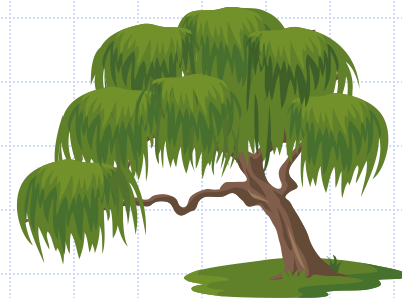


탐색트리



Outline

- ◆ 11.1 이진탐색트리
- ◆ 11.2 AVL 트리
- ◆ 11.3 스펠레이 트리
- ◆ 11.4 응용문제



이진탐색트리

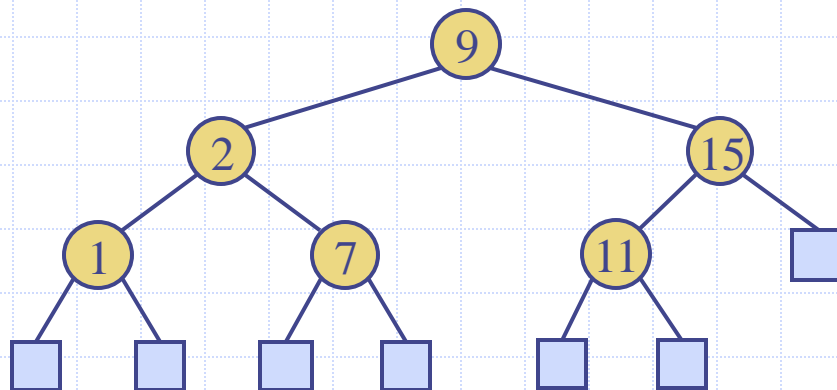
◆ 이진탐색트리(binary search tree): 내부노드에 (키, 원소) 쌍을 저장하며 다음의 성질을 만족하는 이진트리

- u, v, w 는 모두 트리노드며 u 와 w 가 각각 v 의 왼쪽과 오른쪽 부트리에 존재할 때 다음이 성립
 $key(u) < key(v) \leq key(w)$

◆ 전제: 적정이진트리로 구현

◆ 그림 표기: 내부노드 내에 간단히 키만 표시

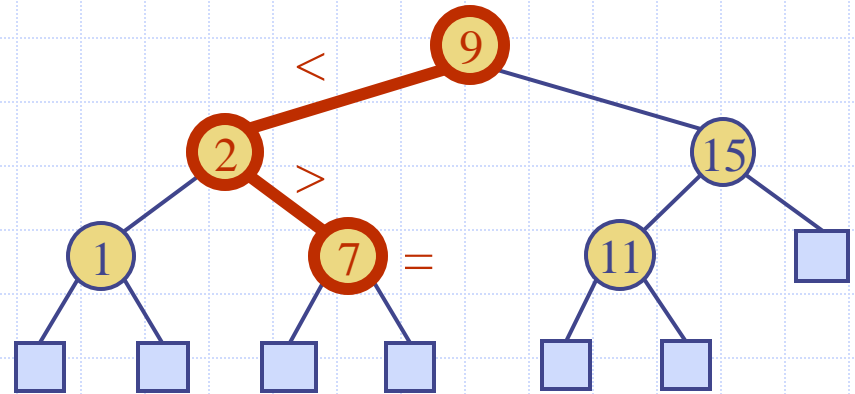
- ◆ 이진탐색트리를 중위순회(inorder traversal)하면 키가 증가하는 순서로 방문
- ◆ 이진탐색트리 예:





탐색

- ◆ 키 k 를 찾기 위해, 루트에서 출발하는 하향 경로를 추적
- ◆ 다음에 방문할 노드는 k 와 현재 노드의 키의 크기를 비교한 결과에 따라 결정
- ◆ 잎(즉, 외부노드)에 다다르면, 키 k 가 발견되지 않은 것이므로 *NoSuchKey*를 반환
- ◆ 예: `findElement(7)`



탐색 (conti.)

Alg *findElement(k)*

input binary search tree T , key k

output element with key k

1. $w \leftarrow \text{treeSearch}(\text{root}(), k)$
2. **if** ($\text{isExternal}(w)$)
 return *NoSuchKey*
 else
 return *element(w)*

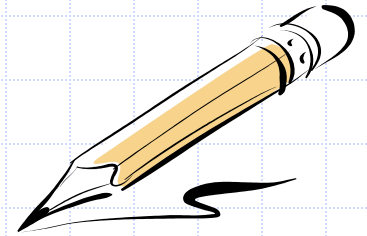
Alg *treeSearch(v, k)* { generic }

input node v of a binary search tree,
key k

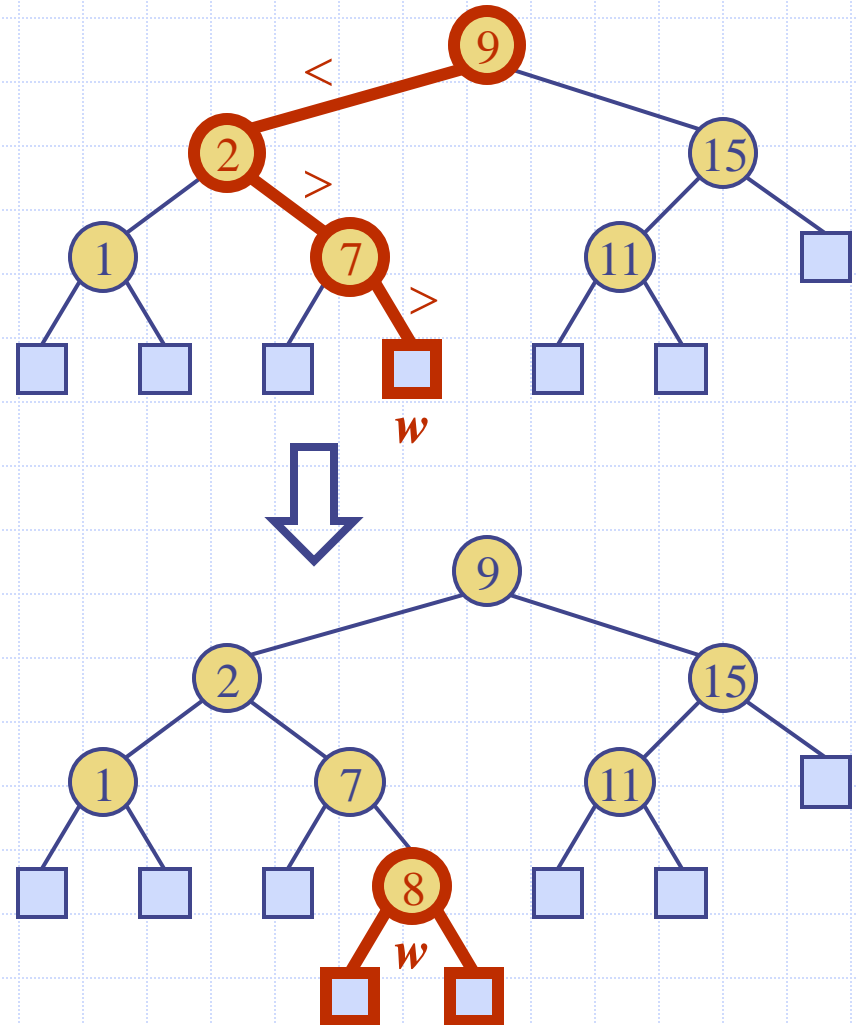
output node w , s.t. either w is an
internal node storing key k or w is
the external node where key k would
belong if it existed

1. **if** ($\text{isExternal}(v)$)
 return v
2. **if** ($k = \text{key}(v)$)
 return v
 elseif ($k < \text{key}(v)$)
 return $\text{treeSearch}(\text{leftChild}(v), k)$
 else $\{k > \text{key}(v)\}$
 return $\text{treeSearch}(\text{rightChild}(v), k)$

삽입



- ◆ **insertItem**(k, e) 작업을 수행하기 위해, 우선 키 k 를 탐색
- ◆ k 가 트리에 존재하지 않을 경우, 탐색은 외부노드(w 라 하자)에 도착
- ◆ 외부노드 w 에 k 를 삽입한 후 **expandExternal**(w) 작업을 사용하여 w 를 내부노드로 확장
- ◆ 예: **insertItem**(8, e)



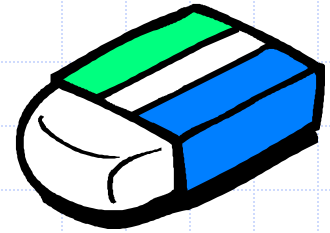
삽입 (conti.)

Alg *insertItem*(k, e)

input binary search tree T , key k ,
element e

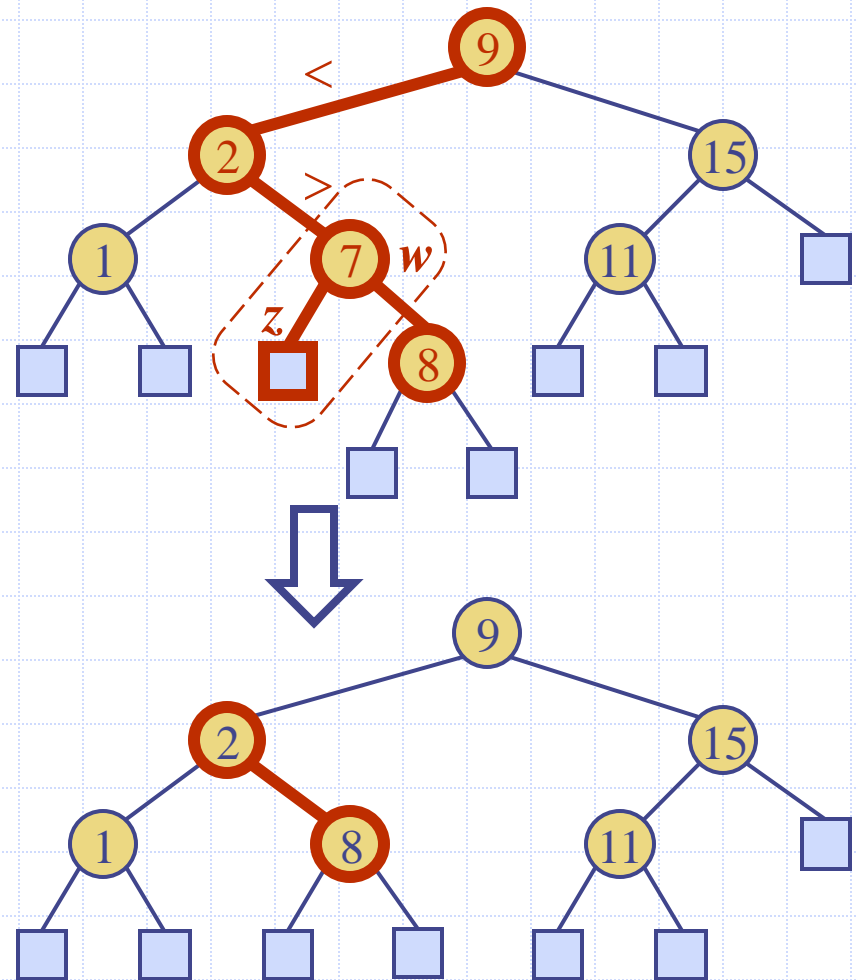
output none

1. $w \leftarrow \text{treeSearch}(\text{root}(), k)$
2. **if** (*isInternal*(w))
 return
else
 Set node w to (k, e)
 expandExternal(w)
 return



삭제: Case 1

- ◆ **removeElement(k)**
작업을 수행하기 위해,
우선 키 k 를 탐색
- ◆ k 가 트리에 존재할 경우,
탐색은 k 를 저장하고
있는 노드(w 라 하자)에
도착
- ◆ 노드 w 의 자식 중
하나가 외부노드(z 라
하자)라면,
reduceExternal(z)
작업을 사용하여 w 와
 z 를 트리로부터 삭제
- ◆ 예: **removeElement(7)**



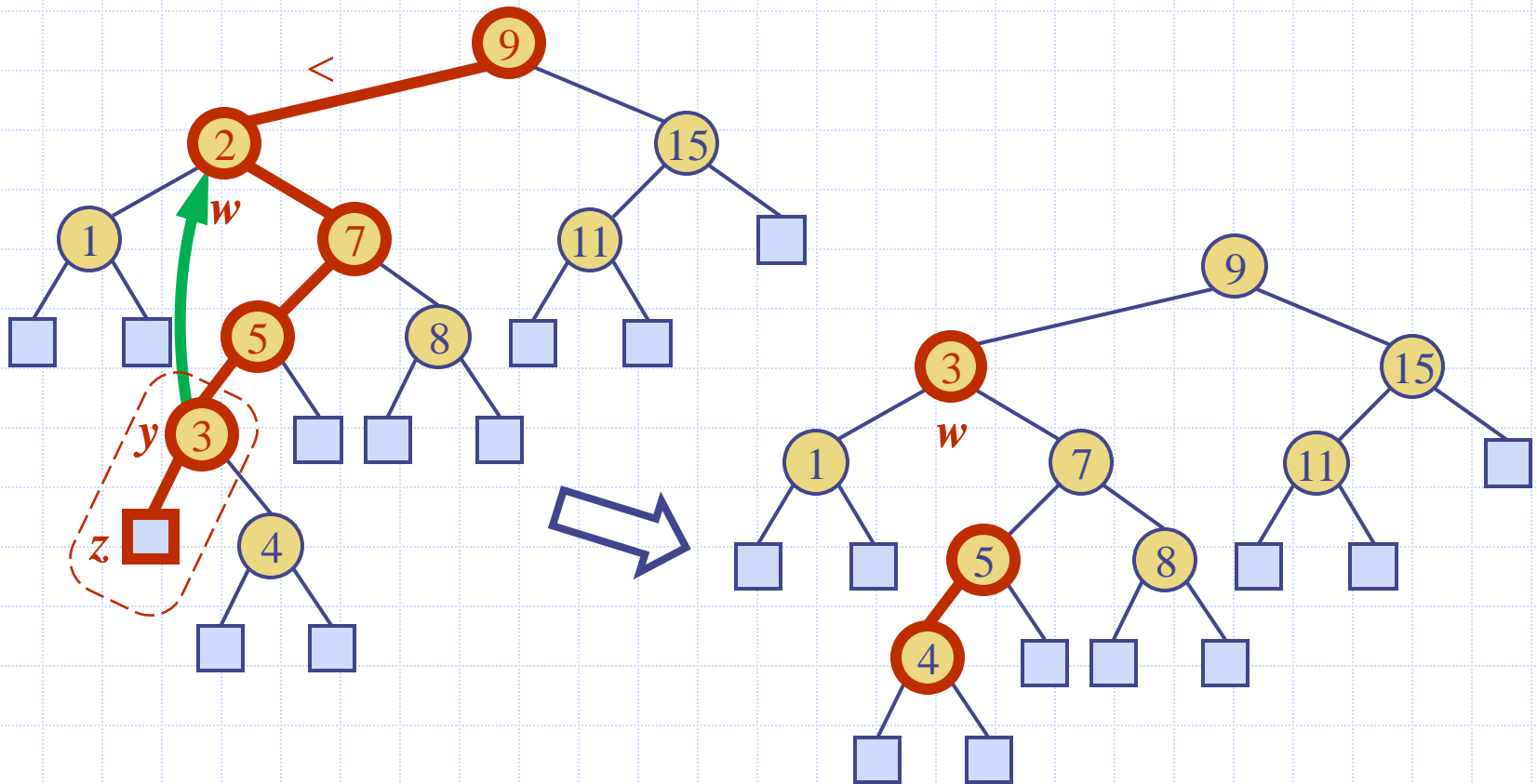
삭제: Case 2

◆ 삭제되어야 할 키 k 가 내부노드만을 자식들로 가지는 노드(w 라 하자)에 저장되어 있다면 다음과 같이 처리

1. 트리 T 에 대해 w 의 **중위순회 계승자** y 와 그 자식노드 z 을 찾아낸다
 - ◆ 노드 y 는 우선 w 의 오른쪽 자식으로 이동한 후, 거기서부터 왼쪽 자식들만을 끝까지 따라 내려가면 도달하게 되는 마지막 내부노드며, 노드 z 은 y 의 왼쪽 자식인 외부노드
 - ◆ y 는 T 를 중위순회할 경우 노드 w 바로 다음에 방문하게 되는 내부노드이므로 w 의 **중위순회 계승자**(inorder successor)라 불린다
 - ◆ 따라서 y 는 w 의 오른쪽 부트리 내 노드 중 **가장 왼쪽으로 돌출된 내부노드**
2. y 의 내용을 w 에 복사
3. **reduceExternal**(z) 작업을 사용하여 노드 y 와 z 를 삭제

삭제: Case 2 (conti.)

예: removeElement(2)



삭제 (conti.)

Alg *removeElement*(*k*)

input binary search tree *T*,
key *k*

output element with key *k*

1. *w* \leftarrow *treeSearch*(*root*(), *k*)
2. **if** (*isExternal*(*w*))
 return *NoSuchKey*

3. *e* \leftarrow *element*(*w*)

4. *z* \leftarrow *leftChild*(*w*)

5. **if** (!*isExternal*(*z*))

z \leftarrow *rightChild*(*w*)

6. **if** (*isExternal*(*z*)) {case 1}

reduceExternal(*z*)

else {case 2}

y \leftarrow *inOrderSucc*(*w*)

z \leftarrow *leftChild*(*y*)

 Set node *w* to (*key*(*y*), *element*(*y*))

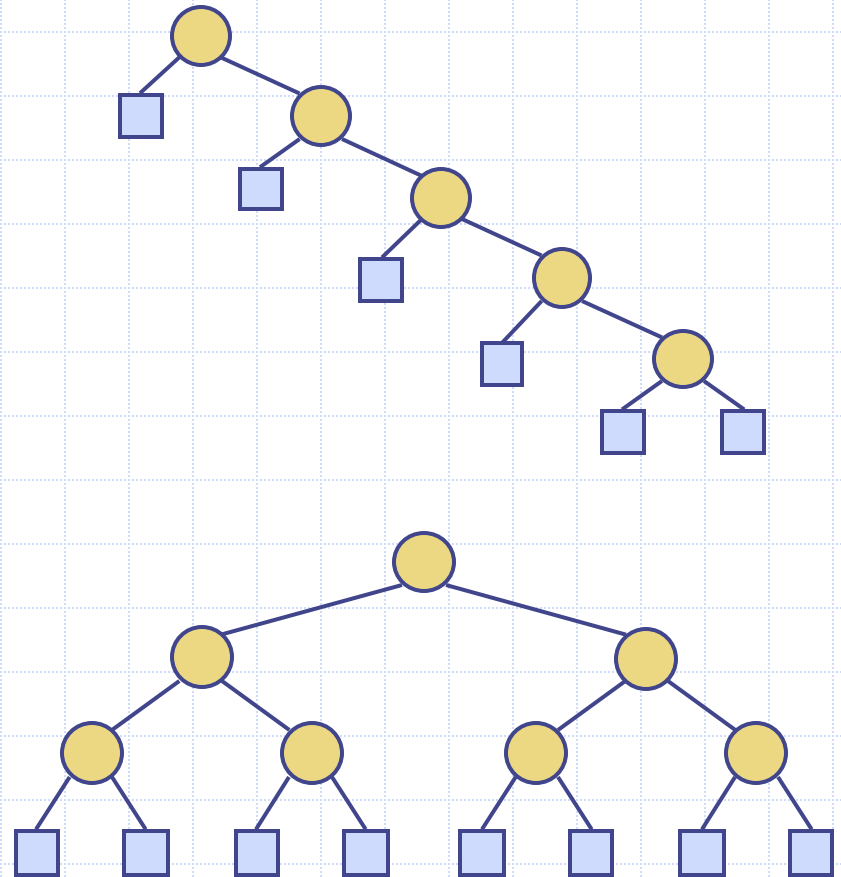
reduceExternal(*z*)

7. **return** *e*

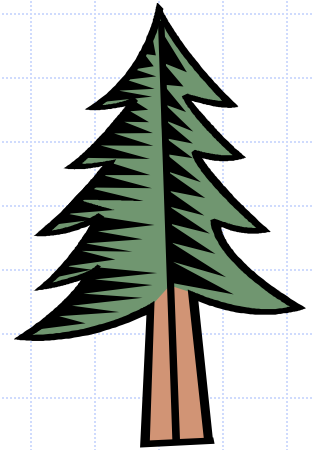
이진탐색트리의 성능

◆ 높이 h 의 이진탐색트리를 사용하여 구현된 n 항목의 사전을 가정하면:

- $O(n)$ 공간 사용
- **findElement**, **insertItem**, **removeElement** 작업 모두 $O(h)$ 시간에 수행
- 높이 h 는:
 - ◆ 최악의 경우 $O(n)$
 - ◆ 최선의 경우 $O(\log n)$

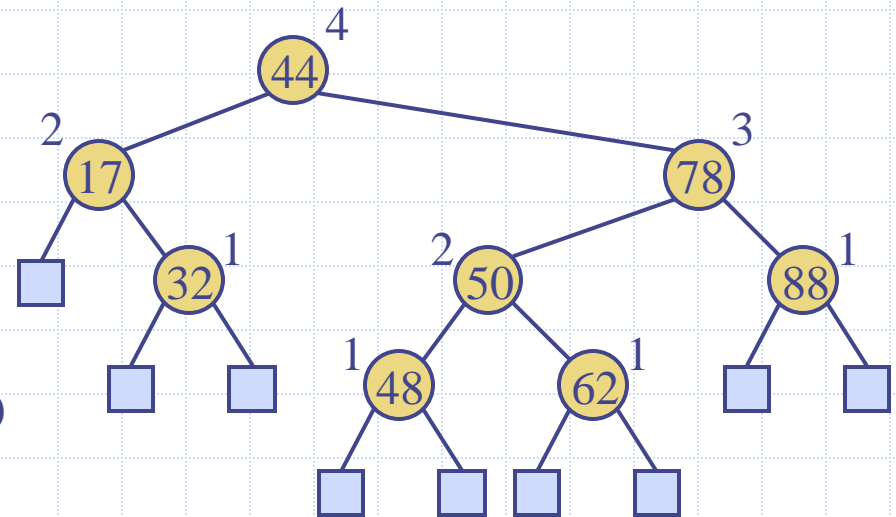


AVL 트리

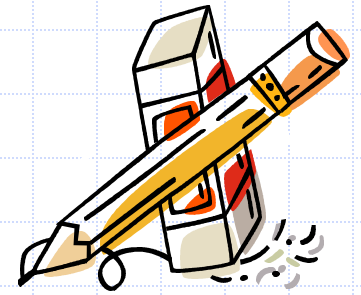


- ◆ AVL 트리: 트리 T 의 모든 내부노드 v 에 대해 v 의 자식들의 좌우 높이 차이가 1을 넘지 않는 이진탐색트리 (즉, **높이균형 속성**, height-balance property)
 - AVL 트리의 부트리 역시 AVL 트리
 - 높이 (또는 균형) 정보는 각 내부노드에 저장
 - n 개의 항목을 저장하는 AVL 트리의 높이: $O(\log n)$
 - **findElement** 작업: $O(\log n)$ 시간 소요

◆ AVL 트리 예:



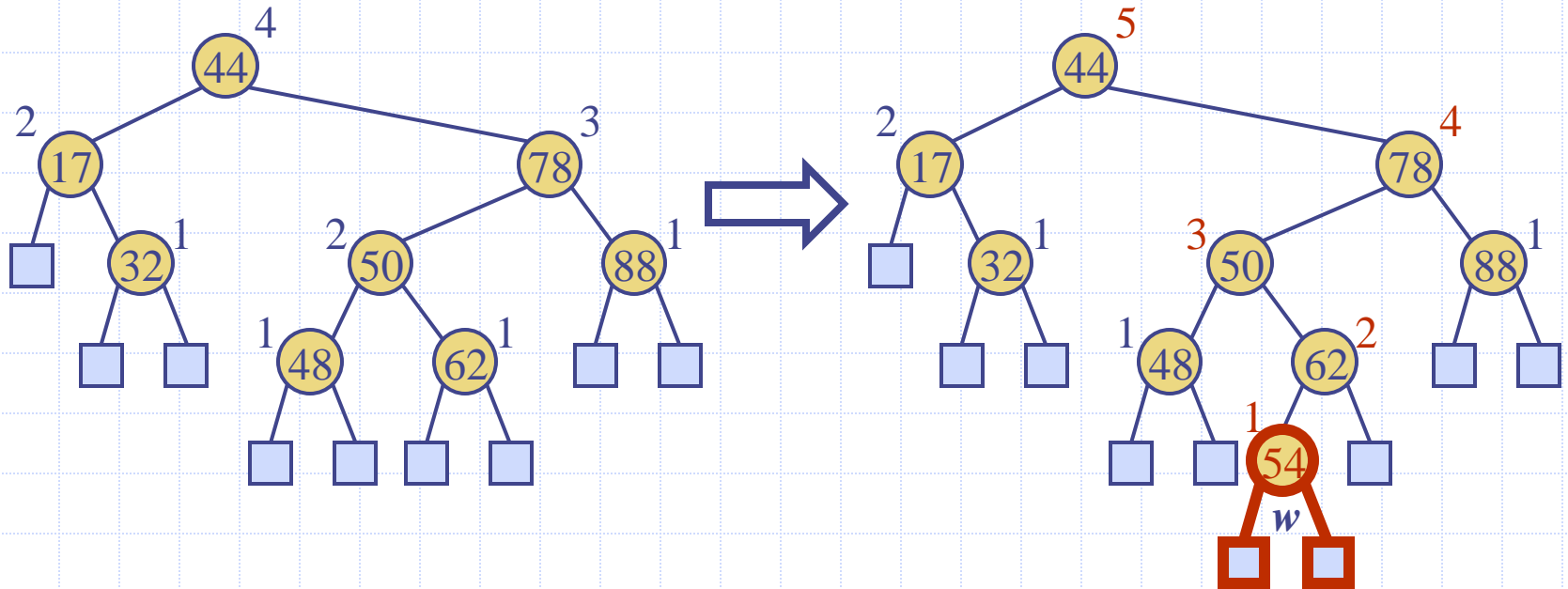
갱신 작업



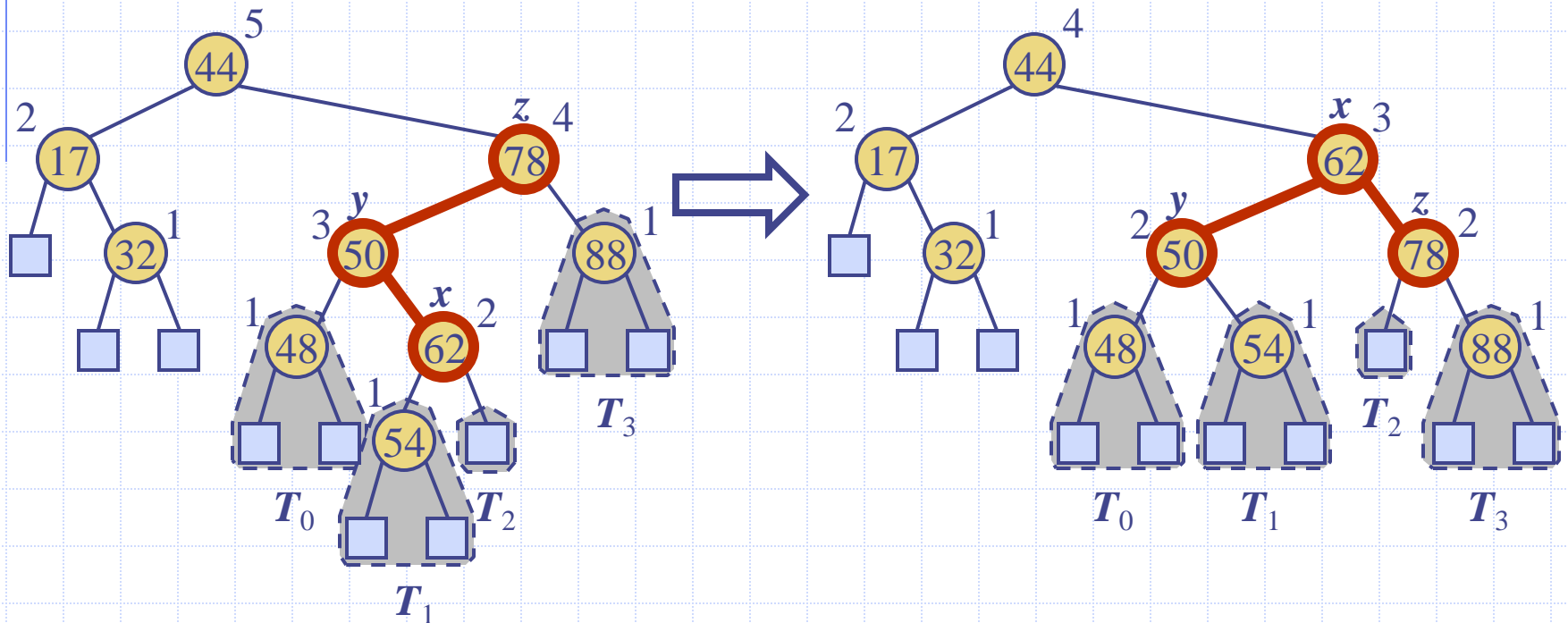
- ◆ AVL 트리에 대한 **삽입** 및 **삭제** 작업은 이진탐색트리에서의 삽입 및 삭제 작업과 유사
- ◆ 삽입이나 삭제 작업의 결과 AVL 트리의 **높이균형 속성**이 파괴될 수도 있다
- ◆ 그러므로 삽입이나 삭제 작업 후에는 혹시 생겼을지도 모를 불균형을 "**찾아서 수리**"해야 한다
 - **불균형 찾기**: 각 노드의 **균형검사**(balance check)를 통해 찾는다
 - **불균형 수리**: **개조**(restructure)라 불리는 작업을 통해 트리의 높이균형 속성을 회복하기 위한 계산 작업을 수행

AVL 트리에서 삽입

- ◆ 삽입은 이진탐색트리에서와 동일하게 수행
- ◆ **expandExternal** 작업에 의해 확장된 노드 w (그리고 조상노드들)가 균형을 잃을 수 있다
- ◆ 예: **insertItem(54, e)**



삽입 후 개조



삽입

Alg *insertItem*(k, e)

input AVL tree T , key k , element e

output none

1. $w \leftarrow \text{treeSearch}(\text{root}(), k)$
2. **if** (*isInternal*(w))
 return
else
 Set node w to (k, e)
 expandExternal(w)
 searchAndFixAfterInsertion(w)
 return

삽입 (conti.)

Alg *searchAndFixAfterInsertion*(w)

input internal node w

output none

1. w 에서 T 의 루트로 향해 올라가다가 처음 만나는 불균형 노드를 z 이라 하자(그러한 z 이 없다면 **return**).
2. z 의 높은 자식을 y 라 하자.
{수행 후, y 는 w 의 조상이 되는 것에 유의}

3. y 의 높은 자식을 x 라 하자.

{수행 후, 노드 x 가 w 와 일치할 수도 있으며 x 가 z 의 손자임에 유의. y 의 높이는 그의 형제의 높이보다 2가 더 많다}

4. *restructure*(x, y, z)

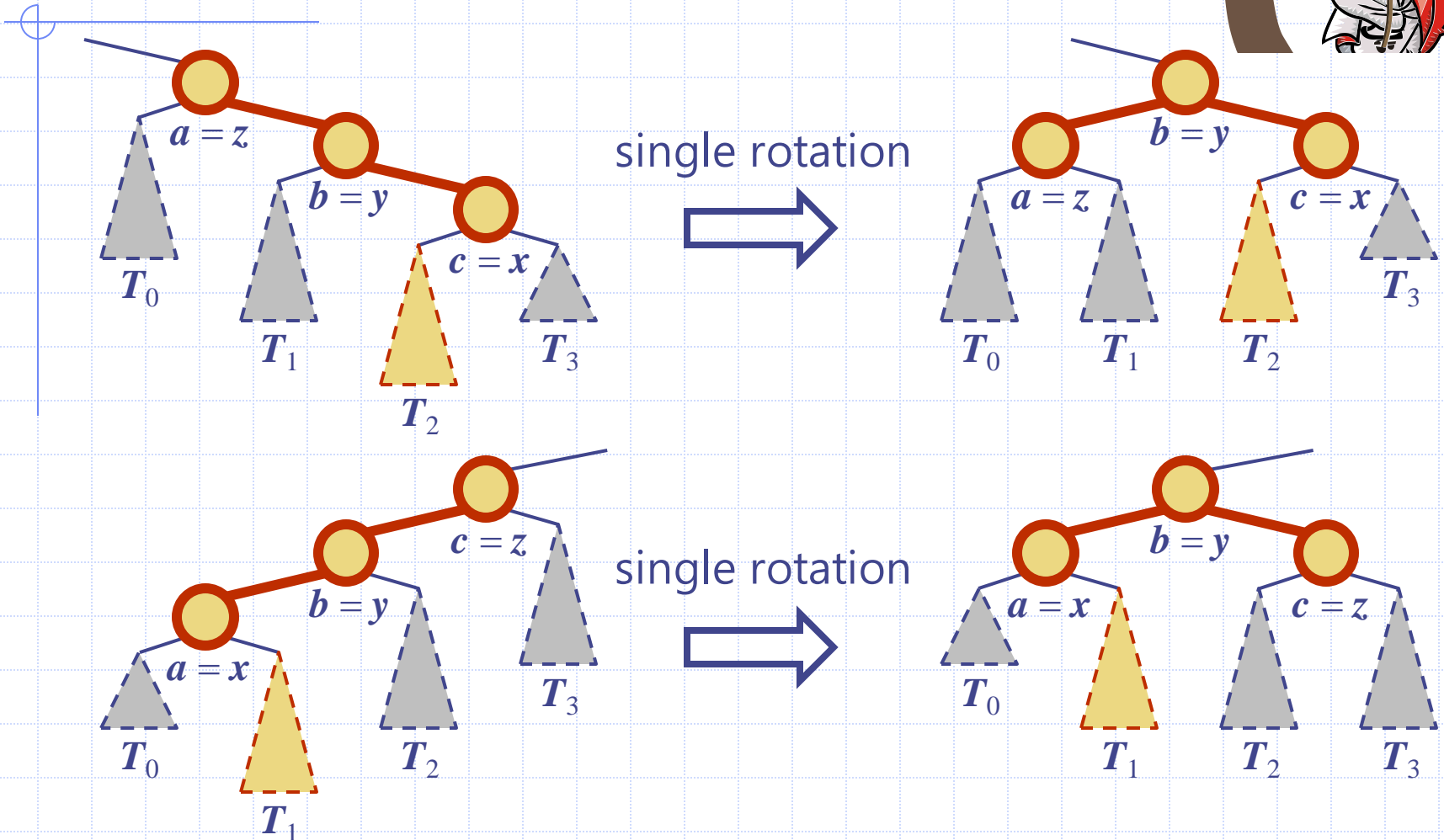
{수행 후, 이제 b 를 루트로 하는 부트리의 모든 노드는 균형을 유지한다. 높이균형 속성은 노드 x, y, z 에서 지역적으로나 전역적으로나 모두 복구된다}

5. **return**

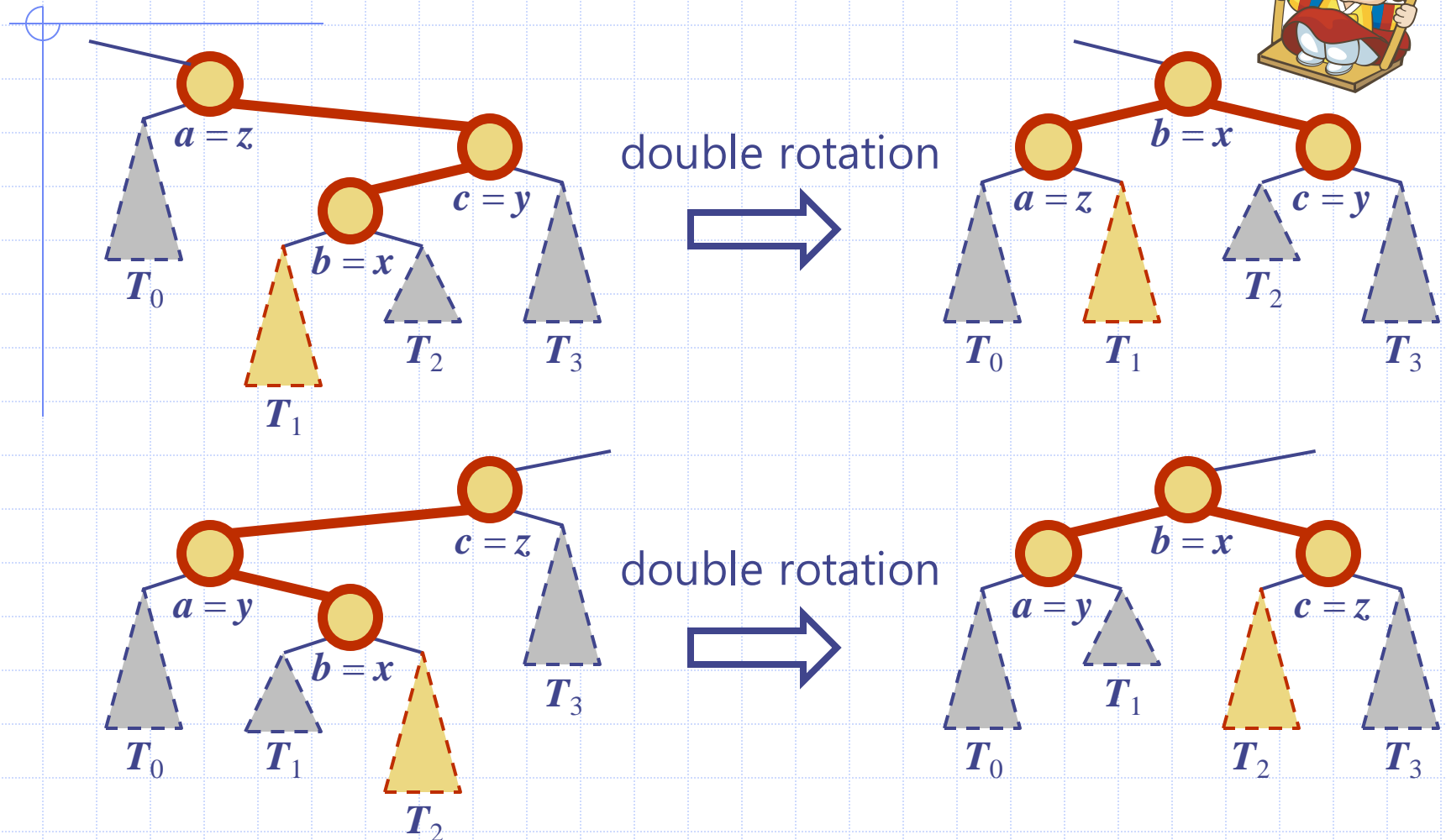
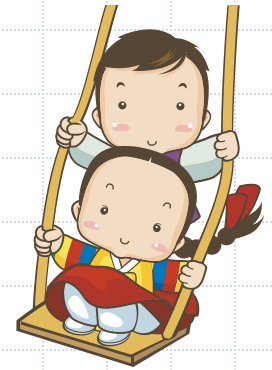
개조

- ◆ 개조는 종종 “회전(rotation)”이라고도 불린다
- ◆ 3-노드 개조(trinode restructure)
 - 3대의 직계 노드 x , y (x 의 부모), z (y 의 부모)의 중위순회 순서 a, b, c 를 회전축으로 하여 수행
- ◆ 단일회전(single rotation)
 - 만약 $b = y$ 면, y 를 중심으로 z 을 회전
- ◆ 이중회전(double rotation)
 - 만약 $b = x$ 면, x 를 중심으로 y 를 회전한 후, 다시 x 를 중심으로 z 을 회전
- ◆ 좌우대칭 포함하여 모두 4종류의 회전 유형 존재

단일회전에 의한 개조



이중회전에 의한 개조



개조를 위한 통합 알고리즘

- ◆ 4가지 유형의 회전 (대칭 모양에 대한 단일 및 이중회전)이 **restructure** 작업에 모두 반영되어 있다
- ◆ T 의 모든 노드의 중위순회 순서는 **보존**
- ◆ T 의 $O(1)$ 개 노드의 부모-자식 관계만 수정



개조

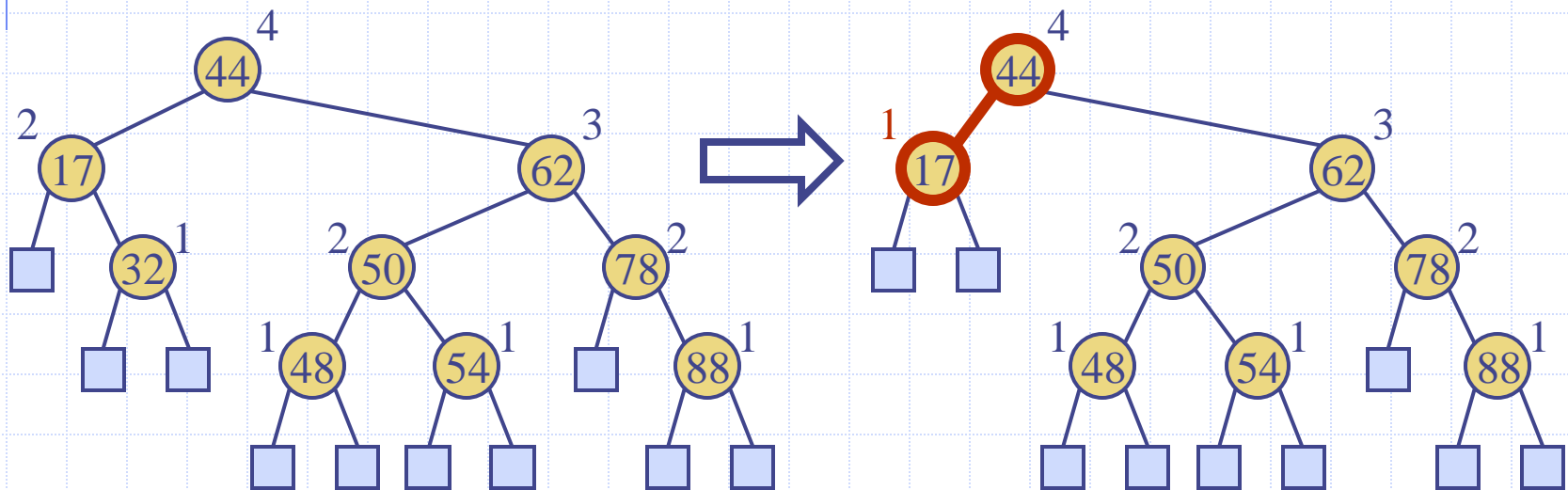
Alg *restructure*(x, y, z)

input a node x of a binary search tree T that has both a parent y and a grandparent z
output tree T after restructuring involving nodes x, y and z

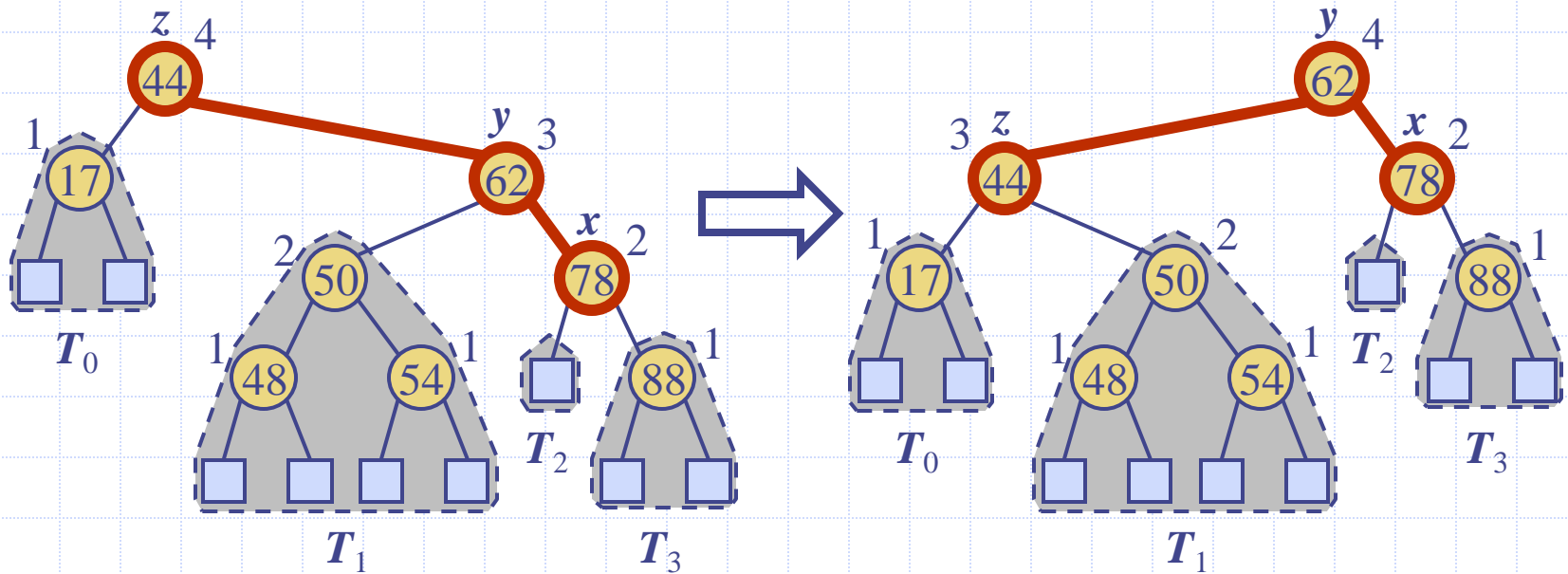
1. x, y, z 의 중위순회 방문 순서의 나열을 (a, b, c) 라 하자.
2. x, y, z 의 부트리들 가운데 x, y, z 를 루트로 하는 부트리를 제외한 4개의 부트리들의 중위순회 방문순서의 나열을 (T_0, T_1, T_2, T_3) 라 하자.
3. z 를 루트로 하는 부트리를 b 를 루트로 하는 부트리로 대체.
4. T_0 와 T_1 을 각각 a 의 왼쪽 및 오른쪽 부트리로 만든다.
5. T_2 와 T_3 를 각각 c 의 왼쪽 및 오른쪽 부트리로 만든다.
6. a 와 c 를 각각 b 의 왼쪽 및 오른쪽 자식으로 만든다.
7. **return** b

AVL 트리에서 삭제

- ◆ 삭제는 이진탐색트리에서와 동일하게 수행
- ◆ **reduceExternal** 작업에 의해 삭제된 노드의 부모노드 w (그리고 조상노드들)가 불균형 상태일 수 있다
- ◆ 예: **removeElement(32)**



삭제 후 개조



삭제

Alg *removeElement(k)*

input AVL tree T , key k

output element with key k

1. $w \leftarrow \text{treeSearch}(\text{root}(), k)$

2. **if** ($\text{isExternal}(w)$)
 return *NoSuchKey*

3. $e \leftarrow \text{element}(w)$

4. $z \leftarrow \text{leftChild}(w)$

5. **if** ($\neg \text{isExternal}(z)$)

$z \leftarrow \text{rightChild}(w)$

6. **if** ($\text{isExternal}(z)$) { case 1 }

$zs \leftarrow \text{reduceExternal}(z)$

else { case 2 }

$y \leftarrow \text{inOrderSucc}(w)$

$z \leftarrow \text{leftChild}(y)$

 Set node w to ($\text{key}(y)$, $\text{element}(y)$)

$zs \leftarrow \text{reduceExternal}(z)$

7. *searchAndFixAfterRemoval(parent(zs))*

8. **return** e

삭제 (conti.)

Alg *searchAndFixAfterRemoval*(w)

input internal node w

output none

1. w 에서 T 의 루트로 향해 올라가다가 처음 만나는 불균형 노드를 z 이라 하자(그러한 z 이 없다면 **return**).
2. z 의 높은 자식을 y 라 하자.
{수행 후, y 는 w 의 조상이 아닌 z 의 자식이 되는 것에 유의}
3. 다음과 같이 하여 y 의 자식 중 하나를 x 라 하자. y 의 두 자식 중 어느 한쪽이 높으면 높은 자식을 x 라 하고, 두 자식의 높이가 같으면 둘 중 y 와 같은 쪽의 자식을 x 로 선택.

4. $b \leftarrow \text{restructure}(x, y, z)$

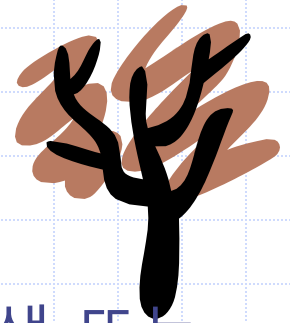
{수행 후, 높이균형 속성은, 방금 전 z 를 루트로 했으나 이젠 변수 b 를 루트로 하는 부트리에서 지역적으로 복구된다. 하지만, 방금의 개조에 의해 b 를 루트로 하는 부트리의 높이가 1 줄어 들 수 있으며 이때문에 b 의 조상이 균형을 잃을 수 있다. 즉, 삭제 후 1회의 개조만으로는 높이균형 속성을 전역적으로 복구하지 못할 수도 있다}

5. T 를 b 의 부모부터 루트까지 올라가면서 균형을 잃은 노드를 찾아 수리하는 것을 계속.

AVL 트리의 성능

- ◆ AVL 트리를 사용하여 구현된 n 개의 항목으로 이루어진 **사전**을 전제하면
 - 공간사용량: $O(n)$
 - 높이: $O(\log n)$
 - 3-노드 개조, 즉 한 번의 **restructure**: $O(1)$
 - ◆ 연결 이진트리 사용을 전제
 - **findElement**: $O(\log n)$
 - ◆ 개조 불필요
 - **insertItem, removeElement**: $O(\log n)$
 - ◆ 초기의 **treeSearch**: $O(\log n)$
 - ◆ 트리를 올라가면서 개조를 수행하여 높이균형을 회복: $O(\log n)$

스플레이 트리

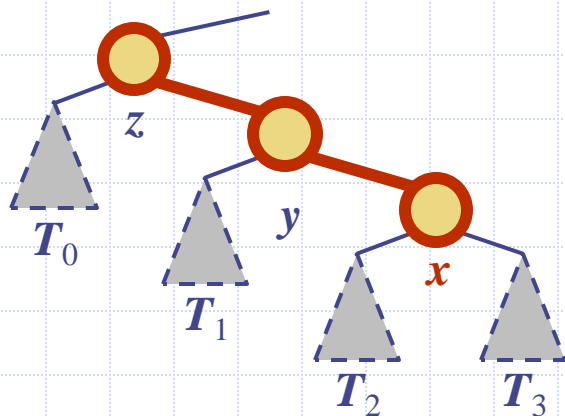
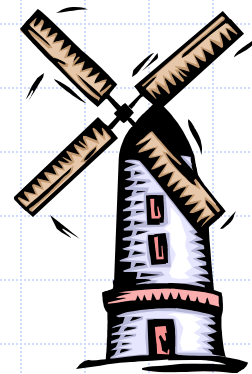


- ◆ **스플레이 트리(splay tree)**: 트리의 노드가 (탐색 또는 갱신을 위해) 접근된 후 스프레이되는 이진탐색트리
 - "노드 x 를 스프레이" = "연속적인 재구성을 통해 노드 x 를 루트로 이동시킴"
 - 가장 깊은 내부노드를 스프레이
- ◆ 일종의 **자기조정(self-adjusting)** 이진탐색트리
- ◆ **이점**
 - 비교적 단순: 높이균형을 유지하지 않으므로
 - 탐색, 삽입, 삭제: $O(\log n)$ **상각실행시간**
 - 자기조정성: 각 항목에 대한 접근빈도가 균등하지 않은 사전에 사용하면 유리

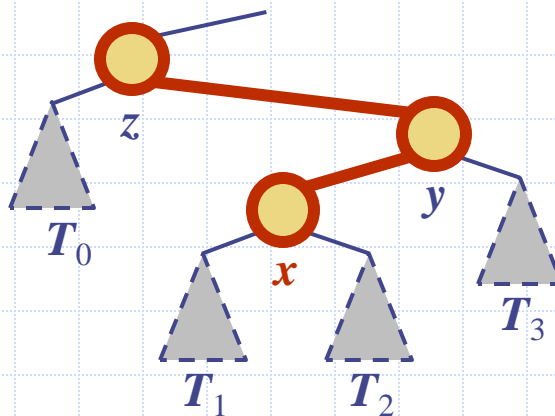
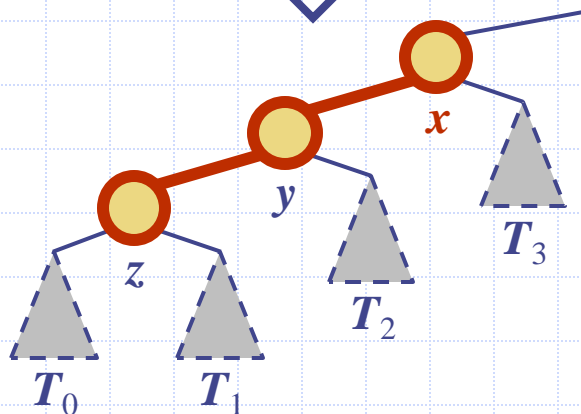
언제 무엇을 스플레이?

언제	어떤 노드를
탐색 시에	<ul style="list-style-type: none">■ 키가 어떤 내부노드에서 발견되면, 그 노드를 스플레이■ 그렇지 않으면 탐색이 실패한 외부노드의 부모노드를 스플레이
삽입 시에	<ul style="list-style-type: none">■ 새로 삽입한 내부노드를 스플레이
삭제 시에	<ul style="list-style-type: none">■ 실제로 삭제된 내부노드의 부모노드를 스플레이

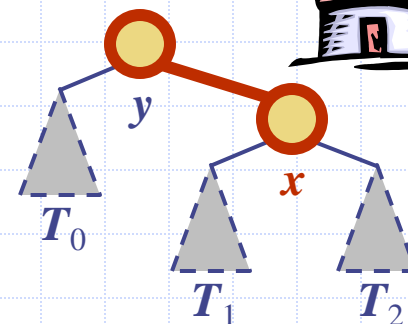
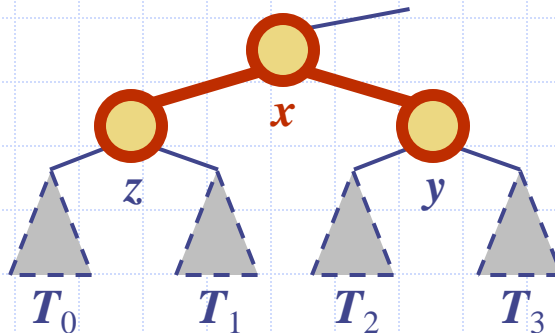
스플레이의 세 가지 경우



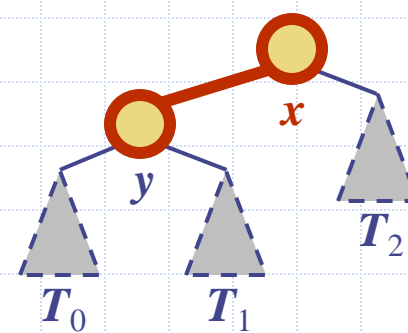
zig-zig 회전



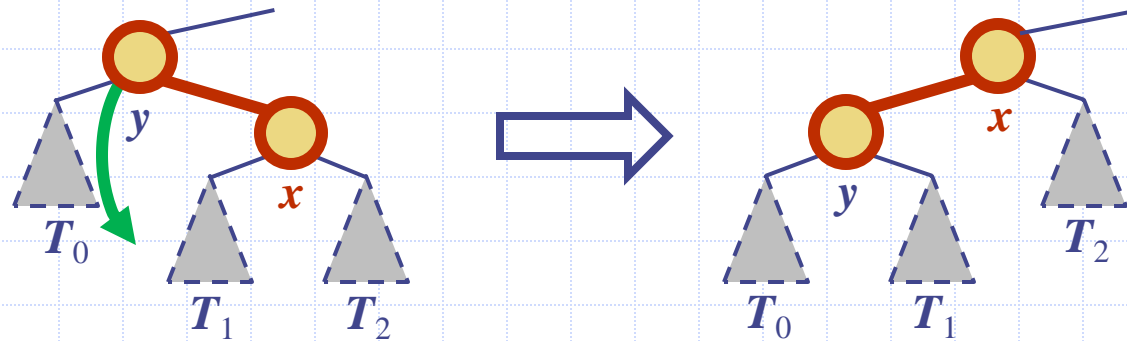
zig-zag 회전



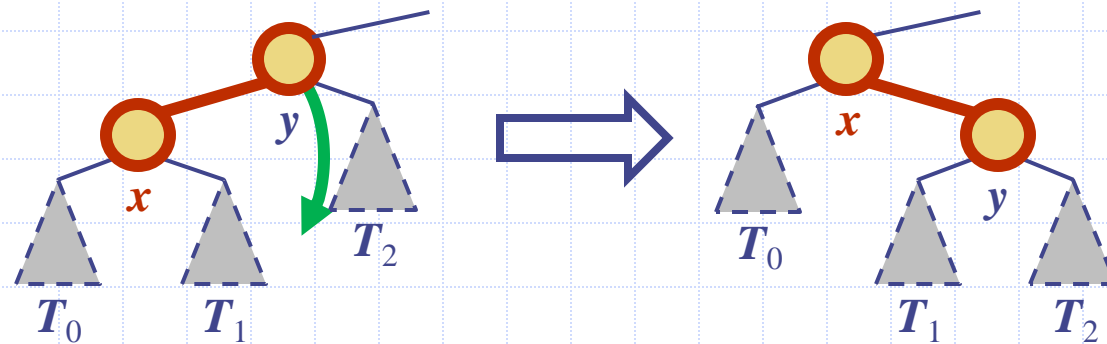
zig 회전



두 가지 기본 회전



(a) leftRotate(x , y)



(a) rightRotate(x , y)

스플레이

Alg *splay*(*x*)

input internal node *x*

output none

1. **if** (*isRoot*(*x*))

return

2. **if** (*isRoot*(*parent*(*x*))) { zig }

if (*x* = *leftChild*(*root*()))

rightRotate(*x*, *root*())

else

leftRotate(*x*, *root*())

return

3. *p* ← *parent*(*x*)

4. *g* ← *parent*(*p*)

5. **if** (*x* = *leftChild*(*leftChild*(*g*))) { zig-zig }

rightRotate(*p*, *g*)

rightRotate(*x*, *p*)

elseif (*x* = *rightChild*(*rightChild*(*g*)))
 { zig-zig }

leftRotate(*p*, *g*)

leftRotate(*x*, *p*)

elseif (*x* = *leftChild*(*rightChild*(*g*)))
 { zig-zag }

rightRotate(*x*, *p*)

leftRotate(*x*, *g*)

else { *x* = *rightChild*(*leftChild*(*g*)) }
 { zig-zag }

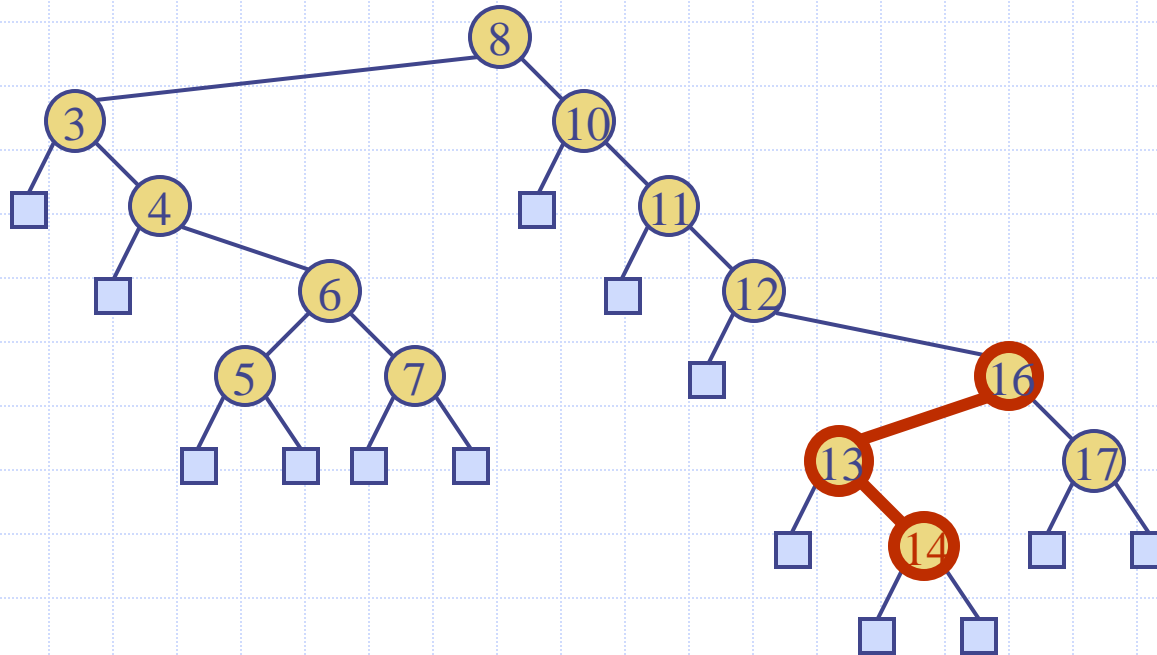
leftRotate(*x*, *p*)

rightRotate(*x*, *g*)

6. *splay*(*x*)

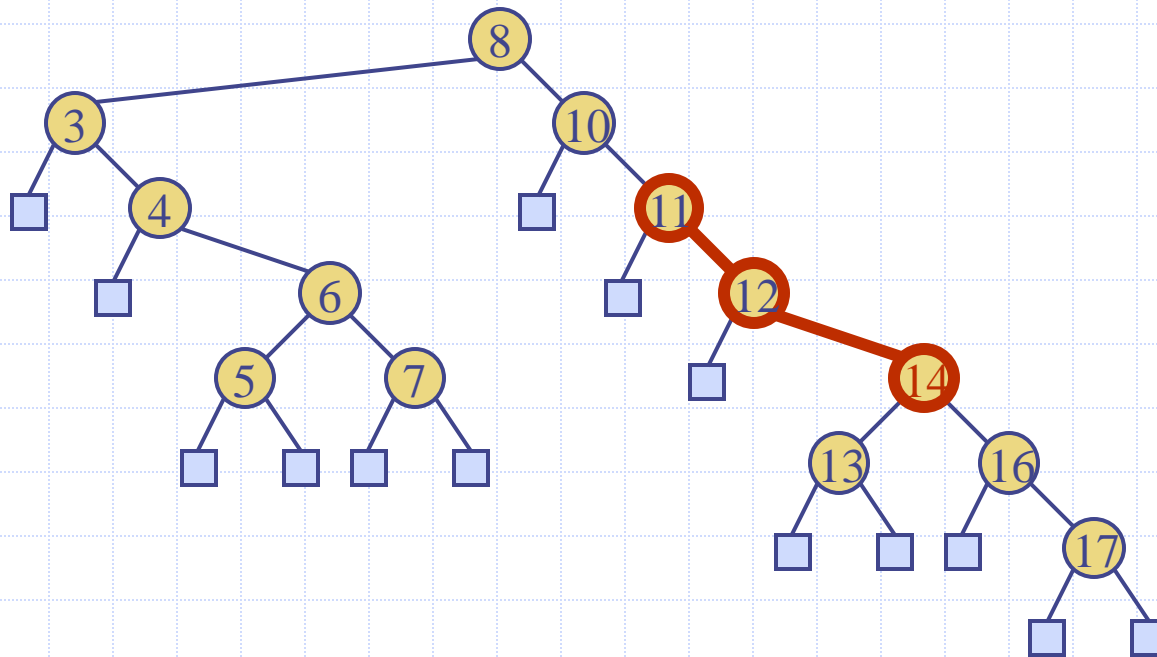
스플레이 예

- ◆ 주어진 스플레이 트리는 아래 셋 가운데 한 작업이 처리된 직후의 모습으로 전제
 - 키 14에 대한 성공적인 탐색, 또는 키 15에 대한 실패한 탐색
 - 키 14를 삽입
 - 키 14를 저장한 노드의 자식노드를 삭제
- ◆ 키 14를 저장한 노드에 대한 스플레이는 zig-zag으로 출발



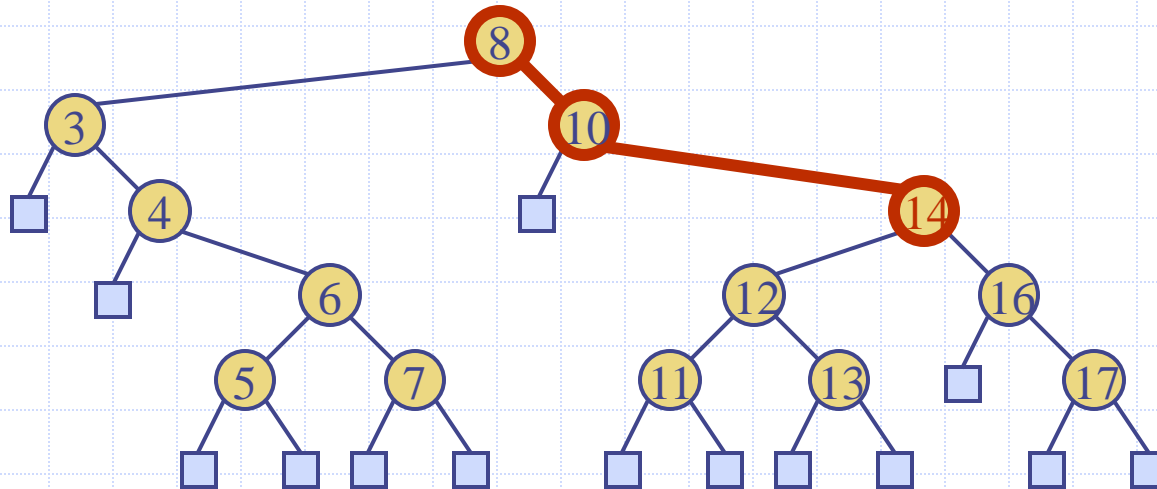
스플레이 예 (conti.)

◆ zig-zag 회전 다음은 zig-zig



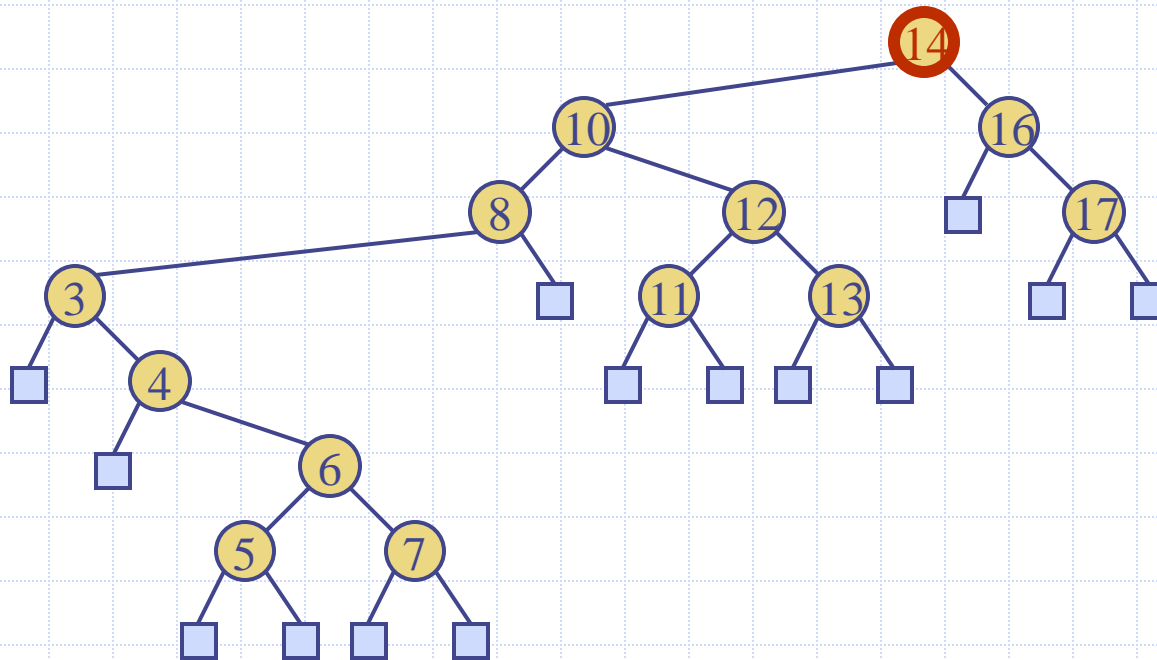
스플레이 예 (conti.)

◆ zig-zig 회전 다음은 또다시 zig-zig



스플레이 예 (conti.)

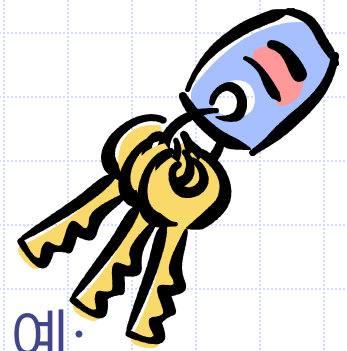
◆ zig-zig 회전으로 스프레이 완료



스플레이 트리의 성능

- ◆ 스플레이 실행시간: $O(h)$
 - 여기서 h 는 트리의 높이이며, 최악의 경우 $O(n)$
- ◆ 스플레이 트리 유용성
 - AVL 트리보다 구현이 간단
 - ◆ AVL 트리보다 경우의 수가 작다
 - ◆ AVL 트리와는 달리 각 노드에 높이나 균형 정보 유지 불필요
 - 탐색, 삽입, 삭제: $O(\log n)$ 상각실행시간

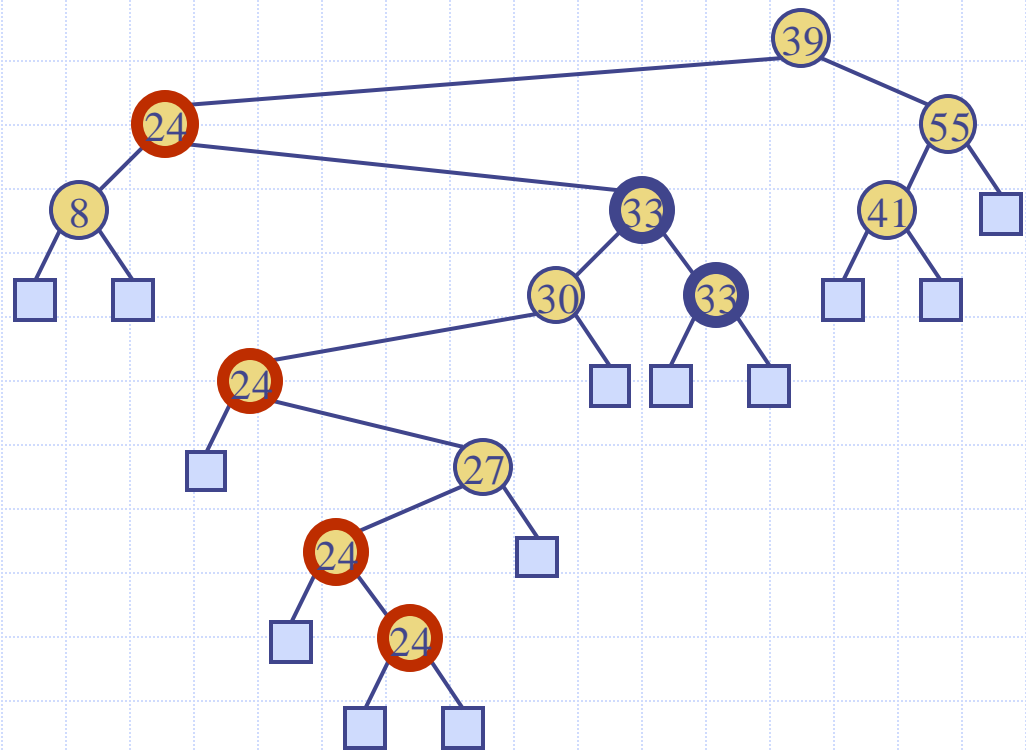
응용문제: 중복 키를 가진 이진탐색트리 메소드



◆ 중복 키가 존재
가능한
이진탐색트리다
- 즉,

- u, v, w 세 개의
노드에 대해, u 와
 w 가 각각 v 의
왼쪽 및 오른쪽
부트리 내의
노드일 때
다음이 성립
 $key(u) < key(v) \leq$
 $key(w)$

◆ 이진탐색트리 예:



응용문제: 중복 키를 가진 이진탐색트리 메소드 (conti.)

- A. 이진탐색트리 T 를 사용하여 구현된 순서 사전에서 주어진 키 k 를 갖는 모든 원소들을 반환하는 `findAllElements(k)` 작업을 수행할 알고리즘을 작성하라
- B. 이진탐색트리 T 를 사용하여 구현된 순서 사전에서 `insertItem(k, e)` 작업을 수행할 알고리즘을 작성하라
- C. 이진탐색트리 T 를 사용하여 구현된 순서 사전에서 주어진 키 k 를 갖는 모든 항목을 삭제하고 해당 원소들을 반환하는 `removeAllElements(k)` 작업을 수행할 알고리즘을 작성하라

◆ 주의

- 모든 알고리즘은 $O(h + s)$ 시간에 수행하여야 한다 – 여기서 h 는 T 의 높이이며 s 는 반환되는 원소의 수
- 원소들이 반환되는 순서는 중요하지 않다
- `treeSearch` 사용 가능

해결

Alg *findAllElements(k)*

input binary search tree T , key k

output elements with key k

1. $L \leftarrow \text{empty list}$
 2. $w \leftarrow \text{treeSearch}(\text{root}(), k)$
 3. **while** (*isInternal*(w))
 $L.\text{addLast}(\text{element}(w))$
 $w \leftarrow \text{treeSearch}(\text{rightChild}(w), k)$
 4. **return** $L.\text{elements}()$
- { Total $O(h + s)$ }

Alg *insertItem(k, e)*

input binary search tree T , key k ,
element e

output none

1. $w \leftarrow \text{treeSearch}(\text{root}(), k)$
 2. **while** (*isInternal*(w))
 $w \leftarrow$
 $\text{treeSearch}(\text{rightChild}(w), k)$
 3. Set node w to (k, e)
 4. *expandExternal*(w)
 5. **return**
- { Total $O(h + s)$ }

해결 (conti.)

Alg *removeAllElements(k)*

input binary search tree T , key k

output elements with key k

1. $L \leftarrow \text{empty list}$

2. $w \leftarrow \text{treeSearch}(\text{root}(), k)$

3. **while** ($\text{isInternal}(w)$)

$L.\text{addLast}(\text{element}(w))$

$z \leftarrow \text{leftChild}(w)$

if ($\neg \text{isExternal}(z)$)

$z \leftarrow \text{rightChild}(w)$

if ($\text{isExternal}(z)$)

{ case 1 }

$w \leftarrow \text{reduceExternal}(z)$

else

{ case 2 }

$y \leftarrow \text{inOrderSucc}(w)$

$x \leftarrow \text{leftChild}(y)$

Set node w to ($\text{key}(y)$, $\text{element}(y)$)

$\text{reduceExternal}(x)$

$w \leftarrow \text{treeSearch}(w, k)$

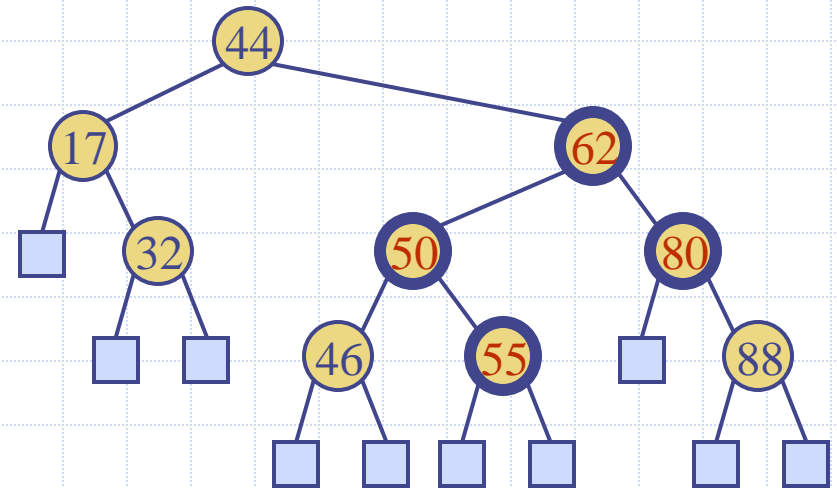
4. **return** $L.\text{elements}()$

{ Total $O(h + s)$ }

응용문제: 주어진 키 범위 내의 원소들



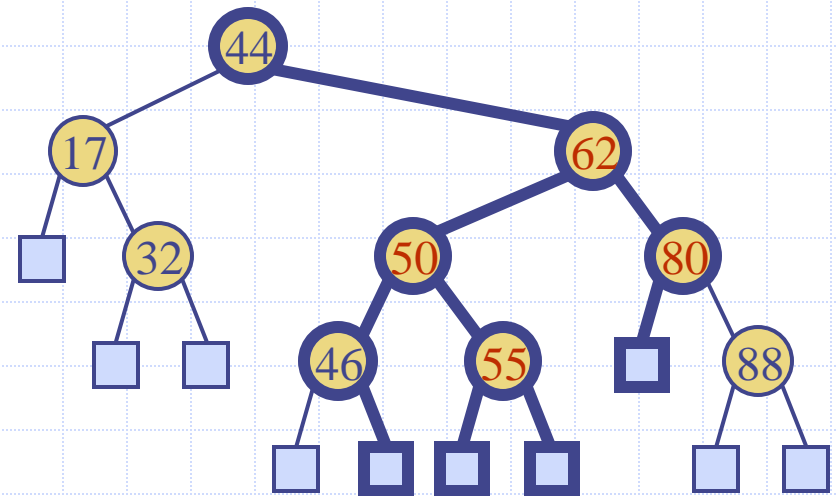
- ◆ AVL 트리 T 를 사용하여 구현된 유일 키로 이루어진 n 항목의 순서사전이 있다
- ◆ T 에서 $O(\log n + s)$ 시간에 수행하는 다음 메소드를 구현하라 - 여기서 s 는 반환되는 원소의 수
 - **findAllInRange**(k_1, k_2): $k_1 \leq k \leq k_2$ 인 키 k 를 가진 T 의 모든 원소를 반환
- ◆ 원소들이 반환되는 순서는 중요하지 않다
- ◆ 예: 오른쪽에 보인 AVL 트리 예에서, **findAllInRange**(48, 80)는 62, 50, 55, 80 키를 가진 원소들을 반환





해결

- ◆ AVL 트리 T 의 각 내부노드 v 에서 $k_1 \leq \text{key}(v) \leq k_2$ 면 노드 v 의 원소 $\text{element}(v)$ 를 수집한 후, $\text{key}(v)$ 의 크기에 따라 다음 세 가지 경우로 나누어 좌우 부트리에 대한 탐색을 계속
 - $\text{key}(v) \leq k_1$: 노드 v 의 오른쪽 부트리에 대해 탐색을 계속
 - $k_2 \leq \text{key}(v)$: 노드 v 의 왼쪽 부트리에 대해 탐색을 계속
 - $k_1 < \text{key}(v) < k_2$: 노드 v 의 좌우 부트리 모두에 대해 각각 탐색을 계속
- ◆ v 가 외부노드인 경우 반환
- ◆ 원소들은 비어 있는 리스트 L 을 초기화하여 수집
- ◆ 루트로부터 외부노드로 향하는 경로를 순회하며 s 개의 노드를 방문하므로 $O(\log n + s)$ 시간에 수행
- ◆ 예: 주어진 AVL 트리 예에서, **findAllInRange**(48, 80)이 방문하는 간선과 노드들을 굵은 선으로 표시



해결

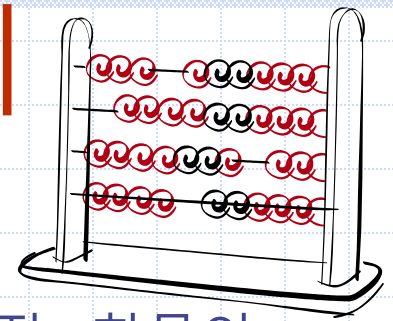
Alg *findAllInRange*(k_1, k_2)
 {distinct keys}
input AVL tree T , key k_1, k_2
output all elements with key k s.t.
 $k_1 \leq k \leq k_2$

1. $L \leftarrow \text{empty list}$
2. $rFAIR(\text{root}(), k_1, k_2)$
3. **return** $L.\text{elements}()$
 {Total $O(\log n + s)$ }

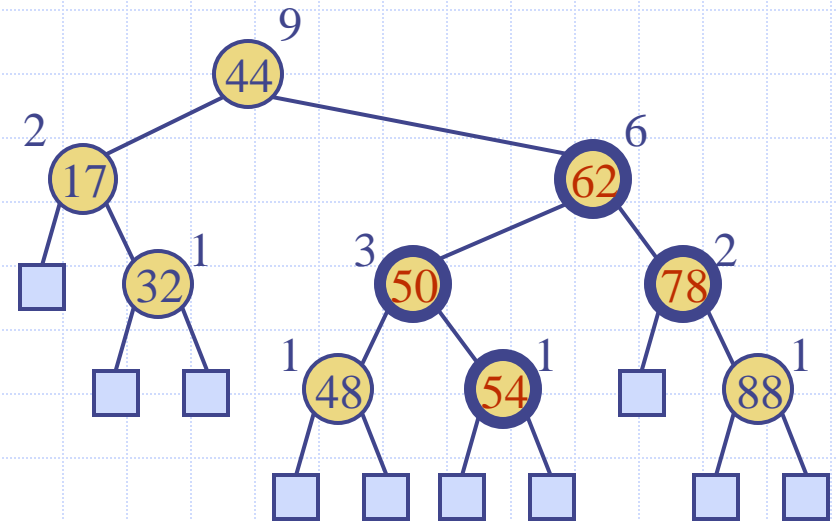
Alg *rFAIR*(v, k_1, k_2)
input list L , node v of an AVL tree T ,
 key k_1, k_2
output list L of elements with key k
 s.t. $k_1 \leq k \leq k_2$

1. **if** ($\text{isExternal}(v)$)
 return
2. **if** ($k_1 \leq \text{key}(v) \leq k_2$)
 $L.\text{addLast}(\text{element}(v))$
3. **if** ($\text{key}(v) \leq k_1$)
 $rFAIR(\text{rightChild}(v), k_1, k_2)$
 elseif ($k_2 \leq \text{key}(v)$)
 $rFAIR(\text{leftChild}(v), k_1, k_2)$
 else $\{k_1 < \text{key}(v) < k_2\}$
 $rFAIR(\text{leftChild}(v), k_1, k_2)$
 $rFAIR(\text{rightChild}(v), k_1, k_2)$

응용문제: 주어진 키 범위 내의 원소 수

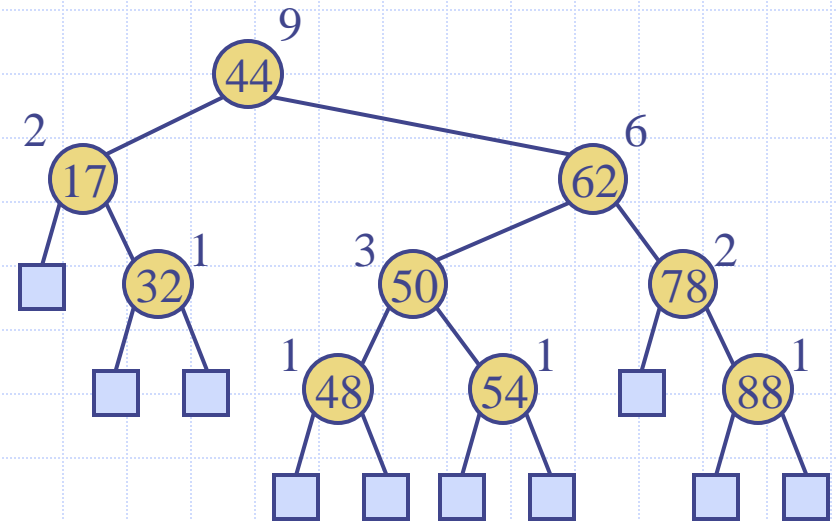


- ◆ AVL 트리 T 를 사용하여 구현된, 유일 키로 이루어진 n 항목의 순서사전이 있다
- ◆ T 에서 $O(\log n)$ 시간에 수행하는 다음 메소드를 구현하라
 - **countAllInRange**(k_1, k_2): AVL 트리 T 의 $k_1 \leq k \leq k_2$ 인 키 k 들의 수를 계산하여 반환
- ◆ **힌트:** AVL 트리의 데이터구조를 확장하여 각 내부노드에 새 라벨을 정의하고 트리가 갱신되면 이 라벨의 값도 갱신
- ◆ **예:** 오른쪽에 보인 AVL 트리 예에서, **countAllInRange**(50, 80)는 4를 반환



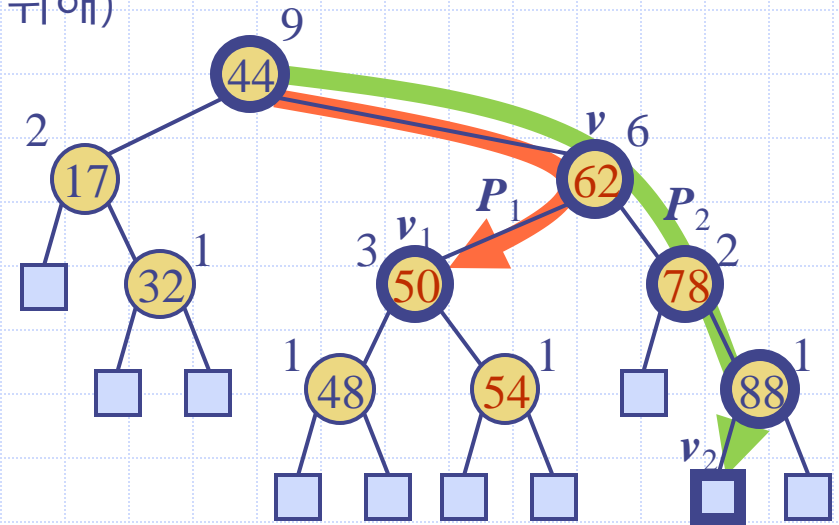
해결

- ◆ AVL 트리의 각 노드에 그 노드를 루트로 하는 부트리의 크기, 즉 부트리 내의 **내부노드** 수를 저장
- ◆ 트리 **갱신** 시에는 삽입이나 삭제가 수행된 경로의 노드들의 부트리의 크기를 증가시키거나 감소시킨다
- ◆ 특히, **3-노드 개조**를 수행할 때 세 노드(즉, 개조 알고리즘의 a , b , c 노드)를 루트로 하는 부트리의 크기를 올바르게 갱신
- ◆ 이런 방식으로 AVL 트리 T 의 데이터구조가 확장되어 T 의 각 노드 v 에 대해 v 를 루트로 하는 부트리의 크기를 반환하는 **size(v)** 메소드가 있다고 전제
- ◆ 예: 이 방식으로 구현된 AVL 트리 - 각 노드 v 의 정수 라벨은 **size(v)** 값



해결 (conti.)

- ◆ (k_1, k_2) 키 쌍 범위 내 노드들의 수를 계산하기 위해 k_1 과 k_2 모두를 탐색하여 각각의 탐색경로를 P_1 과 P_2 라 하고 각각의 탐색이 반환한 노드를 v_1 과 v_2 라 한다
- ◆ 두 경로에 공통된 마지막 노드를 v 라 한다
- ◆ P_1 경로를 v 로부터 v_1 까지 순회
- ◆ 순회 도중 만나는 각 내부노드 $w \neq v$ 에 대해, w 의 **오른쪽** 자식이 P_1 에 존재하지 않는다면, 현재까지의 합에 그 오른쪽 자식의 부트리의 크기와 1을 더한다(w 를 포함하기 위해)
- ◆ 마찬가지로, P_2 경로를 v 로부터 v_2 까지 순회
- ◆ 순회 도중 만나는 각 내부노드 $w \neq v$ 에 대해, w 의 **왼쪽** 자식이 P_2 에 존재하지 않는다면, 현재까지의 합에 그 왼쪽 자식의 부트리의 크기와 1을 더한다
- ◆ 마지막으로 v 가 내부노드면 현재까지의 합에 1을 더한다



해결 (conti.)

- ◆ 알고리즘 `treeSearch`는 방문한 경로의 노드들을 반환하도록 수정한 버전을 사용
- ◆ 분석
 - 두 키에 대한 탐색은 각각 $O(\log n)$ 시간 소요
 - 두 개의 탐색경로에 공통된 마지막 노드 v 를 찾는데 $O(\log n)$ 시간 소요
 - 두 개의 탐색경로 각각에서 v 로부터의 부경로를 추출하는데 $O(\log n)$ 시간 소요
 - 두 개의 탐색경로를 순회하는데 각각 $O(\log n)$ 시간 소요
 - 그러므로 전체적으로 $O(\log n)$ 시간에 수행

해결 (conti.)

Alg *countAllInRange*(k_1, k_2)

input AVL tree T , key k_1, k_2

output the number of items with key k , s.t. $k_1 \leq k \leq k_2$

1. $P_1, P_2 \leftarrow$ empty list

2. $v_1 \leftarrow \text{treeSearch}(P_1, \text{root}(), k_1)$
 $\{\mathbf{O}(\log n)\}$

3. $v_2 \leftarrow \text{treeSearch}(P_2, \text{root}(), k_2)$
 $\{\mathbf{O}(\log n)\}$

4. $v \leftarrow$ last node common to P_1 and P_2
 $\{\mathbf{O}(\log n)\}$

5. $S_1 \leftarrow$ subpath of P_1 starting at v

6. $S_2 \leftarrow$ subpath of P_2 starting at v

7. $c \leftarrow 0$

8. **for** each $w \neq v \in S_1.\text{elements}()$ $\{\mathbf{O}(\log n)\}$

if (*isInternal*(w))

$u \leftarrow \text{rightChild}(w)$

if ($u \notin S_1$)

if (*isInternal*(u))

$c \leftarrow c + \text{size}(u)$

$c \leftarrow c + 1$

$\{\text{count } w\}$

9. **for** each $w \neq v \in S_2.\text{elements}()$ $\{\mathbf{O}(\log n)\}$

if (*isInternal*(w))

$u \leftarrow \text{leftChild}(w)$

if ($u \notin S_2$)

if (*isInternal*(u))

$c \leftarrow c + \text{size}(u)$

$c \leftarrow c + 1$

$\{\text{count } w\}$

10. **if** (*isInternal*(v))

$c \leftarrow c + 1$

$\{\text{count } v\}$

11. **return** c

$\{\text{Total } \mathbf{O}(\log n)\}$

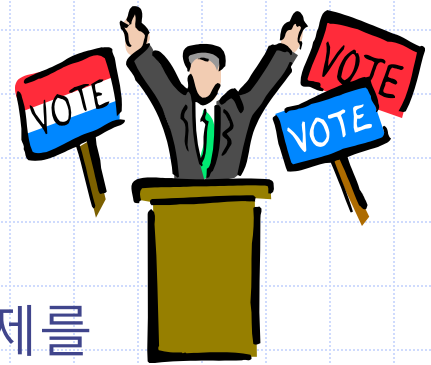
해결 (conti.)

Alg *treeSearch*(L, v, k) { another version }

input list L , node v of a binary search tree,
key k

output node w , s.t. either w is an internal
node storing key k or w is the external
node where key k would belong if it
existed, list L of nodes visited

1. $L.addLast(v)$
2. **if** ($isExternal(v)$)
 return v
3. **if** ($k = key(v)$)
 return v
 elseif ($k < key(v)$)
 return $treeSearch(L, leftChild(v), k)$
 else { $k > key(v)$ }
 return $treeSearch(L, rightChild(v), k)$
 { Total $O(\log n)$ }



응용문제: 투표

- ◆ 정렬 일반 장의 응용문제에서 다루었던 투표 문제를 다시 생각해 보자
- ◆ n -원소 리스트 L 이 주어졌다고 가정하자 – 여기서 L 의 각 원소는 선거에서의 투표를 표현
- ◆ 각 투표는 선택된 후보자의 ID를 나타내는 정수로 주어진다
 - 기호들은 정수지만 빠진 번호가 있을 수도 있다
- ◆ 이번엔 출마한 후보자의 수 $k < n$ 를 안다고 가정
- ◆ 당선자를 찾아내는 $O(n \log k)$ -시간 메소드를 작성하라
- ◆ 전제: 가장 많은 표를 획득한 후보자가 당선된다
- ◆ 예: 아래 투표 리스트에서 기호 7이 당선자다



해결 (Ver. 1)

◆ 메소드 1 (분할을 이용)

1. **inPlacePartition**의 중복 키가 존재하는 경우의 버전을 사용하여 리스트 L 을 분할
2. LT 와 GT 부리스트에 대하여 분할을 반복
3. 분할이 완료된 후 리스트를 스캔하면서 최대 득표자를 찾는다

◆ 실행시간

- 1단계: $O(n)$
- 2단계: $O(\log k)$ 회의 반복 소요
- 3단계: $O(n)$
- 그러므로 총 $O(n \log k)$ 시간

해결 (Ver. 2, 추천)

◆ 메소드 2 (균형탐색트리를 이용)

1. 후보자의 ID를 AVL 트리와 같은 **균형탐색트리**에 저장 – 이 트리에서, 각 ID와 함께 해당 ID의 득표수를 저장
2. 초기에는 득표수를 모두 0으로 설정
3. 그 다음엔 투표 리스트를 순회하며, 각 투표의 ID에 해당하는 득표수를 증가시킨다

◆ 이 데이터구조는 k 개의 원소를 저장하므로, 각 투표에 대한 탐색과 갱신은 $O(\log k)$ 시간에 수행

◆ 그러므로 총 실행시간은 $O(n \log k)$