<알고리즘 실습> - 탐색트리

※ 입출력에 대한 안내

- 특별한 언급이 없으면 문제의 조건에 맞지 않는 입력은 입력되지 않는다고 가정하라.
- 특별한 언급이 없으면, 각 줄의 맨 앞과 맨 뒤에는 공백을 출력하지 않는다.
- 출력 예시에서 □는 각 줄의 맨 앞과 맨 뒤에 출력되는 공백을 의미한다.
- 입출력 예시에서 ↦ 이 후는 각 입력과 출력에 대한 설명이다.

*** 이 문제에서 사용되는 알고리즘은 교재를 중심으로 기술되었음.**

[문제 1] (이진탐색트리 구현) 주어진 조건을 만족하는 이진탐색트리를 구현하는 프로그램을 작성하라.

※ **조건:** 종료(**q**) 명령 때까지 삽입(**i**), 탐색(**s**), 삭제(**d**), 인쇄(**p**), 명령을 반복 입력받아 수행한다.

i <키> : 입력 <키>에 대한 노드 생성 및 트리에 삽입

d <키> : 입력 <키>가 트리에 존재하면 해당 노드 삭제 후 삭제된 키를 출력, 없으면 'X'를 출력

s <키> : 입력 <키>가 트리에 존재하면 해당 키를 출력, 없으면 'X'를 출력

p : 현재 트리를 전위순회로 인쇄

q : 프로그램 종료

주의:

1. 중복 키가 없는 것으로 전제한다.
2. 문제를 단순화하기 위해, 키만 존재하고 원소(element)는 없는 것으로 구현한다.
3. **main** 함수는 반복적으로 명령을 입력받기 전에 빈(empty) 이진탐색트리를 초기화해야 한다 – 즉, 외부노드 **1**개로만 구성된 이진트리를 말한다.

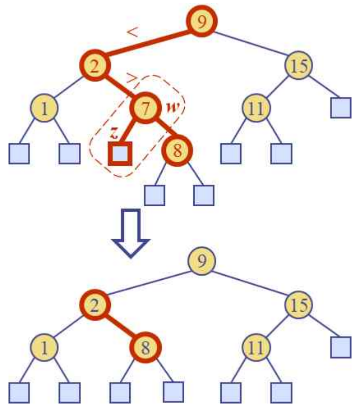
힌트:

1. 트리 **노드**는 아래의 **구조체**를 이용하여 구현한다.

<table>
<tr><td rowspan="2">lChild</td><td>parent</td><td rowspan="2">rChild</td></tr>
<tr><td>key</td></tr>
</table>

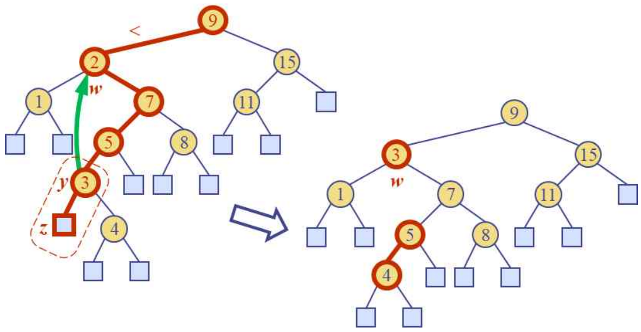
node

2. **전위순회**는 루트 노드를 먼저 방문하고, 왼쪽 부트리, 오른쪽 부트리 순서로 순회한다.
3. 트리 노드 **삭제** 시, 삭제할 노드 **w**의 자식 중 하나(**z**이라 하자)라도 잎인 경우는 아래 그림 <삭제 예시 1>처럼 **w**를 **z**과 함께 삭제하고, 반대쪽 자식 노드(그림에서 **8**을 저장한 노드)가 **w**를 계승한다 – **reduceExternal**(**z**) 함수 사용.



<삭제 예시 **1**>

4. 삭제할 노드 **w**의 자식 둘 다 내부 노드인 경우는 **w**의 **중위순회 후계자 y**가 삭제한 노드 위치에 오도록 한다. 중위순회 후계자를 찾는 방법은 오른쪽 자식으로 이동한 후, 거기서부터 왼쪽 자식들만을 끝까지 따라 내려가서 도달하게 되는 내부노드를 찾는 것이다(그림 <삭제 예시 **2**> 참고).



<삭제 예시 **2**>

입출력 형식:

1) **main** 함수는 아래 형식으로 명령을 표준입력받는다.

입력 : 문자(**i**, **d**, **s**)와 정수형 <키> 또는 문자(**p**, **q**)

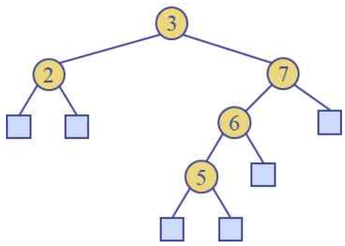
2) **main** 함수는 각 명령에 대해 아래 형식으로 표준출력한다.

출력 : 키 또는 X (**d**, **s** 명령인 경우)

트리의 전위순회 인쇄 (**p** 명령인 경우)

입력 예시 1		출력 예시 1	
i 3	↦ 3 삽입		
i 2	↦ 2 삽입		
i 7	↦ 7 삽입		
s 4	↦ 4 탐색	X	↦ 4 탐색 결과
i 6	↦ 6 삽입		

p i 5 s 6 q	↦ 전위순회 인쇄 ↦ 5 삽입 ↦ 6 탐색 ↦ 종료	□3 2 7 6 6	↦ 전위순회 인쇄 ↦ 6 탐색 결과



<구축된 이진탐색트리>

입력 예시 2		출력 예시 2	
i 9	↦ 9 삽입		
i 2	↦ 2 삽입		
i 15	↦ 15 삽입		
i 1	↦ 1 삽입		
i 7	↦ 7 삽입		
i 11	↦ 11 삽입		
i 5	↦ 5 삽입		
i 8	↦ 8 삽입		
i 3	↦ 3 삽입		
i 4	↦ 4 삽입		
p	↦ 전위순회 인쇄	□9 2 1 7 5 3 4 8 15 11	↦ 전위순회 인쇄
d 2	↦ 2 삭제	2	↦ 2 삭제 결과
d 13	↦ 13 삭제	X	↦ 13 삭제 결과
p	↦ 전위순회 인쇄	□9 3 1 7 5 4 8 15 11	↦ 전위순회 인쇄
q	↦ 종료		

주요 필요 함수:

○ **main**() 함수
- 인자: 없음
- 반환값: 없음
- 내용: 빈 트리를 초기화한 후 반복적으로 명령을 입력받아 처리

○ **findElement**(**k**) 함수
- 인자: 탐색 키 **k**
- 반환값: 원소(이 문제에서 원소 = 키)
- 내용: 현재 트리에서 키 **k**를 저장한 노드를 찾아 그 노드에 저장된 원소를 반환

○ **insertItem**(**k**) 함수
- 인자: 삽입 키 **k**
- 반환값: 없음

- 내용: 현재 트리에 키 **k**를 저장한 새 노드를 삽입

○ **treeSearch(k)** 함수
  - 인자: 탐색 키 **k**
  - 반환값: 키 **k**를 저장한 내부 노드 반환, 혹은 그런 노드가 없으면 만약 있었다면 위치할 외부 노드를 반환
  - 내용: 현재 트리에서 키 **k**를 저장한 노드를 반환

○ **removeElement(k)** 함수
  - 인자: 삭제 키 **k**
  - 반환값: 삭제된 원소(이 문제에서 원소 = 키)
  - 내용: 현재 트리에서 키 **k**를 저장한 노드를 삭제한 후 원소를 반환

○ **isExternal(w)** 함수
  - 인자: 노드 **w**
  - 반환값: 참 또는 거짓
  - 내용: 노드 **w**가 외부노드인지 여부를 반환

○ **isInternal(w)** 함수
  - 인자: 노드 **w**
  - 반환값: 참 또는 거짓
  - 내용: 노드 **w**가 내부노드인지 여부를 반환

○ **inOrderSucc(w)** 함수
  - 인자: 내부노드 **w**
  - 반환값: 내부노드
  - 내용: 노드 **w**의 중위순회 후계자를 반환

알고리즘 설계를 위한 의사코드 가이드:

```
Alg isExternal(w)
       input node w
       output boolean

1. if (w.left = ∅ and w.right = ∅)
       return True
   else {w.left ≠ ∅ or w.right ≠ ∅}
       return False
```

```
Alg isInternal(w)
       input node w
       output boolean

1. if (w.left ≠ ∅ or w.right ≠ ∅)
       return True
   else {w.left = ∅ and w.right = ∅}
       return False
```

```
Alg sibling(w)
       input node w
       output sibling of w

1. if (isRoot(w))
       invalidNodeException()                {root has no sibling}
2. if (leftChild(parent(w)) = w)
       return rightChild(parent(w))
   else
       return leftChild(parent(w))
```

```
Alg inOrderSucc(w)
       input internal node w
       output inorder successor of w

1. w ← rightChild(w)
2. if (isExternal(w))
       invalidNodeException()                {No inorder successor}
3. while (isInternal(leftChild(w)))
       w ← leftChild(w)
4. return w
```

```
Alg reduceExternal(z)
       input external node z
       output the node replacing the parent node of the removed node z

1. w ← z.parent
2. zs ← sibling(z)
3. if (isRoot(w))
       root ← zs                             {renew root}
       zs.parent ← ∅
   else
       g ← w.parent
       zs.parent ← g
       if (w = g.left)
              g.left ← zs
       else {w = g.right}
              g.right ← zs

4. putnode(z)                                {deallocate node z}
5. putnode(w)                                {deallocate node w}
6. return zs
```

* 수록되지 않은 함수의 의사코드는 교재의 코드를 참고할 수 있다. 하지만 가능하면 교재를 참조하지 않고 스스로 작성해본다.

[문제 2] (AVL 트리 생성) 주어진 조건을 만족하는 AVL 트리를 구현하는 프로그램을 작성하라.

1) 기본적인 입출력 구조는 문제 **1**과 동일하나 **삭제를 제외**한 삽입, 탐색, 출력을 구현한다.
2) **main** 함수에서 명령과 <키>를 입력받는다.
3) 명령에 따라 AVL 트리를 생성, 탐색한다.

주의: 문제 **1**과 동일

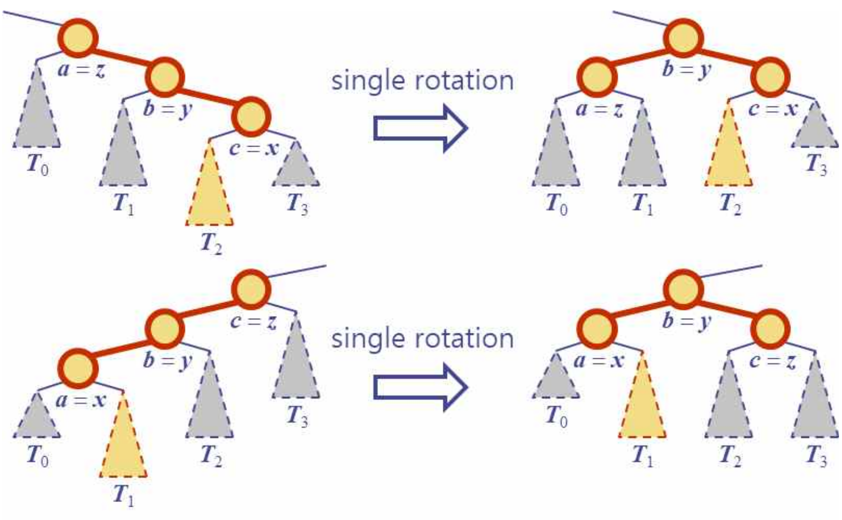
힌트:

1. 노드 구조체에 **높이**(height)를 추가한다.

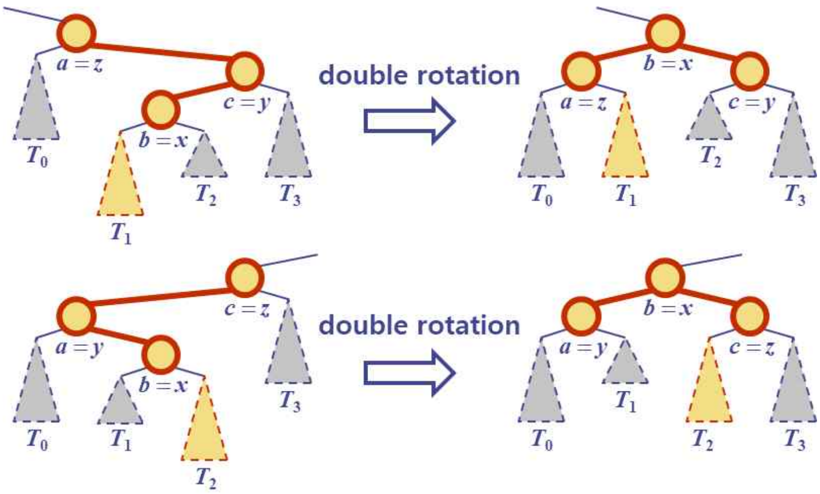
<table>
<tr><td rowspan="3">lChild</td><td>parent</td><td rowspan="3">rChild</td></tr>
<tr><td>key</td></tr>
<tr><td>height</td></tr>
</table>

node

2. 문제 **1**의 이진탐색트리에서 구현한 함수들을 그대로 또는 수정해서 사용하고 더 필요한 함수를 추가한다.
3. 키 삽입 후 트리에 불균형이 발생했을 경우, **개조**(restructure)를 수행한다. 개조는 종종 **회전**(rotation)이라고도 불리며, 좌우대칭을 포함하여 모두 **4**개 유형이 존재한다.



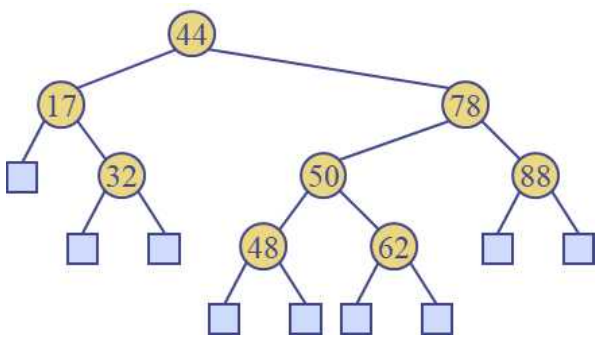
<단일회전(single rotation) 예시>



<이중회전(double rotation) 예시>

입출력 형식: 문제 **1**과 동일(삭제는 제외)

입력 예시 1		출력 예시 1	
i 44	↦ 44 삽입		
i 17	↦ 17 삽입		
i 78	↦ 78 삽입		
i 32	↦ 32 삽입		
i 50	↦ 50 삽입		
i 88	↦ 88 삽입		
i 48	↦ 48 삽입		
i 62	↦ 62 삽입		
s 88	↦ 88 탐색	88	↦ 탐색 결과
p	↦ 전위순회 인쇄	□44 17 32 78 50 48 62 88	↦ 전위순회 인쇄
q	↦ 프로그램 종료		



<구축된 AVL 트리>

필요 함수:

- **insertItem**(**k**) 함수
  - 인자: 삽입 키 **k**

- 반환값: 없음
- 내용: 현재 트리에 **k**를 저장한 새 노드 **w**를 삽입하고, **searchAndFixAfterInsertion**(**w**) 함수를 호출

○ **searchAndFixAfterInsertion**(**w**) 함수
- 인자: 내부노드 **w**
- 반환값: 없음
- 내용: 균형검사를 수행하고 불균형이 있으면 개조를 통해 높이균형 속성을 회복

○ **updateHeight**(**w**) 함수
- 인자: 내부노드 **w**
- 반환값: 참 또는 거짓
- 내용: 노드 **w**의 높이를 (필요하면) 갱신한 후 갱신 여부를 반환

○ **isBalanced**(**w**) 함수
- 인자: 내부노드 **w**
- 반환값: 참 또는 거짓
- 내용: 노드 **w**의 높이균형 여부를 반환

○ **restructure**(**x**, **y**, **z**) 함수
- 인자: 내부노드 **x**, **y**, **z**
- 반환값: 참 또는 거짓
- 내용: **3**-노드 개조를 수행한 후 (갱신된) **3**-노드의 루트를 반환

알고리즘 설계를 위한 의사코드 가이드:

```
Alg expandExternal(w)
        input external node w
        output none

1. l ← getnode()
2. r ← getnode()

3. l.left = ∅
4. l.right = ∅
5. l.parent = w
6. l.height = 0

7. r.left = ∅
8. r.right = ∅
9. r.parent = w
10. r.height = 0

11. w.left = l
12. w.right = r
13. w.height = 1
14. return
```

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

1. w ← treeSearch(root(), k)                    {삽입 키를 저장한 노드 찾기}

2. if (isInternal(w))                          {이미 존재하면 반환}
        return
   else
        Set node w to k
        expandExternal(w)
        searchAndFixAfterInsertion(w)
        return
```

다음 알고리즘은 교재의 연습문제 **11-15**의 답을 참고할 수 있다.

Alg searchAndFixAfterInsertion

Alg restructure

Alg updateHeight

Alg isBalanced

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

{Update heights and search for imbalance}
1. w.left.height, w.right.height, w.height
   ← 0, 0, 1
2. if (isRoot(w))
       return
3. z ← w.parent
4. while (updateHeight(z) & isBalanced(z))
       if (isRoot(z))
           return
       z ← z.parent
5. if (isBalanced(z))
       return
{Fix imbalance}
6. if (z.left.height > z.right.height)
       y ← z.left
   else {z.left.height < z.right.height}
       y ← z.right
7. if (y.left.height > y.right.height)
       x ← y.left
   else {y.left.height < y.right.height}
       x ← y.right
8. restructure(x, y, z)
9. return
                              {Total O(log n)}
```

```
Alg restructure(x, y, z)
   input internal node x, y, z, s.t. y is the parent of x and z is the parent of y
   output internal node

{Let (a, b, c) be an inorder listing of the nodes x, y, and z and let (T0, T1, T2, T3)
be an inorder listing of the four subtrees of x, y, and z not rooted at x, y, or z}
1. if (key(z) < key(y) < key(x))
       a, b, c ← z, y, x
       T0, T1, T2, T3 ← a.left, b.left, c.left, c.right
   elseif (key(x) < key(y) < key(z))
       a, b, c ← x, y, z
       T0, T1, T2, T3 ← a.left, a.right, b.right, c.right
   elseif (key(z) < key(x) < key(y))
       a, b, c ← z, x, y
       T0, T1, T2, T3 ← a.left, b.left, b.right, c.right
   else {key(y) < key(x) < key(z)}
       a, b, c ← y, x, z
       T0, T1, T2, T3 ← a.left, b.left, b.right, c.right
```

{Replace the subtree rooted at z with a new subtree rooted at b}

2. **if** ($isRoot(z)$)
    $root \leftarrow b$
    b.**parent** $\leftarrow Null$
  elseif (z.**parent**.**left** $= z$)
    z.**parent**.**left** $\leftarrow b$
    b.**parent** $\leftarrow z$.**parent**
  else {z.**parent**.**right** $= z$}
    z.**parent**.**right** $\leftarrow b$
    b.**parent** $\leftarrow z$.**parent**

{Let T_0 and T_1 be the left and the right subtree of a, resp.}

3. a.**left**, a.**right** $\leftarrow T_0, T_1$
4. T_0.**parent**, T_1.**parent** $\leftarrow a$
5. $updateHeight(a)$

{Let T_2 and T_3 be the left and the right subtree of c, resp.}

6. c.**left**, c.**right** $\leftarrow T_2, T_3$
7. T_2.**parent**, T_3.**parent** $\leftarrow c$
8. $updateHeight(c)$

{Let a and c be the left and the right child of b, resp.}

9. b.**left**, b.**right** $\leftarrow a, c$
10. a.**parent**, c.**parent** $\leftarrow b$
11. $updateHeight(b)$

12. **return** b

{Total **O**(1)}

Alg *updateHeight*(w)
  input internal node w
  output boolean

1. $h \leftarrow max(w.\mathbf{left}.\mathbf{height}, w.\mathbf{right}.\mathbf{height}) + 1$
2. **if** ($h \neq w$.**height**)
    w.**height** $\leftarrow h$
    return *True*
  else
    return *False*

Alg *isBalanced*(w)
  input internal node w
  output boolean

1. **return** $|w.\mathbf{left}.\mathbf{height} - w.\mathbf{right}.\mathbf{height}| < 2$

[문제 3] (AVL 트리 삭제) AVL 트리에서 **삭제**를 구현하라.

1) 기본적인 입출력 구조는 문제 **2**와 동일하며 삭제를 추가하여 구현한다.
2) **main** 함수에서 명령과 <키>를 입력받는다.
3) 명령에 따라 AVL 트리를 생성한다.

주의: 문제 **1**과 동일

힌트:

1. 문제 **2**에서 만든 함수들을 그대로 이용하고, **삭제 관련 함수**를 추가로 구현한다.
2. AVL 트리에서 삭제는 이진탐색트리에서와 동일하게 수행되지만, 마지막 단계에서 **reduceExternal** 작업으로 삭제된 노드의 부모 노드(그리고 조상 노드들)가 불균형이 될 수 있음에 유의한다.
3. 트리의 불균형을 해소하기 위한 개조는 문제 **2**에서 작성한 **restructure** 함수를 그대로 사용할 수 있다.

입출력 형식:

1) 문제 **1**과 동일

입력 예시 1		출력 예시 1	
i 9	↦ 9 삽입		
i 31	↦ 31 삽입		
i 66	↦ 66 삽입		
i 30	↦ 30 삽입		
i 1	↦ 1 삽입		
s 30	↦ 30 탐색	30	↦ 30 탐색 결과
i 24	↦ 24 삽입		
p	↦ 전위순회 인쇄	□30 9 1 24 31 66	↦ 전위순회 인쇄
s 47	↦ 47 탐색	X	↦ 47 탐색 결과
i 61	↦ 61 삽입		
d 30	↦ 30 삭제	30	↦ 30 삭제
i 13	↦ 13 삽입		
q	↦ 프로그램 종료		

필요 함수:

- **removeElement**(**k**) 함수
  - 인자: 삭제 키 **k**
  - 반환값: 삭제된 원소(이 문제에서 원소 = 키)
  - 내용: 현재 트리에서 **k**를 저장한 노드를 삭제한 후, **reduceExternal** 작업으로 삭제된 노드의 부모 노드 **w**에 대해 **searchAndFixAfterRemoval**(**w**) 함수를 이용하여 균형검사 및 수리를 수행

```
Alg removeElement(k)
        input AVL tree T, key k
        output key

1. w ← treeSearch(root(), k)                {삭제 키를 저장한 노드 찾기}
2. if (isExternal(w))                        {그런 노드가 없으면 반환}
        return NoSuchKey
3. z ← leftChild(w)
4. if (!isExternal(z))
        z ← rightChild(w)
5. if (isExternal(z))                        {case 1}
        zs ← reduceExternal(z)
   else                                      {case 2}
        y ← inOrderSucc(w)
        z ← leftChild(y)
        Set node w to key(y)
        zs ← reduceExternal(z)
6. searchAndFixAfterRemoval(parent(zs))
7. return k
```

```
Alg searchAndFixAfterRemoval(z)
    input internal node z
    output none

{Update heights and search for imbalance}
1. while (updateHeight(z) & isBalanced(z))
        if (isRoot(z))
            return
        z ← z.parent
2. if (isBalanced(z))
        return

{Fix imbalance}
3. if (z.left.height > z.right.height)
        y ← z.left
   else {z.left.height < z.right.height}
        y ← z.right
4. if (y.left.height > y.right.height)
        x ← y.left
   elseif (y.left.height < y.right.height)
        x ← y.right
   else {y.left.height = y.right.height}
        if (z.left = y)
            x ← y.left
        else {z.right = y}
            x ← y.right
5. b ← restructure(x, y, z)
6. if (isRoot(b))
        return
7. searchAndFixAfterRemoval(b.parent)
                                    {Total O(log n)}
```