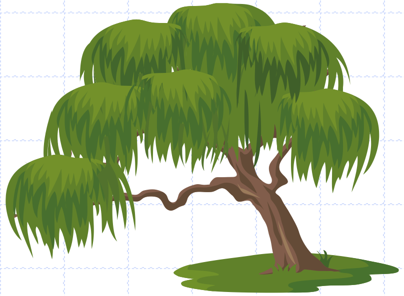
탐색트리

Outline

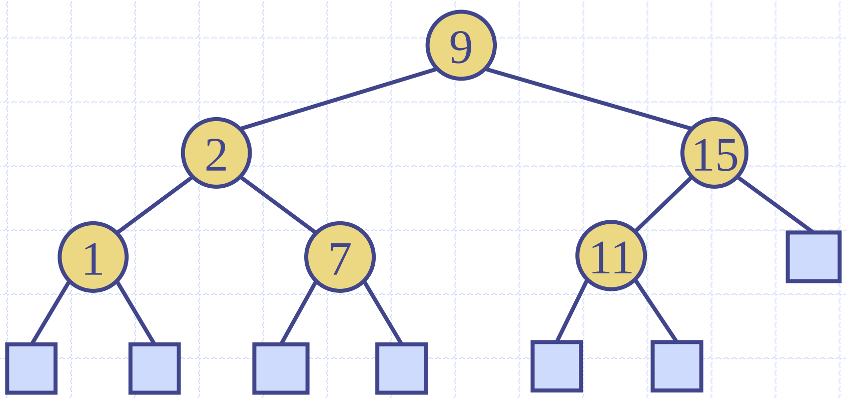
- 11.1 이진탐색트리
- 11.2 AVL 트리
- 11.3 스플레이 트리
- 11.4 응용문제

이진탐색트리



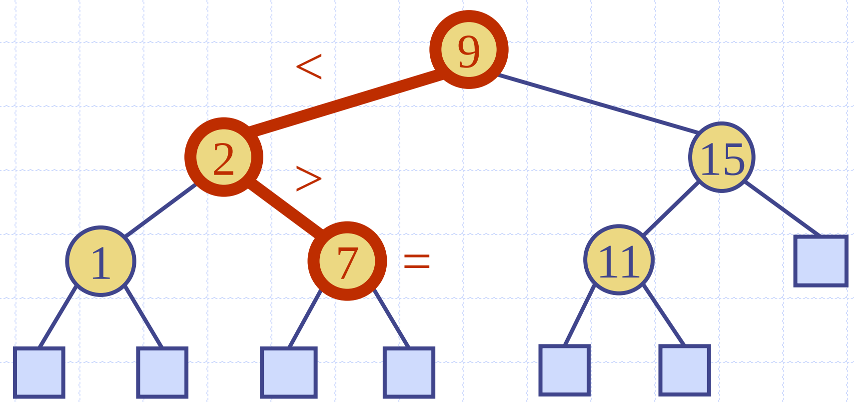
- **이진탐색트리**(binary search tree): 내부노드에 (**키, 원소**) **쌍**을 저장하며 다음의 성질을 만족하는 **이진트리**
  - u, v, w는 모두 트리노드며 u와 w가 각각 v의 왼쪽과 오른쪽 부트리에 존재할 때 다음이 성립
    $key(u) < key(v) \le key(w)$
- **전제: 적정이진트리**로 구현
- **그림 표기:** 내부노드 내에 간단히 키만 표시

- 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***

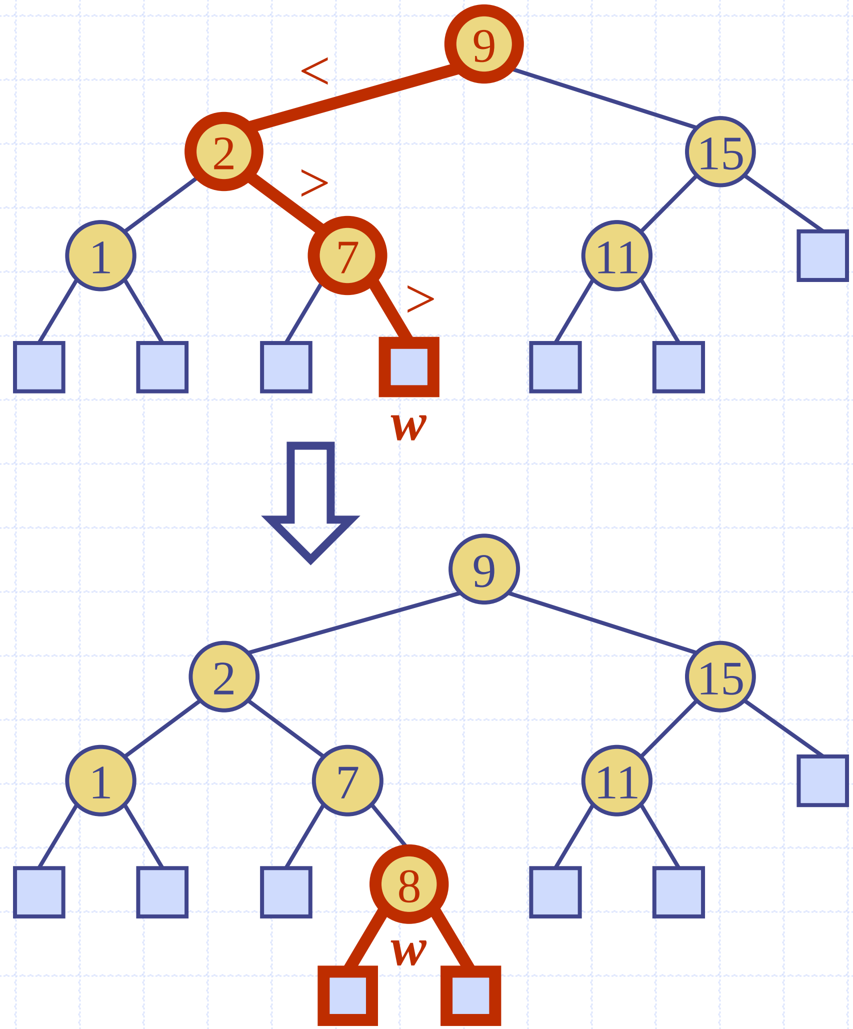
1. ***w*** ← ***treeSearch***(***root***(), ***k***)
2. **if** (***isExternal***(***w***))
  return ***NoSuchKey***
 else
  return ***element***(***w***)

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

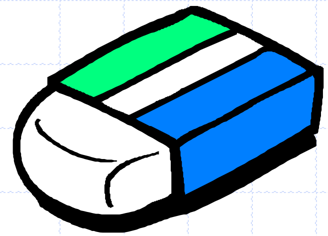


삽입 (conti.)

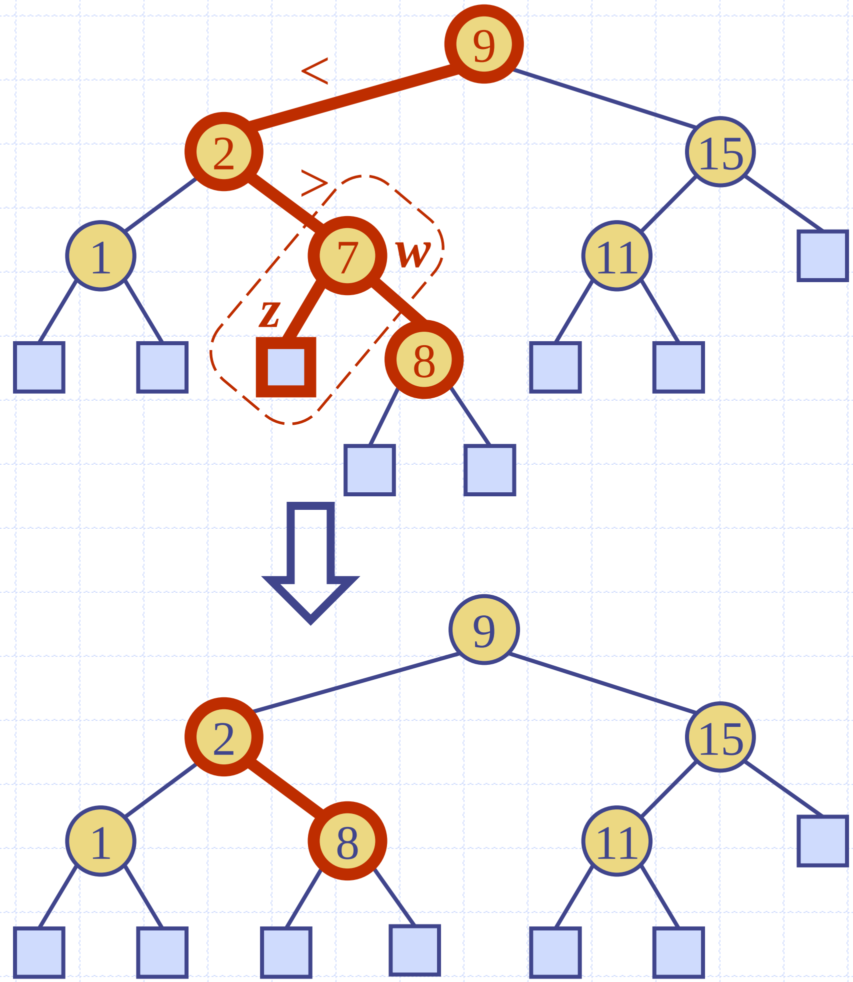
```
Alg insertItem(k, e)
    input binary search tree T, key k,
        element e
    output none

1. w ← treeSearch(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)



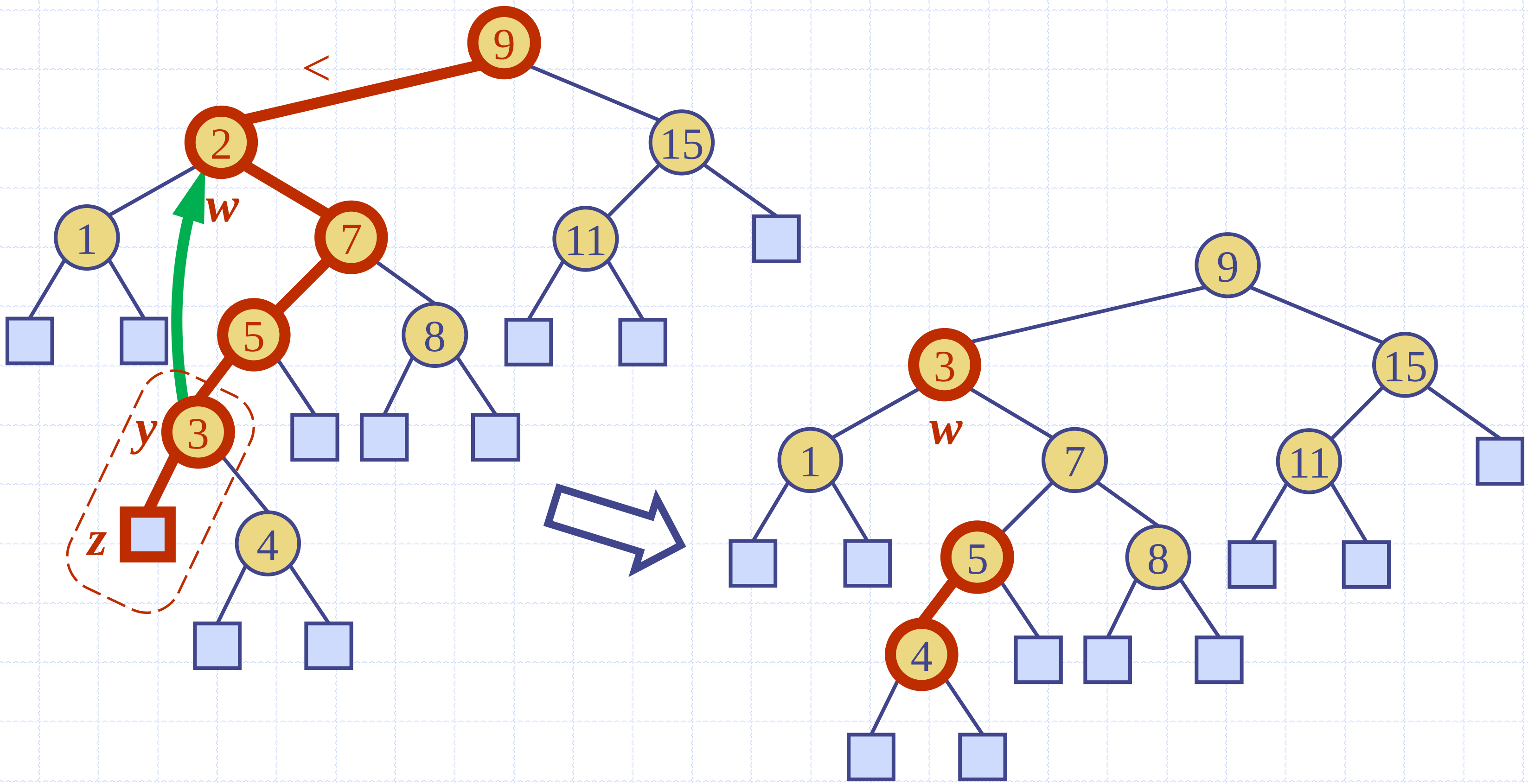
삭제: Case 2

- 삭제되어야 할 키 k가 내부노드만을 자식들로 가지는 노드(w라 하자)에 저장되어 있다면 다음과 같이 처리

1. 트리 T에 대해 w의 **중위순회 계승자** y와 그 자식노드 z을 찾아낸다
   - 노드 y는 우선 w의 오른쪽 자식으로 이동한 후, 거기서부터 왼쪽 자식들만을 끝까지 따라 내려가면 도달하게 되는 마지막 내부노드며, 노드 z은 y의 왼쪽 자식인 외부노드
   - y는 T를 중위순회할 경우 노드 w 바로 다음에 방문하게 되는 내부노드이므로 w의 **중위순회 계승자**(inorder successor)라 불린다
   - 따라서 y는 w의 오른쪽 부트리 내 노드 중 **가장 왼쪽으로 돌출된 내부노드**
2. y의 내용을 w에 복사
3. reduceExternal(z) 작업을 사용하여 노드 y와 z를 삭제

삭제: Case 2 (conti.)

- **예**: removeElement(2)



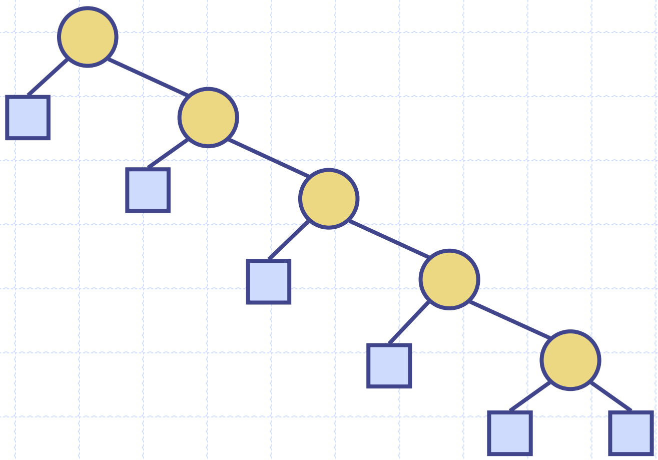
삭제 (conti.)

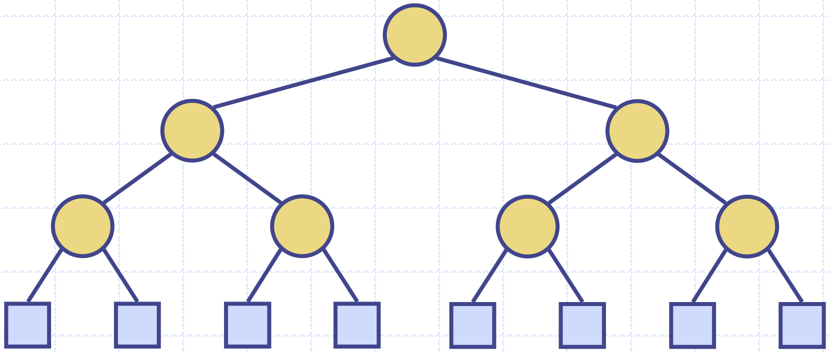
Alg ***removeElement***(***k***)
  input binary search tree ***T***,
    key ***k***
  output element with key ***k***

1. ***w*** ← ***treeSearch***(***root***(), ***k***)
2. **if** (***isExternal***(***w***))
    return ***NoSuchKey***
3. ***e*** ← ***element***(***w***)
4. ***z*** ← ***leftChild***(***w***)
5. **if** (!***isExternal***(***z***))
    z ← ***rightChild***(***w***)
6. **if** (***isExternal***(***z***)) {case 1}
    reduceExternal(***z***)
  else {case 2}
    y ← ***inOrderSucc***(***w***)
    z ← ***leftChild***(***y***)
    *Set node **w** to* (***key***(***y***), ***element***(***y***))
    reduceExternal(***z***)
7. **return** ***e***

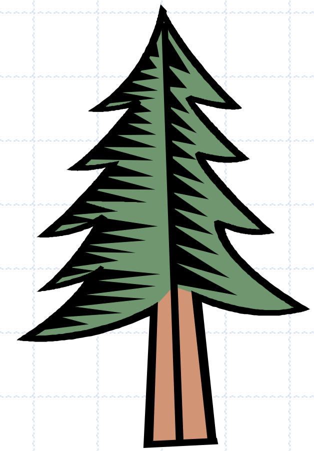
이진탐색트리의 성능

- 높이 h의 이진탐색트리를 사용하여 구현된 n항목의 사전을 가정하면:
  - $O(n)$ 공간 사용
  - findElement, insertItem, removeElement 작업 모두 $O(h)$ 시간에 수행
  - 높이 h는:
    - 최악의 경우 $O(n)$
    - 최선의 경우 $O(\log n)$



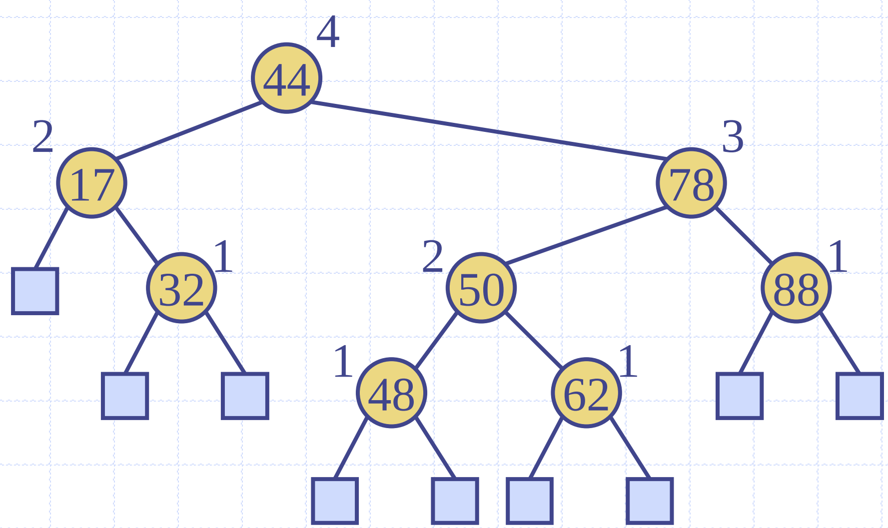


AVL 트리

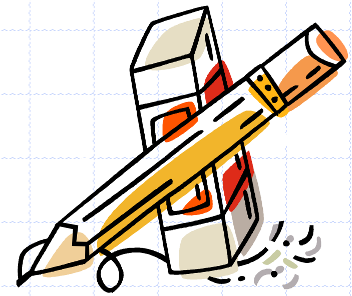


- **AVL 트리:** 트리 T의 모든 내부노드 v에 대해 v의 자식들의 좌우 높이 차이가 1을 넘지 않는 이진탐색트리 (즉, **높이균형 속성**, height-balance property)
  - AVL 트리의 부트리 역시 AVL 트리
  - 높이 (또는 균형) 정보는 각 내부노드에 저장
  - n개의 항목을 저장하는 AVL 트리의 높이: $O(\log n)$
  - findElement 작업: $O(\log n)$ 시간 소요

- AVL 트리 **예:**



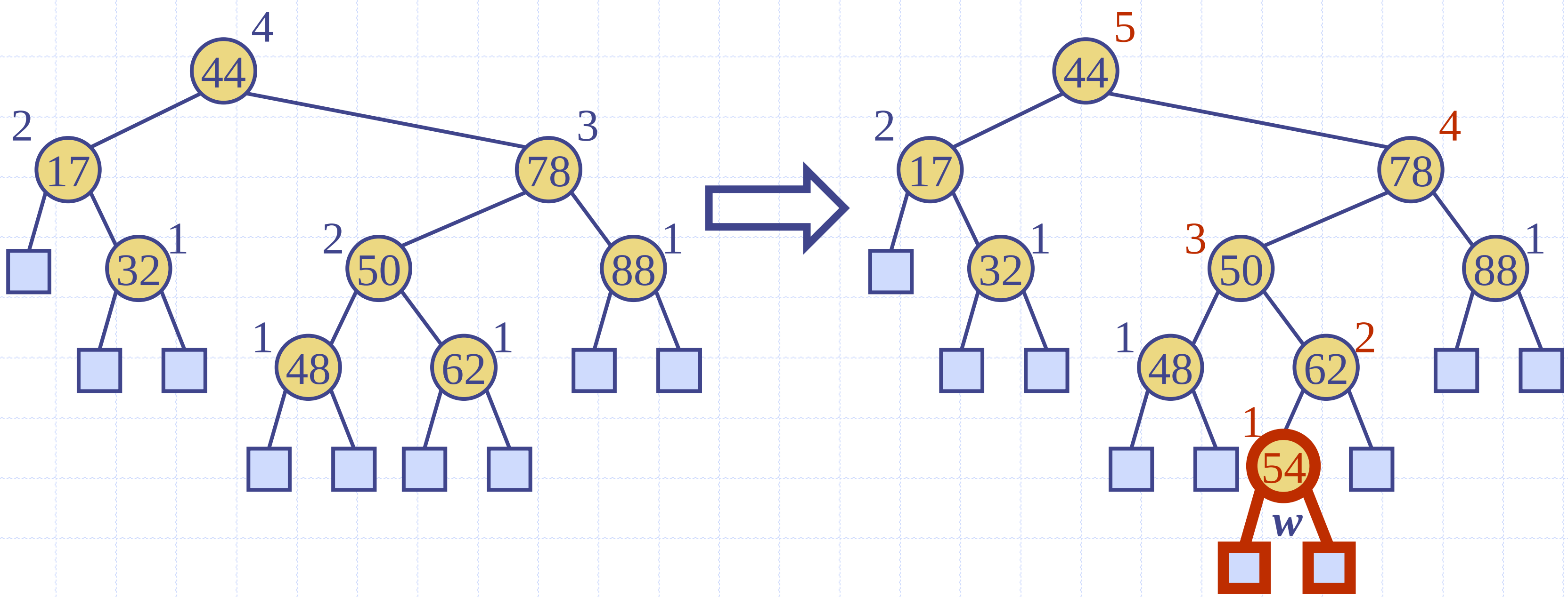
갱신 작업



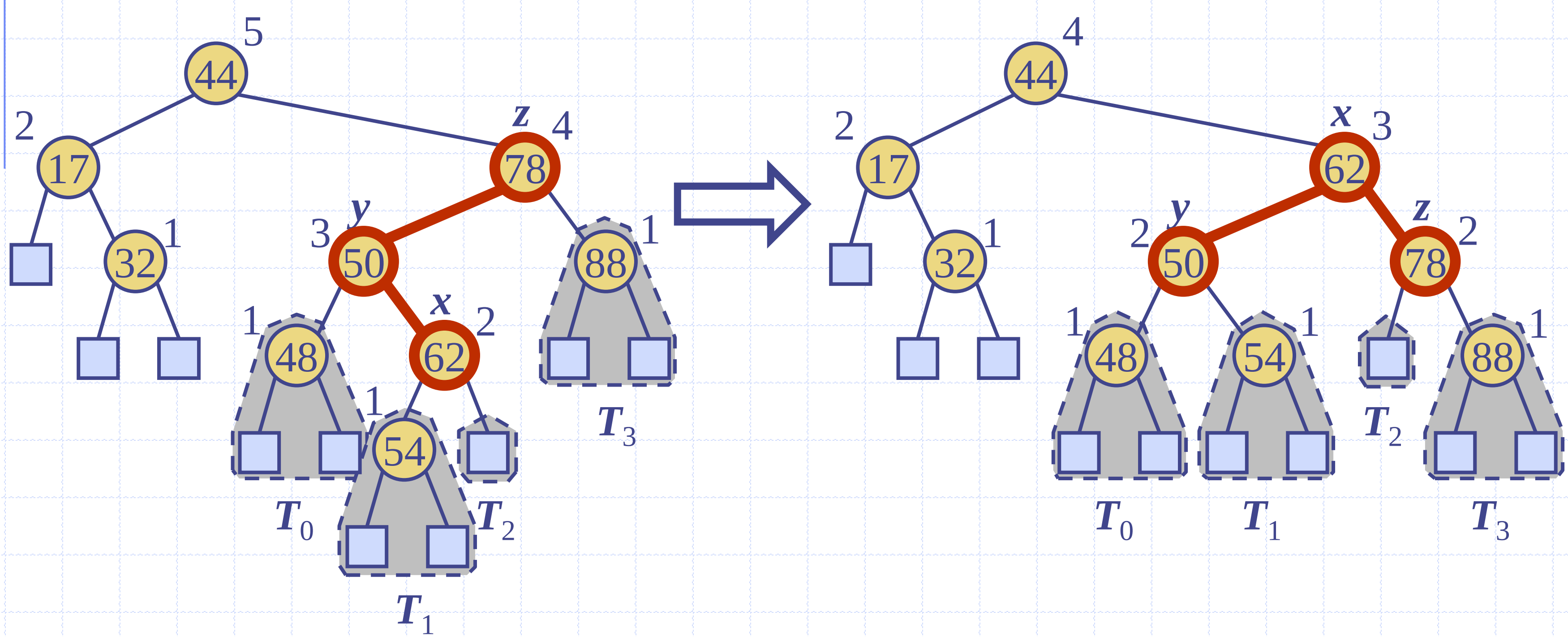
- AVL 트리에 대한 **삽입** 및 **삭제** 작업은 이진탐색트리에서의 삽입 및 삭제 작업과 유사
- 삽입이나 삭제 작업의 결과 AVL 트리의 **높이균형 속성**이 파괴될 수도 있다
- 그러므로 삽입이나 삭제 작업 후에는 혹시 생겼을지도 모를 불균형을 "**찾아서 수리**"해야 한다
  - **불균형 찾기:** 각 노드의 **균형검사**(balance check)를 통해 찾는다
  - **불균형 수리: 개조**(restructure)라 불리는 작업을 통해 트리의 높이균형 속성을 회복하기 위한 계산 작업을 수행

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



5
44
2
17
1
32
z
4
78
y
3
50
1
48
x
2
62
1
54
1
88
T_0
T_1
T_2
T_3
4
44
2
17
1
32
x
3
62
y
2
50
z
2
78
1
48
1
54
1
88
T_0
T_1
T_2
T_3

삽입

```
Alg insertItem(k, e)
    input AVL tree T, key k, element e
    output none

1. w ← treeSearch(root(), k)
2. if (isInternal(w))
        return
    else
        Set node w to (k, e)
        expandExternal(w)
        searchAndFixAfterInsertion(w)
        return
```

삽입 (conti.)

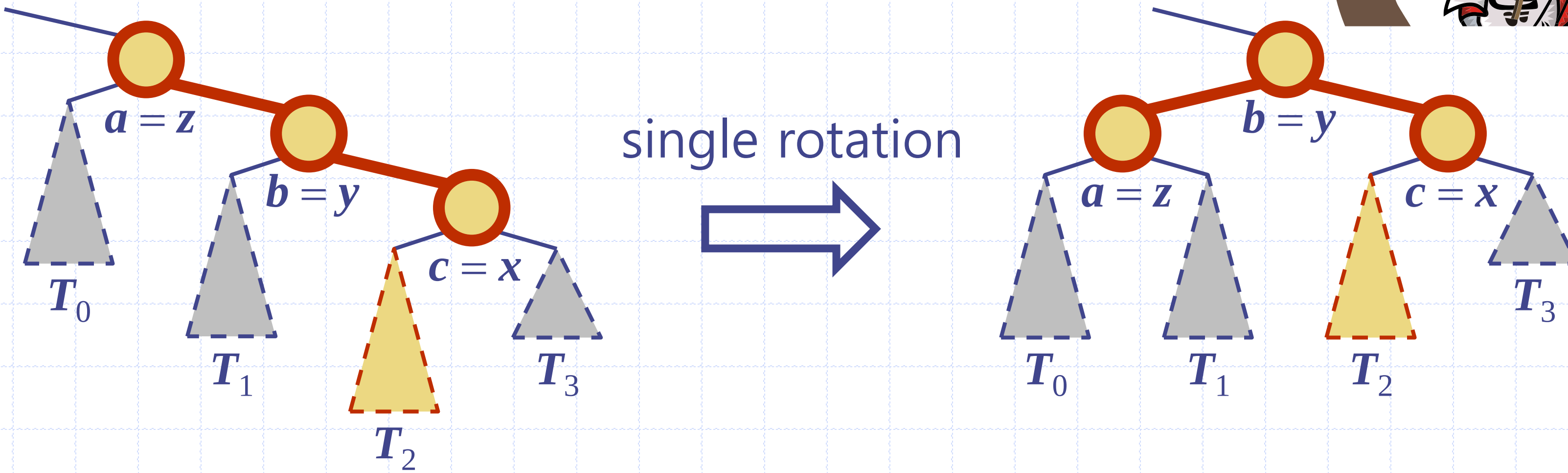
Alg ***searchAndFixAfterInsertion*(*w*)**
input internal node ***w***
output none

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 *지역적*으로나 *전역적*으로나 모두 복구된다}
5. **return**

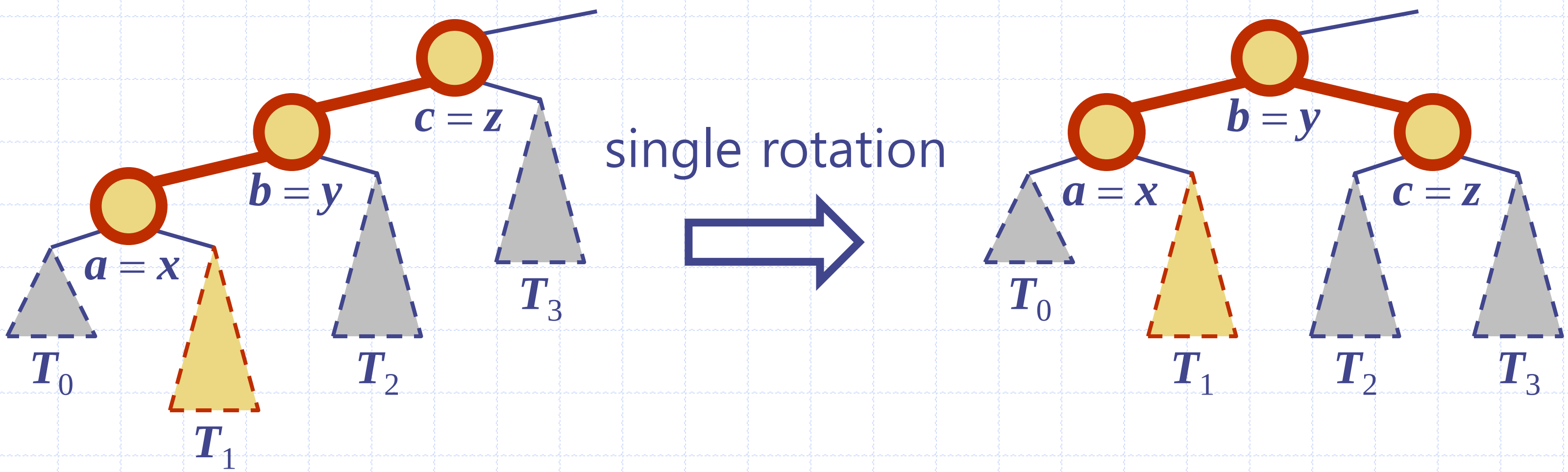
개조

- 개조는 종종 "**회전**(rotation)"이라고도 불린다
- **3-노드 개조**(trinode restructure)
  - 3대의 직계 노드 x, y(x의 부모), z(y의 부모)의 **중위순회 순서** a, b, c를 회전축으로 하여 수행
- **단일회전**(single rotation)
  - 만약 $b = y$면, y를 중심으로 z을 회전
- **이중회전**(double rotation)
  - 만약 $b = x$면, x를 중심으로 y를 회전한 후, 다시 x를 중심으로 z을 회전
- 좌우대칭 포함하여 모두 4종류의 회전 유형 존재

단일회전에 의한 개조

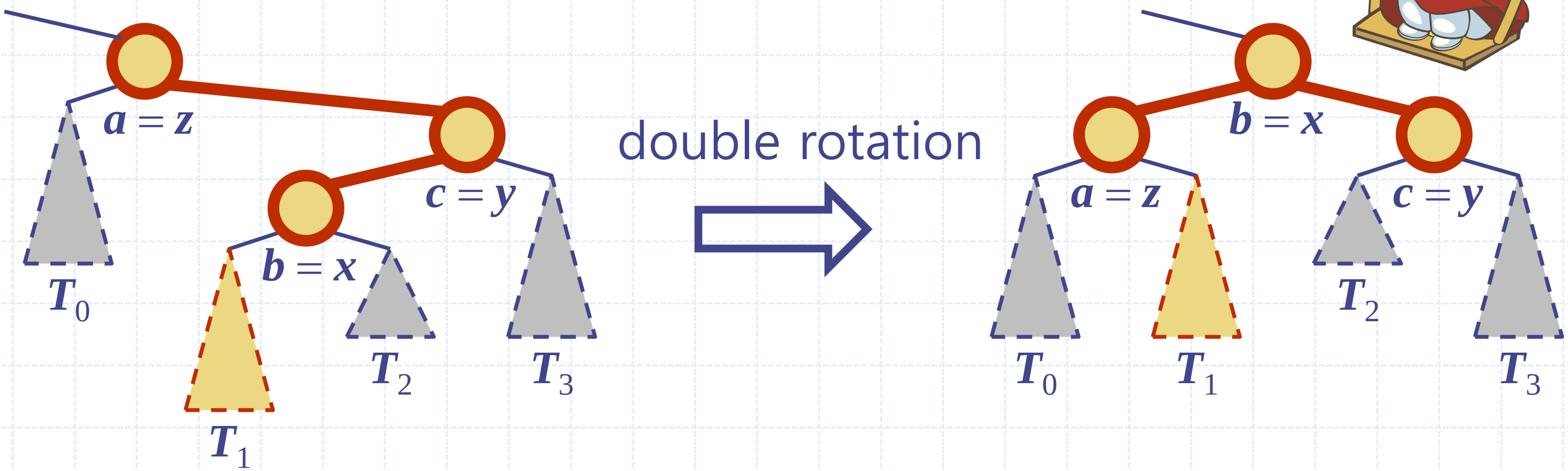


a = z
b = y
c = x
T0
T1
T2
T3
single rotation
b = y
a = z
c = x
T0
T1
T2
T3

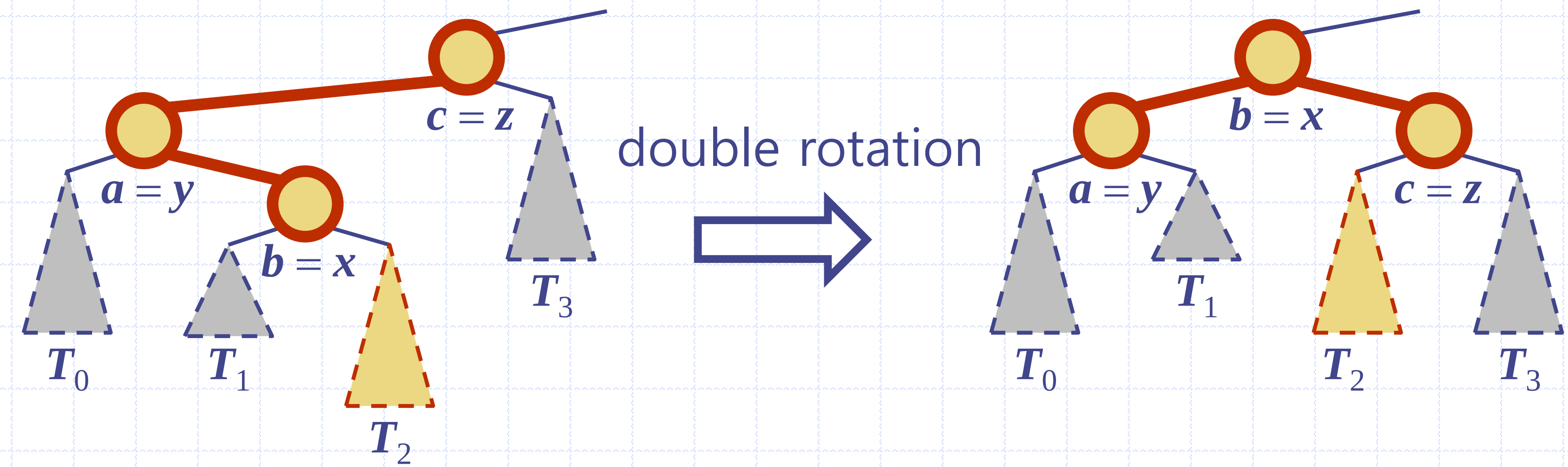


c = z
b = y
a = x
T0
T1
T2
T3
single rotation
b = y
a = x
c = z
T0
T1
T2
T3

이중회전에 의한 개조



$a = z$
$c = y$
$b = x$
T_0
T_1
T_2
T_3
double rotation
$b = x$
$a = z$
$c = y$
T_0
T_1
T_2
T_3



$c = z$
$a = y$
$b = x$
T_0
T_1
T_2
T_3
double rotation
$b = x$
$a = y$
$c = z$
T_0
T_1
T_2
T_3

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*)**

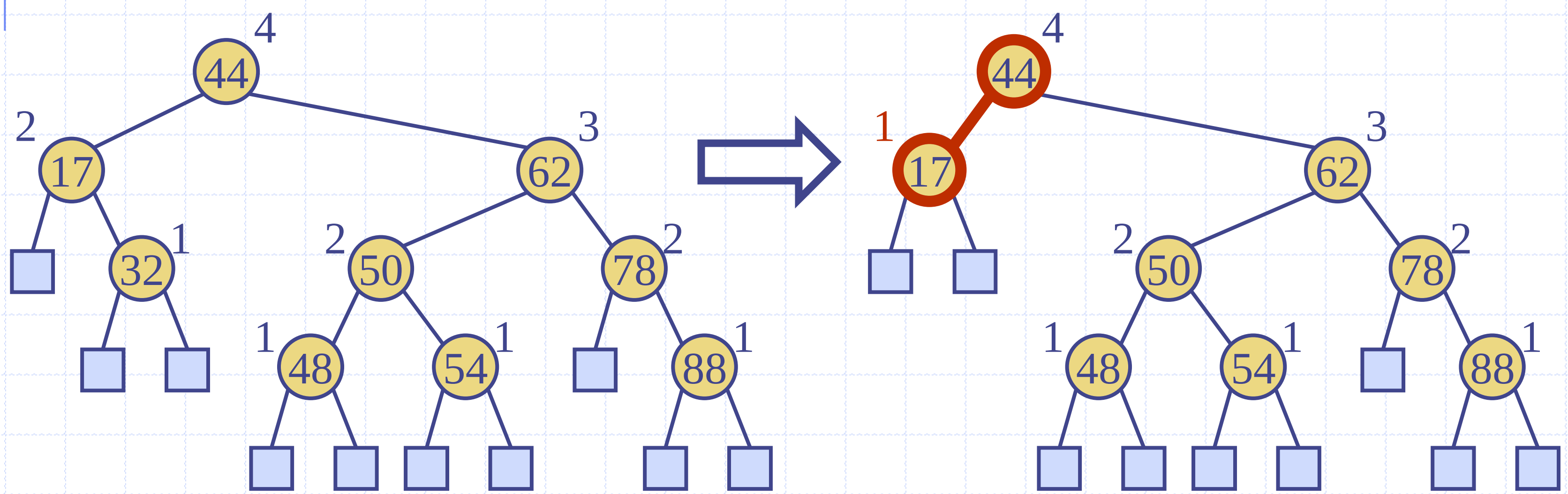
input a node ***x*** of a binary search tree ***T*** that has both a parent ***y*** and a grandparent ***z***

output tree ***T*** after restructuring involving nodes ***x***, ***y*** and ***z***

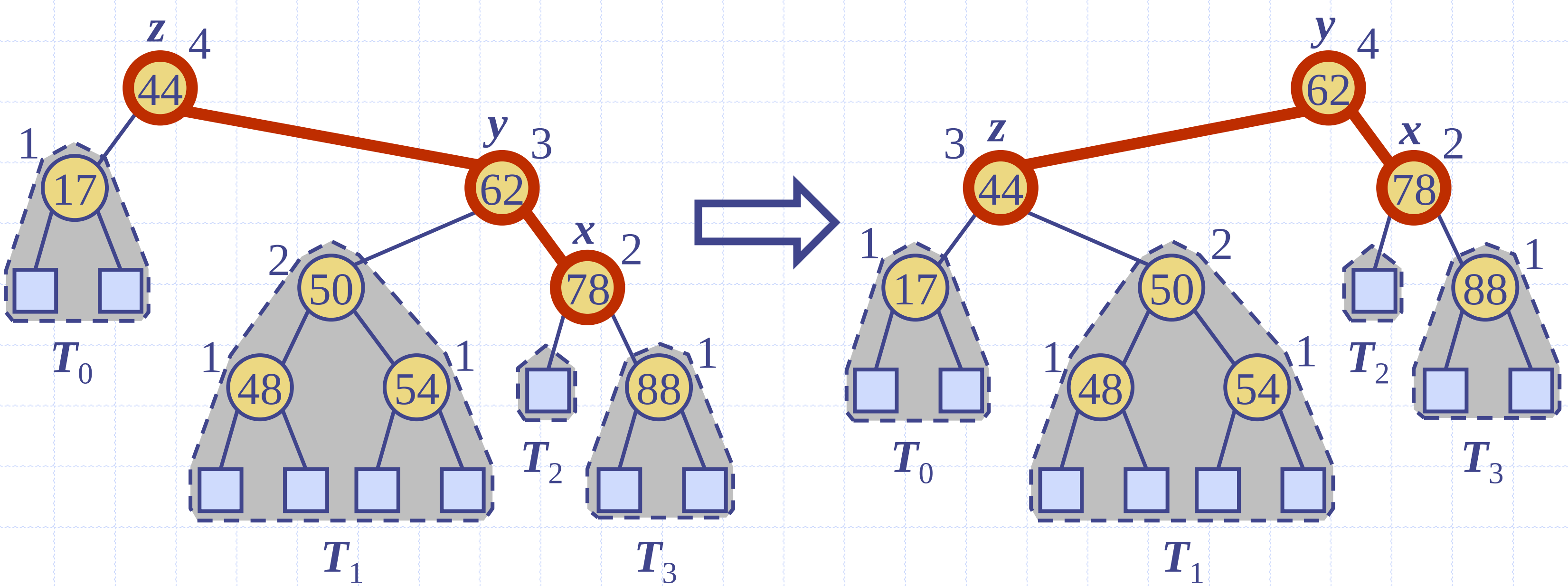
1. ***x***, ***y***, ***z***의 중위순회 방문 순서의 나열을 (***a***, ***b***, ***c***)라 하자.
2. ***x***, ***y***, ***z***의 부트리들 가운데 ***x***, ***y***, ***z***를 루트로 하는 부트리를 제외한 4개의 부트리들의 중위순회 방문순서의 나열을 (T_0, T_1, T_2, T_3)라 하자.
3. ***z***를 루트로 하는 부트리를 ***b***를 루트로 하는 부트리로 대체.
4. T_0와 T_1을 각각 ***a***의 왼쪽 및 오른쪽 부트리로 만든다.
5. T_2와 T_3를 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



z
4
44
y
3
62
x
2
78
1
17
2
50
1
48
1
54
1
88
T0
T1
T2
T3
y
4
62
z
3
44
x
2
78
1
17
2
50
1
48
1
54
1
88
T0
T1
T2
T3

삭제

Alg ***removeElement(k)***
input AVL tree ***T***, key ***k***
output element with key ***k***

1. ***w*** ← ***treeSearch***(***root***(), ***k***)
2. **if** (***isExternal***(***w***))
   return *NoSuchKey*
3. ***e*** ← ***element***(***w***)
4. ***z*** ← ***leftChild***(***w***)
5. **if** (!***isExternal***(***z***))
   z ← ***rightChild***(***w***)
6. **if** (***isExternal***(***z***)) {case 1}
   zs ← ***reduceExternal***(***z***)
   else {case 2}
   y ← ***inOrderSucc***(***w***)
   z ← ***leftChild***(***y***)
   Set node ***w*** *to* (***key***(***y***), ***element***(***y***))
   zs ← ***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*** ← ***restructure*(*x*, *y*, *z*)**
   {수행 후, 높이균형 속성은, 방금 전 ***z***를 루트로 했으나 이젠 변수 ***b***를 루트로 하는 부트리에서 *지역적*으로 복구된다. 하지만, 방금의 개조에 의해 ***b***를 루트로 하는 부트리의 높이가 1 줄어들 수 있으며 이때문에 ***b***의 조상이 균형을 잃을 수 있다. 즉, 삭제 후 1회의 개조만으로는 높이균형 속성을 *전역적*으로 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 n개의 항목으로 이루어진 **사전**을 전제하면
  - 공간사용량: $\mathbf{O}(n)$
  - 높이: $\mathbf{O}(\log n)$
  - **3-노드 개조**, 즉 한 번의 restructure: $\mathbf{O}(1)$
    - 연결 이진트리 사용을 전제
  - findElement: $\mathbf{O}(\log n)$
    - 개조 불필요
  - insertItem, removeElement: $\mathbf{O}(\log n)$
    - 초기의 treeSearch: $\mathbf{O}(\log n)$
    - 트리를 올라가면서 개조를 수행하여 높이균형을 회복: $\mathbf{O}(\log n)$