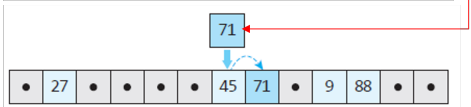
- 개방 주소법 (open addressing)
  - 주소를 개방해 테이블의 다른 주소에서 처리할 수 있도록 허용
  - 예: 선형 조사법 (linear probing)
- 선형 조사법 (linear probing)
  - ht[k] 에서 충돌이 발생하면 다음 위치인 ht[k+1] 부터 순서대로 비어 있는지를 살피고, 빈 곳이 있으면 저장
- 선형 조사법의 예: M=13, h(k)=k%M
  - 삽입 연산, 탐색 연산, 삭제 연산

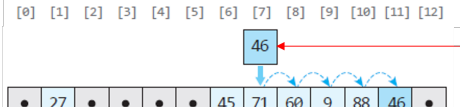
dongguk UNIVERSITY WISE

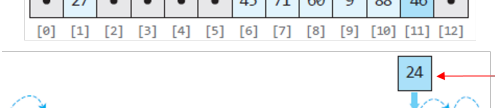
삽입 연산

key	45	27	88	9	71	60	46	38	24
h(key)	6	1	10	9	6	8	7	12	11





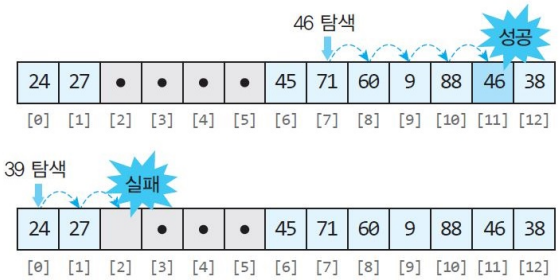




군집화!

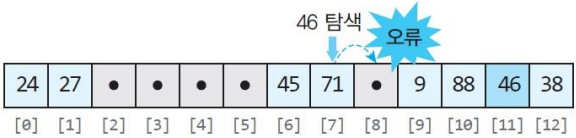
탐색 연산

- 탐색 키가 입력되면 해시 주소를 계산
- 그 주소에 같은 키의 레코드가 있으면 탐색 성공
- 없으면 다음 버킷 조사
  - 해당 키의 레코드를 찾거나,
  - 레코드가 없는 버킷을 만나거나,
  - 모든 버킷을 다 검사할 때까지 진행



삭제 연산

- 빈 버킷을 두 가지로 구분해야 함
  - 한 번도 사용하지 않은 것
  - 사용되었다가 삭제되어 현재 비어 있는 것

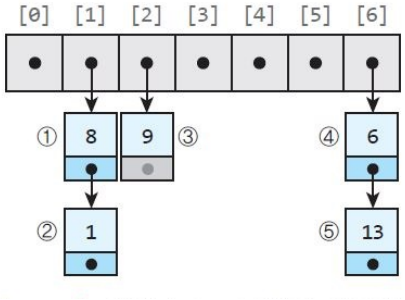


군집화 완화 방법

- 충돌이 발생했을 때 다음에 조사할 위치에 변화를 주는 방법
- 이차 조사법 (quadratic probing)
  - h[k] , h[k]+1, h[k]+4, h[k]+9 와 같이 제곱 순으로 움직이는 방법
- 이중 해싱법 (double hashing) 또는 재해싱 (rehashing) -원래 해시 함수와 다른 별개의 함수를 이용

오버플로 처리: 체이닝 (chaining)

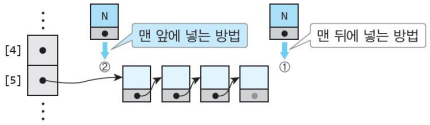
- 주소를 개방하지 않고 계산된 해시 주소 안에서 어떻게든 처리하는 방법
  - 버킷을 연결 리스트와 같은 구조로 변경해 아무리 많은 충돌이 발생해도 저장할 수 있도록 함
  - 예: M=7, h(k)=k % 7, 킷값 8, 1, 9, 6, 13 을 삽입



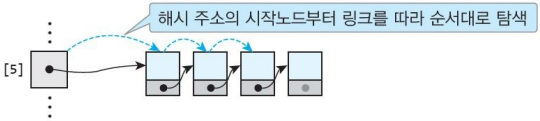
- 모든 연산이 해시 주소에 해당하는 버킷에서 만 이루어 짐

오버플로 처리: 체이닝 (chaining)

- 체이닝의 삽입 연산 (맨 앞이 유리)



- 체이닝의 탐색 연산



- 체이닝의 삭제 연산

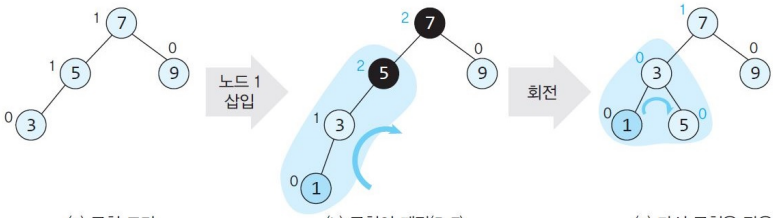
- 이진 탐색 트리의 균형 기법 ⇒ 다양함
- **AVL 트리**
  - Adelson-Velskii 와 Landis 에 의해 제안된 트리
  - 항상 균형 트리를 보장 ⇒ 탐색, 삽입, 삭제 모두 O(log n) 보장
  - 균형 인수가 1 이하인 트리로 정의

(a) 이진 탐색 트리: AVL 트리 (b) 노드 1 삽입: 균형이 깨짐

- 삽입과 삭제 연산에서 균형이 깨질 수 있음

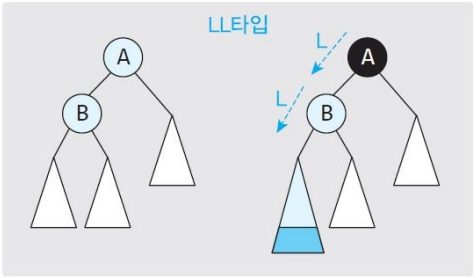
AVL 트리의 삽입 연산

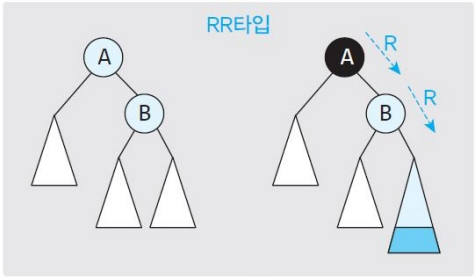
- 불균형이 발생하면 회전을 이용해 다시 균형을 잡음

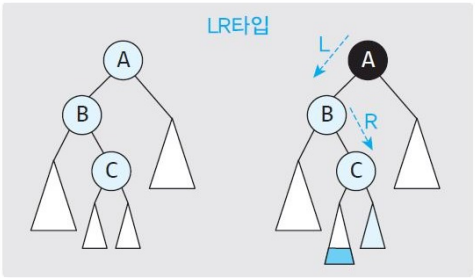


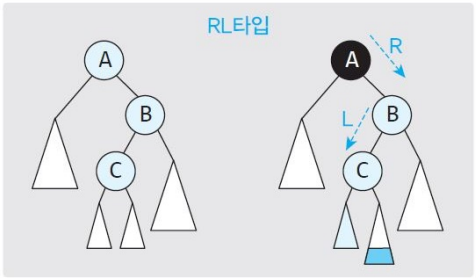
(a) 균형 트리 (b) 균형이 깨짐(5, 7) (c) 다시 균형을 잡음

균형이 깨지는 네 가지 경우



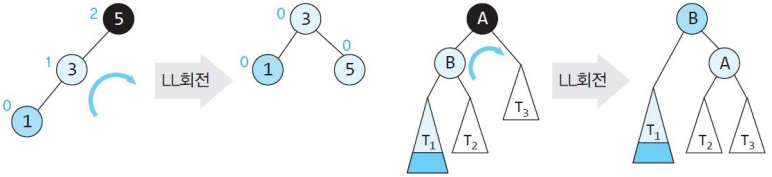






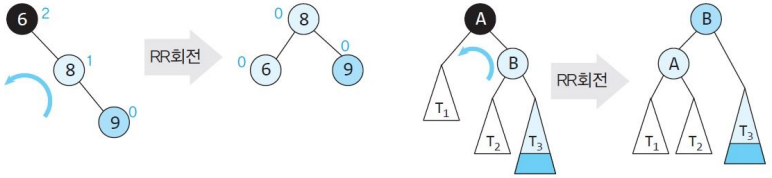
균형이 깨지는 네 가지 경우

LL 회전



(a) LL회전의 예 (b) 일반적인 LL회전

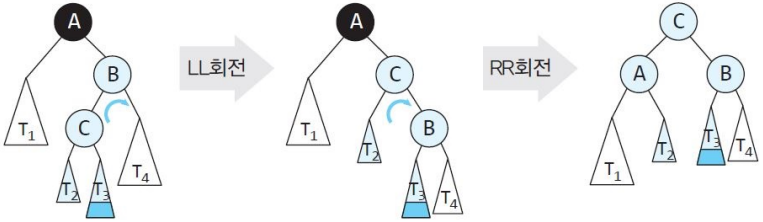
RR 회전



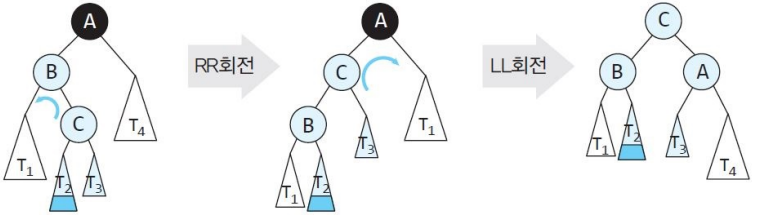
(a) RR회전의 예 (b) 일반적인 RR회전

균형이 깨지는 네 가지 경우

RL 회전

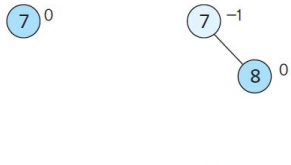


LR 회전



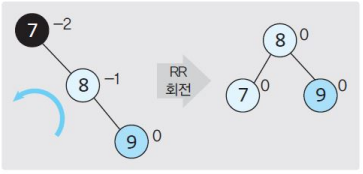
AVL 트리 구축 예

- 공백 트리에 [7, 8, 9, 2, 1, 5, 3, 6, 4] 를 삽입하는 과정

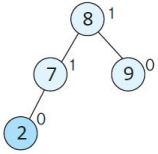


(a) 7삽입

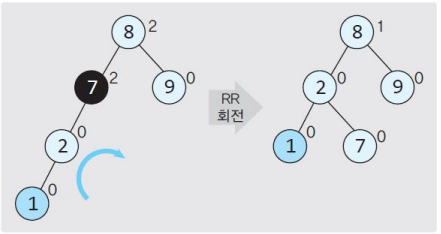
(b) 8삽입



(c) 9삽입: RR회전

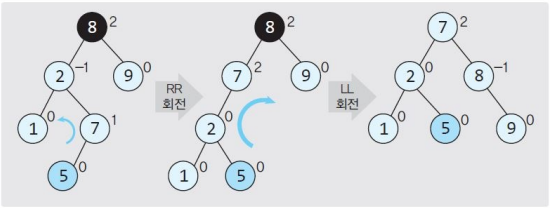


(d) 2삽입

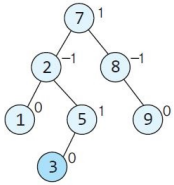


(e) 1삽입: LL회전

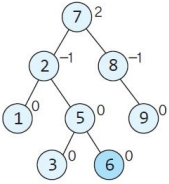
AVL 트리 구축 예



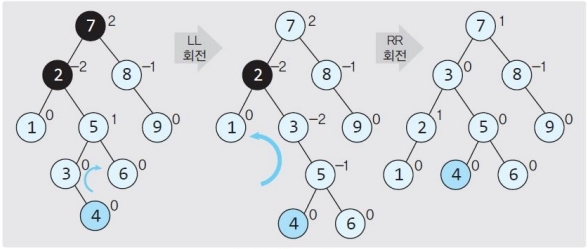
(f) 5삽입: LR회전



(g) 3삽입



(h) 6 삽입



(i) 4삽입: RL 회전

연습문제