Lecture #10: Binary Search Trees

School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeoung Nam

Search trees

- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree.
  - For complete binary tree with n nodes: worst case $\theta(lgn)$
  - For linear chain of n nodes: worst case $\theta(n)$
- Different types of search trees include binary search trees, red-black trees (covered in Chapter 13), and B-trees (covered in Chapter 18).
- We will cover binary search trees, tree walks, and operations on binary search trees.

Binary search trees

- 검색 속도가 빠르다. 이진 검색트리는 데이터가 정렬되어 저장되기 때문에, 검색할 때 원하는 값을 찾는데 필요한 평균 시간 복잡도는 O(log n)

- 데이터의 삽입과 삭제가 용이. 이진 검색트리는 노드를 삽입하거나 삭제할 때, 해당 노드를 탐색한 뒤 적절한 위치에 삽입하거나 삭제할 노드를 찾아서 제거하면 됨 이진 검색트리는 특정 노드를 찾는 데 필요한 시간복잡도가 O(log n)이기 때문에, 데이터의 삽입과 삭제가 매우 빠르다.

- 자료가 정렬되어 있기 때문에, 정렬된 데이터를 탐색할 때 매우 효율적이다. 이진 검색트리에서는 왼쪽 서브트리의 모든 노드 값이 현재 노드의 값보다 작고, 오른쪽 서브트리의 모든 노드 값이 현재 노드의 값보다 크기 때문에, 모든 노드를 중위 순회(inorder traversal)하면 정렬된 순서대로 데이터를 얻을 수 있다.

Binary search trees

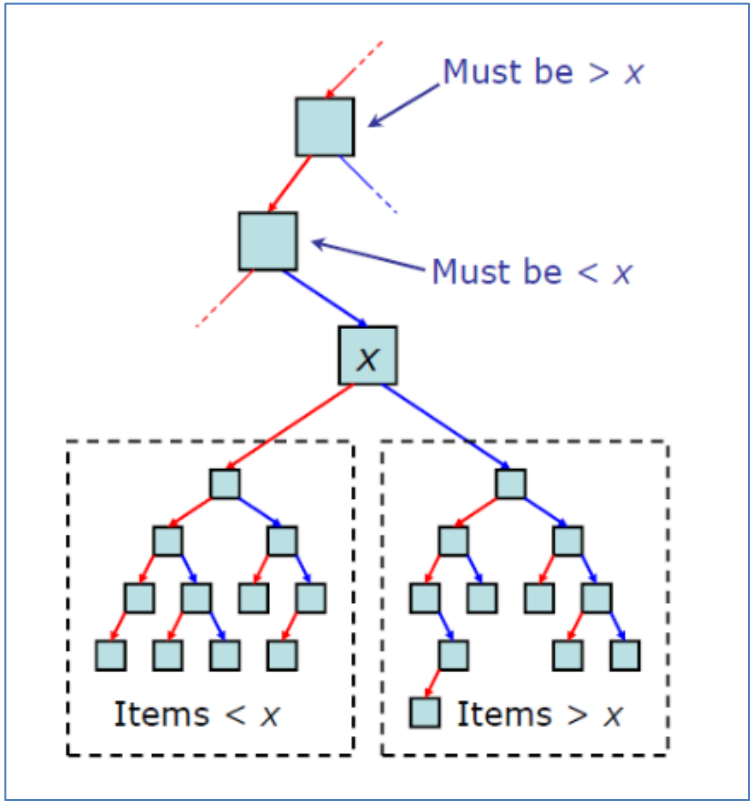
- 모든 노드의 왼쪽 서브트리에 있는 노드들은 해당 노드보다 작은 값을 가짐
- 모든 노드의 오른쪽 서브트리에 있는 노드들은 해당 노드보다 큰 값을 가짐

- 이진 검색 트리의 특징
  - 1. 모든 노드는 최대 두 개의 자식 노드를 갖는다.
  - 2. 모든 왼쪽 자식 노드는 부모 노드보다 작은 값을 갖는다.
  - 3. 모든 오른쪽 자식 노드는 부모 노드보다 큰 값을 갖는다.
- 이진 검색 트리의 노드들은 일정한 규칙에 따라 정렬되어 있으므로, 특정 값의 검색이 가능

- Key 값을 통한 검색
  - 루트 노드부터 시작하여 검색하고자 하는 값과 비교하며, 해당 값보다 작으면 왼쪽 자식 노드로 이동하고, 크면 오른쪽 자식 노드로 이동

Binary search trees

- 각 노드의 왼쪽 서브트리에는 해당 노드의 값보다 작은 값을 지닌 노드들로 이루어져 있다.
- 각 노드의 오른쪽 서브트리에는 해당 노드의 값보다 큰 값을 지닌 노드들로 이루어져 있다.
- 중복된 노드가 없어야 한다.
- 왼쪽 서브트리, 오른쪽 서브트리 또한 이진탐색트리이다.



Binary search trees

- Accomplish many dynamic-set operations in O(h) time, where h height of tree.
- *T.root* points to the root of tree T
- Each node contains the attributes
  - *key* (and possibly other satellite(부속) data).
  - *left*: points to left child.
  - *right*: points to right child.
  - *p*: points to parent. *T.root.p = NIL*
- Stored keys must satisfy the binary-search-tree property
  - If y is in left subtree of x, then y.key $\leq$ x.key.
  - If y is in right subtree of x, then y.key $\geq$ x.key.

중위 트리 순회(inorder tree walk)

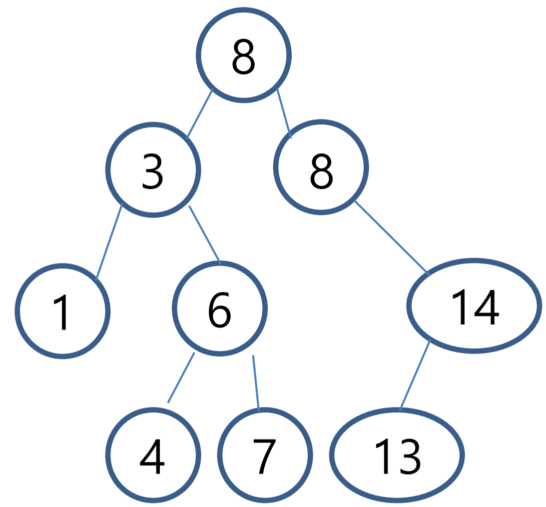
- 이진트리의 특성때문에 간단한 재귀호출로 이진검색 트리의 모든 키를 정렬된 순서대로 호출 가능
- 왼쪽 서브트리, 루트 노드, 오른쪽 서브트리 순서로 순회를 진행
  - 왼쪽 서브트리 탐색: 먼저, 현재 노드의 왼쪽 서브트리를 재귀적으로 탐색. 이 과정은 왼쪽 서브트리가 존재할 때까지 반복.
  - 현재 노드 처리: 왼쪽 서브트리의 탐색이 완료되면, 현재 노드를 처리. 이는 노드의 값을 출력하거나, 다른 처리를 수행하는 것.
  - 오른쪽 서브트리 탐색: 현재 노드를 처리한 후, 오른쪽 서브트리를 재귀적으로 탐색. 이 과정은 오른쪽 서브트리가 존재할 때까지 반복.

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

중위 트리 순회(inorder tree walk)

- **예제**
  - 왼쪽 서브트리 순서대로 탐색
  - 루트 노드 처리
  - 오른쪽 서브트리 탐색

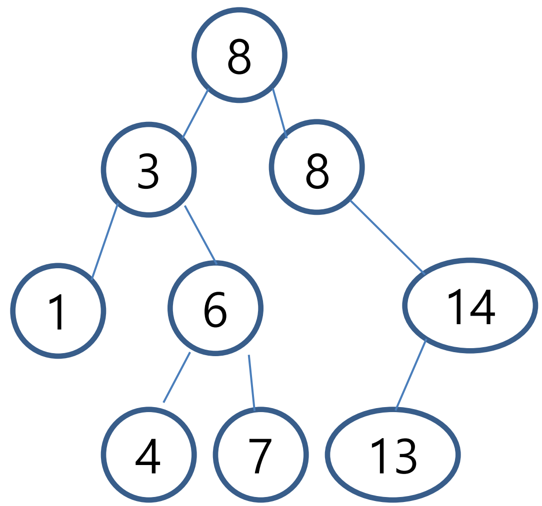


- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

INORDER-TREE-WALK(x)
  if $x \neq$ NIL
    INORDER-TREE-WALK($x.left$)
    print $key[x]$
    INORDER-TREE-WALK($x.right$)

중위 트리 순회(inorder tree walk)

- 예제
  - 노드 1 방문
  - 노드 3 방문
  - 노드 4 방문
  - 노드 6 방문
  - 노드 7 방문
  - 노드 8 방문
  - 노드 10 방문
  - 노드 13 방문
  - 노드 14 방문



- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

중위 트리 순회(inorder tree walk)

- **Theorem 12.1(pg. 289)**
  - X가 n개의 노드로 이루어진 서브 트리의 루트면 Inorder-tree-walk(x)는 $\boldsymbol{\theta(n)}$시간이 걸린다

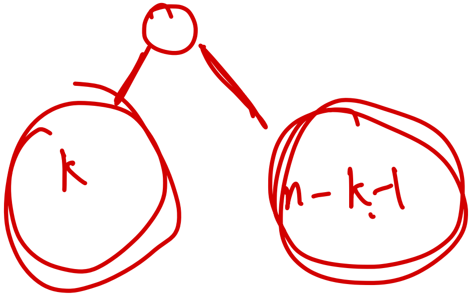
$\Omega(n) < \theta(n) < O(n)$

- **Proof**
  - Inorder-tree-walk를 n개의 노드 트리에 대해 호출했을때 시간: T(n)
  - N개의 노드를 모두 방문하기 때문에 $\boldsymbol{\Omega(n)}$이다
  - $\boldsymbol{T(n)}$이 $\boldsymbol{O(n)}$임을 보이면 된다

$T(n) \le T(k) + T(n-k-1) + d$

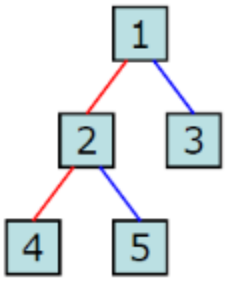
$T(n) \le$

  - n=0일때(빈 트리) T(0)는 경미한 상수 c
  - n>0인 경우 왼쪽 서브트리 k개 노드, 오른쪽 서브트리 n-k-1개 있다고 가정
  - 수행시간 $T(n) <= T(k) + T(n-k-1) + d, d > 0$는 재귀호출에 걸리는 시간 상한
  - $T(n) <= (c+d)n + c$ 치환법 사용
  - n=0일때 $(c+d) * 0 + c \;=\; c \;=\; T(0)$
  - n>0일때
    - $T(n) \le \; T(k) + T(n-k-1) + d$
    - $\le ((c+d)k + c) + ((c+d)(n-k-1) + c) + d$
    - $= (c+d)n - (c+d) + c + d$
    - $= (c+d)n + c$



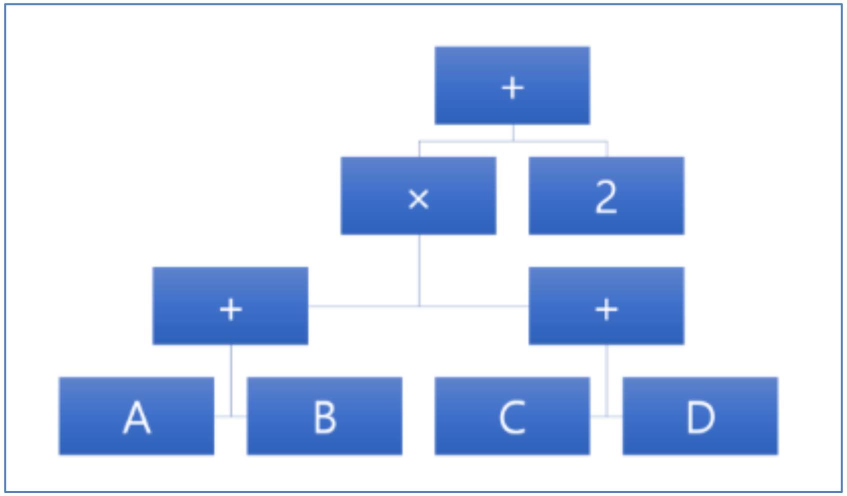
트리 순회(tree traversal)

- 트리순회(tree traversal)란 트리의 각 노드를 체계적인 방법으로 방문하는 과정
- 전위순회(preorder), 중위순회(inorder), 후위순회(postorder)
- Preorder, 깊이우선순회(depth-first traversal)
  - 루트 노드에서 시작해서 노드-왼쪽 서브트리-오른쪽 서브트리 순으로 순회하는 방식
  - 1, 2, 4, 5, 3
- Inorder, 대칭순회(Symmetric traversal)
  - 루트 노드에서 시작해서 왼쪽 서브트리-노드-오른쪽 서브트리 순으로 순회하는 방식
  - 4, 2, 5, 1, 3
- Postorder,
  - 루트 노드에서 시작해서 왼쪽 서브트리-오른쪽 서브트리-노드 순으로 순회하는 방식
  - 4, 5, 2, 3, 1



트리 순회(tree traversal)

- **Ex) 사칙연산**
  - $(A + B) * (C + D) + 2$
  - **후위순회**
  - **A,B,+,C,D,+,*,2,+**
  - **전위순회**
  - **+,*,+,A,B,+,C,D,2**



Querying a binary search tree

- Search, Minimum, Maximum, Successor, Predecessor 등의 질의가 가능

Querying a binary search tree

- **Searching**
  - 루트에 대한 포인터, 키 k
  - 키 k인 노드 존재한다면 포인터 리턴
  - 없으면 nil 리턴
- 작으면 왼쪽, 크면 오른쪽
- 수행시간은 트리의 높이 O(h)

Searching

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

Initial call is TREE-SEARCH(T.root, k).

x == NIL or (k == key[x])

return x

if k < x.key

x == NIL or k == key[x]

Querying a binary search tree

- **Minumum and maximum**
  - the minimum key of a binary search tree is located at the leftmost node
  - the maximum key of a binary search tree is located at the rightmost node
- 둘의 관계는 대칭적
- 높이가 h인 트리에 대해 O(h), 하나의 단순경로만 존재

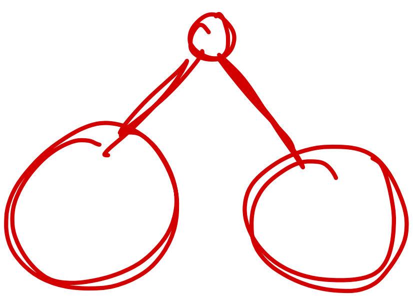
```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x

TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

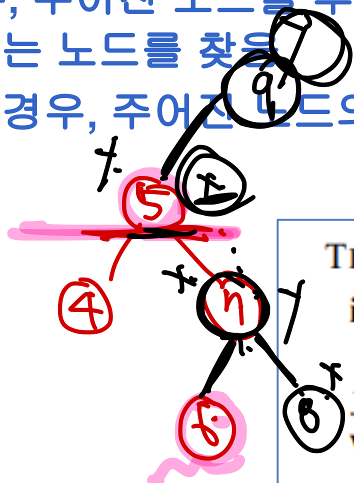
x.left ≠ NIL
  x=x.left
return x

Querying a binary search tree

if

- **Successor(직후원소)** → 그중에서 가장 작은 거
  - 주어진 노드의 바로 다음에 위치한 노드를 의미
  - 다음 노드는 해당 노드의 오른쪽 서브트리에서 가장 작은 값 또는 왼쪽 서브트리에서 해당 노드보다 큰 값 중 가장 작은 값을 갖는 노드
- **과정**
  - ① 주어진 노드의 오른쪽 서브트리에서 가장 작은 값을 갖는 노드를 찾음.
  - 오른쪽 서브트리가 비어 있지 않은 경우, 해당 노드가 주어진 노드의 successor가 됨.
  - ② 오른쪽 서브트리가 비어 있는 경우, 주어진 노드를 루트로 하는 서브트리에서 해당 노드보다 큰 값 중 가장 작은 값을 갖는 노드를 찾음.
  - 이 과정을 더 이상 진행할 수 없는 경우, 주어진 노드의 successor는 존재하지 않음.
  - .p는 parent





```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

→ 오른쪽 subtree가 nil이 아니면.

x가 y의 left 자식이 된 순간 멈추고 y return.

successor (x)

후행자

if x.right ≠ NIL

TREE-MINIMUM (x.right)

y = x.p

x가 y의 오른쪽 자식 인지

while y ≠ NIL and x == y.right

x = y

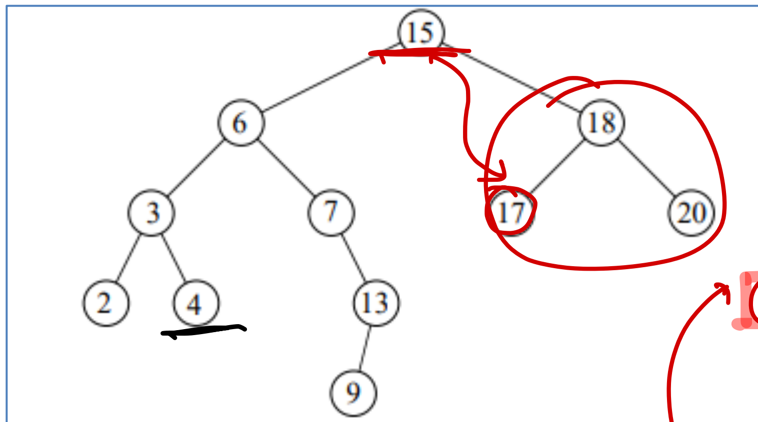
y = y.p

Querying a binary search tree

- **Predecessor(직전원소)**
  - 주어진 노드의 바로 이전에 위치한 노드를 의미
  - 이전 노드는 해당 노드의 왼쪽 서브트리에서 가장 큰 값 또는 오른쪽 서브트리에서 해당 노드보다 작은 값 중 가장 큰 값을 갖는 노드
- **과정**
  - 주어진 노드의 왼쪽 서브트리에서 가장 큰 값을 갖는 노드를 찾음.
  - 왼쪽 서브트리가 비어 있지 않은 경우, 해당 노드가 주어진 노드의 predecessor가 됨.
  - 왼쪽 서브트리가 비어 있는 경우, 주어진 노드를 루트로 하는 서브트리에서 해당 노드보다 작은 값 중 가장 큰 값을 갖는 노드를 찾음.
  - 이 과정을 더 이상 진행할 수 없는 경우, 주어진 노드의 predecessor는 존재하지 않음.

Querying a binary search tree

- **Ex)**



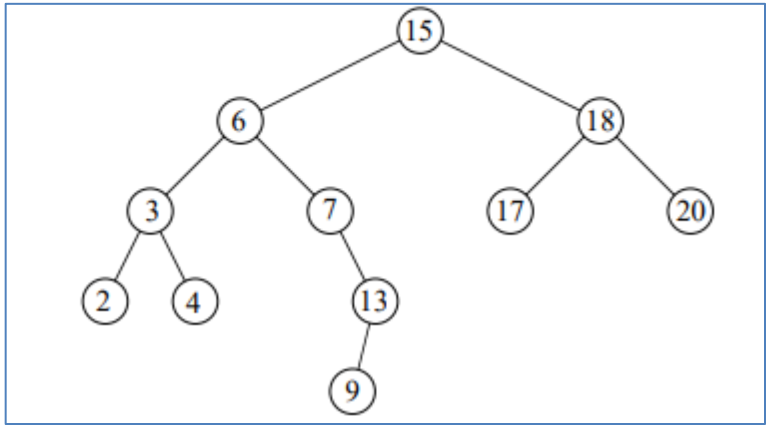
- Find the successor of the node with key value 15.
- Find the successor of the node with key value 6.
- Find the successor of the node with key value 4.
- Find the predecessor of the node with key value 6.

오서의 min.

왼쪽 처음

Querying a binary search tree

- **Ex)**



- **Find the successor of the node with key value 15. (Answer: Key value 17)**
- **Find the successor of the node with key value 6. (Answer: Key value 7)**
- **Find the successor of the node with key value 4. (Answer: Key value 6)**
- **Find the predecessor of the node with key value 6. (Answer: Key value 4)**

Querying a binary search tree

- **Time**
  - For both the TREE-SUCCESSOR and TREE-PREDECESSOR procedures, in both cases, we visit nodes on a path down the tree or up the tree
  - Thus, running time is O(h), where h is the height of the tree.

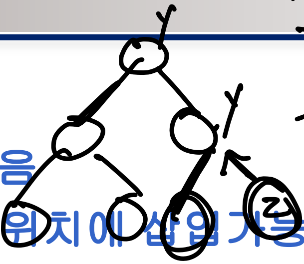
Insertion and deletion

- 삽입과 삭제를 통해 이진트리로 표현된 동적집합을 변형
- 계속해서 이진검색 트리의 특성을 유지해야된다
- 삽입
  - 루트 노드부터 시작하여, 삽입할 노드를 삽입할 위치를 찾습니다.
  - 삽입할 위치를 찾은 후, 해당 위치에 새로운 노드를 삽입합니다.
  - 만약 삽입할 위치에 이미 노드가 존재하는 경우, 해당 노드를 왼쪽 또는 오른쪽 서브트리 중 하나로 이동시킵니다.
  - 삽입된 노드를 기준으로, 트리의 구조를 다시 조정하여 이진트리의 규칙을 유지합니다.
- 삭제
  - 삭제할 노드를 찾습니다.
  - 삭제할 노드의 자식 노드가 2개인 경우, 삭제할 노드의 오른쪽 서브트리에서 가장 작은 값을 갖는 노드를 찾아 해당 노드를 삭제할 노드 위치에 대체합니다.
  - 삭제할 노드의 자식 노드가 1개인 경우, 삭제할 노드의 자식 노드를 삭제할 노드 위치에 대체합니다.
  - 삭제할 노드의 자식 노드가 없는 경우, 삭제할 노드를 단순히 삭제합니다.
  - 삭제된 노드를 기준으로, 트리의 구조를 다시 조정하여 이진트리의 규칙을 유지합니다.

Insertion

TREE-INSERT(T, z)
y=NIL
x=T.root
while x != NIL
y=x
if z.key < x.key
x = x.left
else x = x.right
z.p = y
if y==NIL
T.root=z
else if z.key < y.key
y.left = z

- **Insertion**
  - 노드 z를 인자로 받음
  - Z가 트리의 적절한 위치에 삽입가능하도록 함



Insertion

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z        // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

y
z.key < y.key
y.left = z.
parent도 연결하려면
방향 정보가 필요 없음.

- To insert value v into the binary search tree, the procedure is given node z, with $z.key = v$, $z.left =$ NIL, and $z.right =$ NIL.
- Beginning at root of the tree, trace a downward path, maintaining two pointers.
  - Pointer x: traces the downward path.
  - Pointer y: "trailing pointer" to keep track of parent of x.
- Traverse the tree downward by comparing the value of node at x with v, and move to the left or right child accordingly.
- When x is NIL, it is at the correct position for node z.
- Compare z's value with y's value, and insert z at either y's *left* or *right*, appropriately.

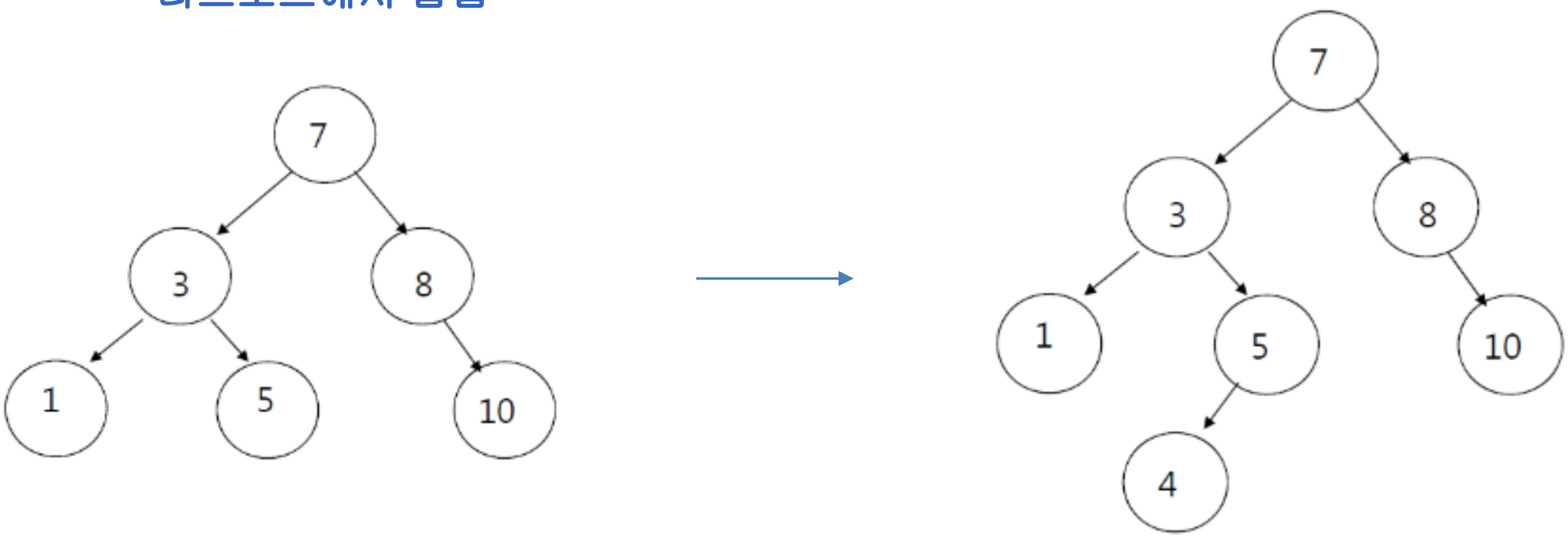
Insertion

- **Insertion**
  - 노드 z를 인자로 받음
  - Z가 트리의 적절한 위치에 삽입가능하도록 함
- **Ex) 4를 삽입**
  - 이진탐색트리가 커질 경우 이렇게 트리의 중간에 새 데이터를 삽입하게 되면 서브트리의 속성이 깨질 수 있기 때문에 삽입 연산은 반드시 리프노드에서 삽입

Insertion

TREE-INSERT(T, z)

y = NIL
x = $T.root$
while $x \neq$ NIL
  $y = x$
  if $z.key < x.key$
    $x = x.left$
  else $x = x.right$
$z.p = y$
if y == NIL
  $T.root = z$ // tree T was empty
elseif $z.key < y.key$
  $y.left = z$
else $y.right = z$



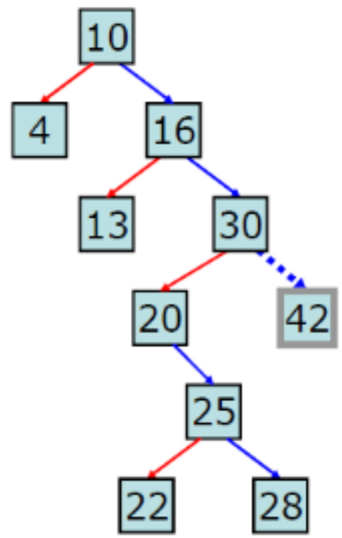
Deletion

→ 시험 볼 듯…?

- Insertion 보다 복잡함
  - 세가지 경우의 수를 따져서 진행
  - Z가 자식 노드가 없는 경우
    - Z의 부모의 자식노드를 NIL로 가리키게 바꿈으로서 손쉽게 삭제
  - Z가 자식 노드가 하나 있는 경우
    - Z의 자식노드를 z의 자리로 상승시킴
  - Z가 자식 노드가 두개 있는 경우
    - Z의 직후원소 y를 찾고 y가 z의 자리를 차지하도록 한다

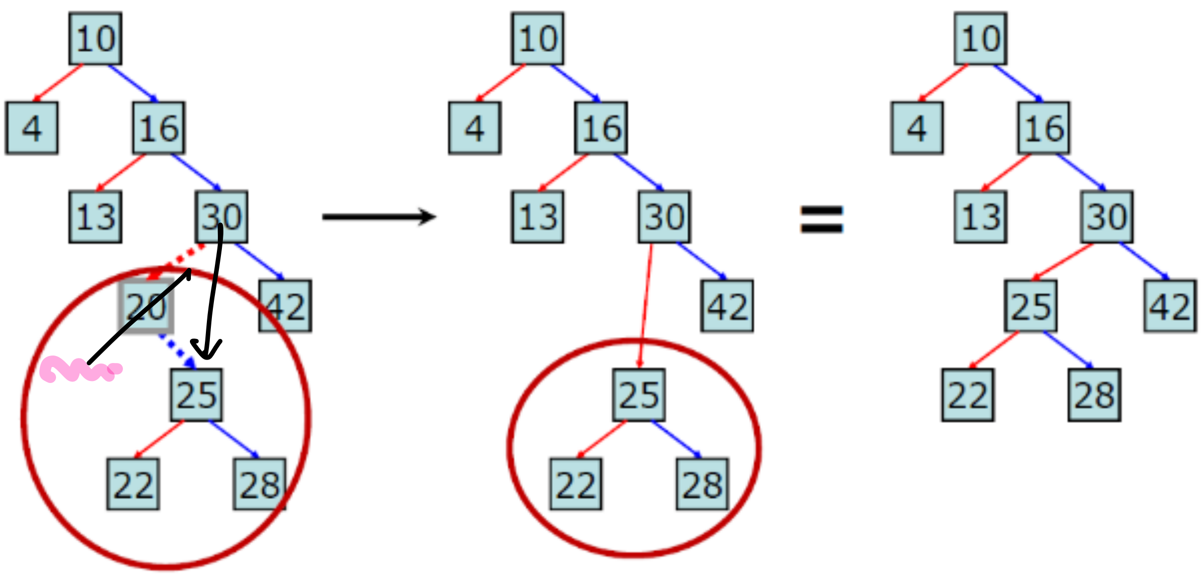
Deletion

- **Case1: 자식노드가 없는 경우** → 그냥 삭제
  - 예시그림 에서 42를 삭제
  - 자식노드가 없기 때문에 그냥 삭제 가능
  - 이진검색트리의 속성을 깨트리지 않는다



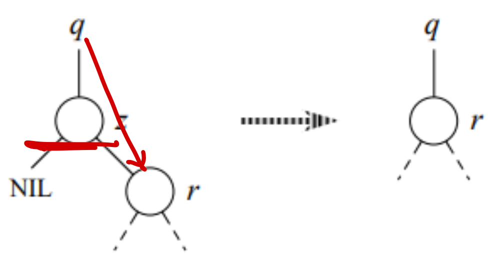
Deletion

- **Case2: 자식노드가 하나인 경우**
  - 해당 노드를 지우고, 해당노드의 자식노드와 부모노드를 연결
  - 왼쪽 자식이 없다면?
    - 오른쪽 자식과 교체
    - 반대인 케이스에는 왼쪽 자식과 교체
    - 20을 루트노드로 하는 서브트리의 모든 값은 20의 부모노드인 30보다 작거나 같음
    - 하나뿐인 자식노드(25)와 부모노드(30)를 연결해도 이진탐색트리의 속성이 깨지지 않음

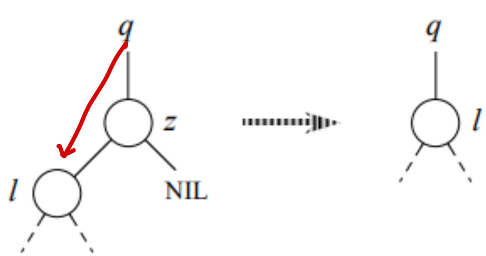


Deletion

- **Case2: 자식노드가 하나인 경우**
  - 해당 노드를 지우고, 해당노드의 자식노드와 부모노드를 연결
  - 왼쪽 자식이 없다면?
    - 오른쪽 자식과 교체
    - 반대인 케이스에는 왼쪽 자식과 교체
    - If z has no left child, replace z by its right child. The right child may or may not be NIL. (If z's right child is NIL, then this case handles the situation in which z has no children.)

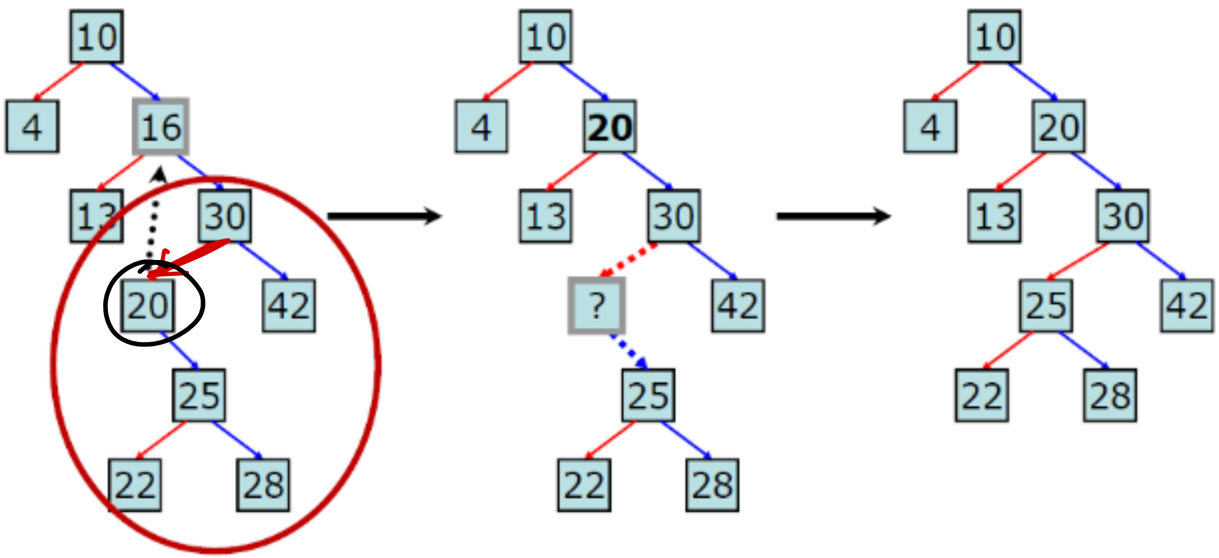


    - If z has just one child, and that child is its left child, then replace z by its left child.



Deletion

- **Case3: 자식노드가 두개인 경우**
  - 예제 그림에서 16 삭제한다고 가정
  - 무작정으로 삭제시 트리의 속성이 깨질 수 있다

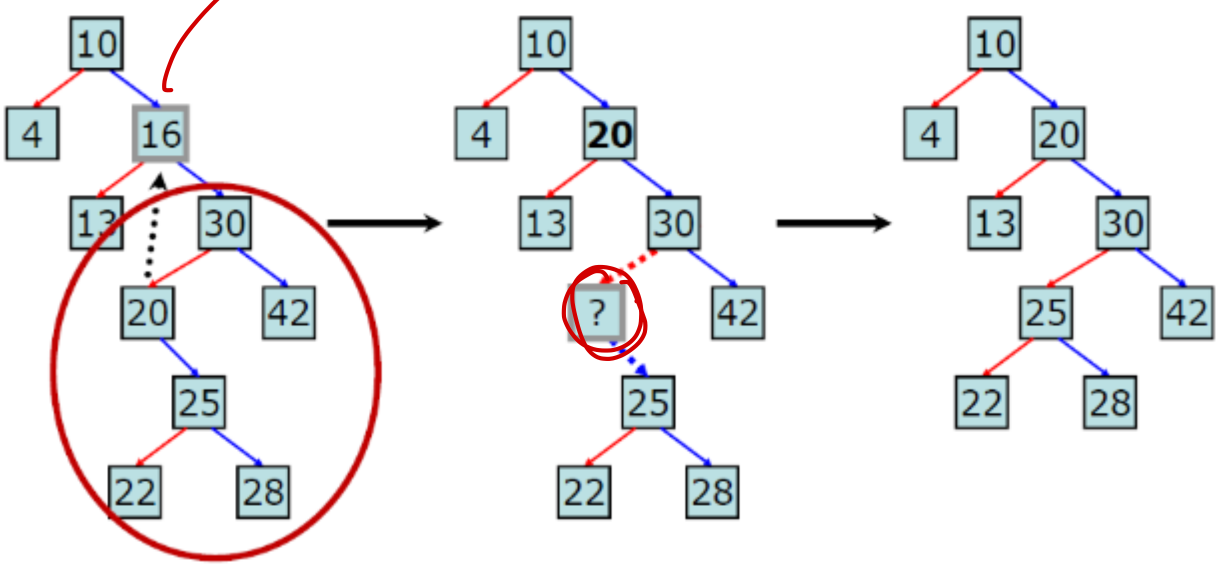


Deletion

- **Case3: 자식노드가 두개인 경우**
  - 중위순회방식(왼쪽서브트리-노드-오른쪽서브트리 순회)
  - 4,10,13,16,20,22,25,28,30,42
  - Predecessor: 13(왼쪽 서브트리 중 최대값)
  - Successor: 20(오른쪽 서브트리중 최소값)
  - 16의 위치에 13 또는 20을 복사 후 기존 위치의 노드를 삭제

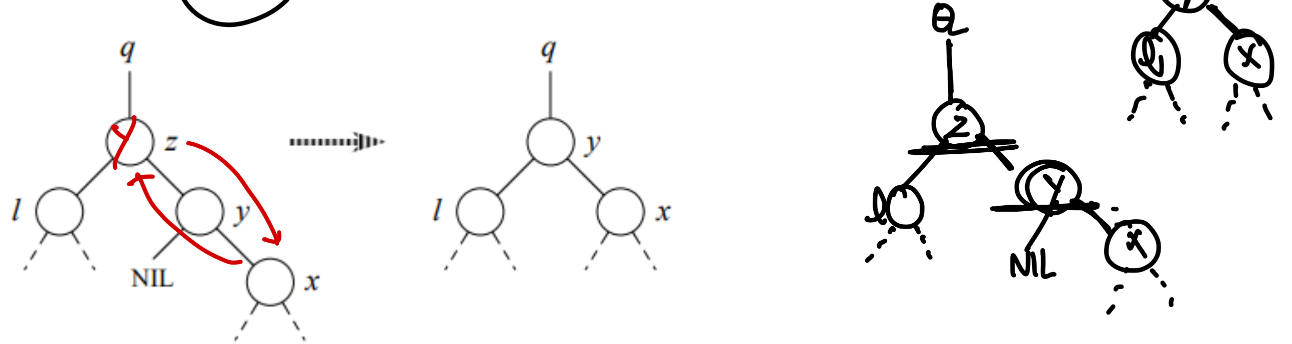
$\boxed{16}$

16의 위치에 13 or 20을 복사하고



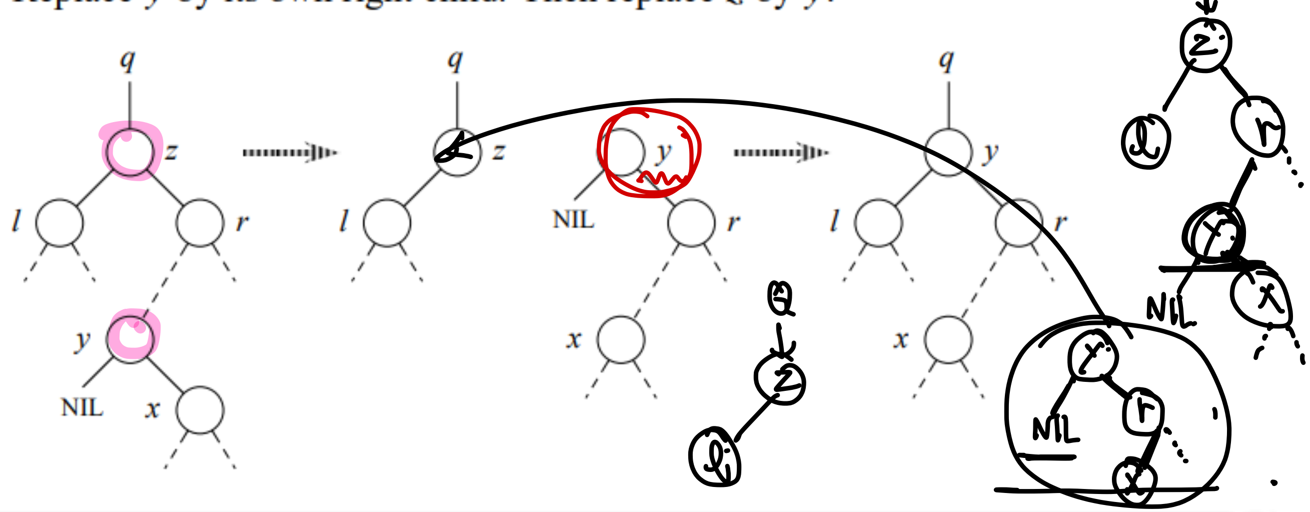
Deletion

- **Case3: 자식노드가 두개인 경우**
  - If y is z's right child, replace z by y and leave y's right child alone.



2 경우 처리

  - Otherwise, y lies within z's right subtree but is not the root of this subtree. Replace y by its own right child. Then replace z by y.



Deletion-pseudocode

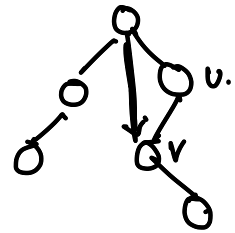
- 서브 트리를 이동시키기 위해 한 서브트리를 다른 서브트리로 교체하는 서브루틴 Transplant()
- U와 v가 root인 두 서브트리를 교체
- .p는 부모노드 의미
- 1~2행: u가 T의 루트인지 체크
- 3~4행: u가 왼쪽자식인 경우 왼쪽부분 갱신
- 5행: 오른쪽 자식인 경우
- 6~7행: v가 nil이 아닐 경우 갱신

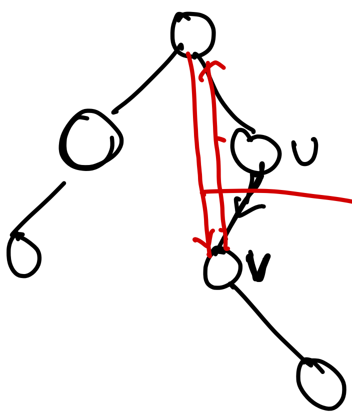
```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

root check

u가 왼쪽 자식인 경우

v가 nil일 수도 있음,



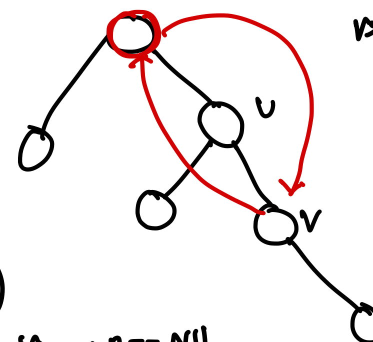


u.p.left = v.

u.p.right = V

t v.p = u.p

T,u,V



if u.p == NIL.
T.root = v.

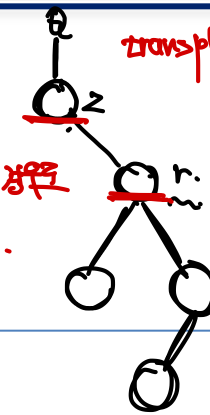
else if u == u.p.left
u.p.left = V

else u.p.right = v

if v != NIL
v.p = u.p

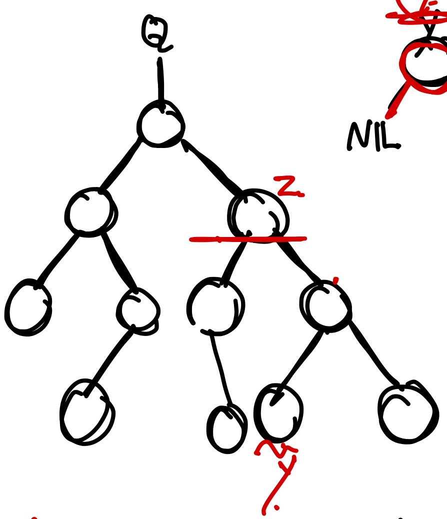
Deletion-pseudocode

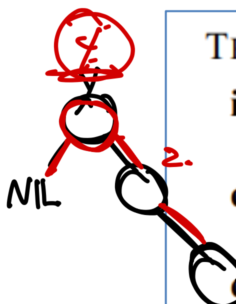
- 1~2행: z가 왼쪽자식이 없는 경우
- 3~4행: z가 오른쪽 자식이 없는 경우
- 5~12행: z가 두 자식 노드를 갖는 경우
- 5행의 Tree-Minumum은 z의 직후원소를 찾는다



transplant (z, z.right) 하면 z의 서브트리를 z.right의 서브 트리로.

자식이 없는 경우는 위 두 경우로 활용이 가능하다.






```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

y.left = z.left

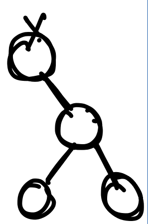
직후 원소.

직전 원소로 해도 ok.

y와 y.right를 이식

y.right = z.right

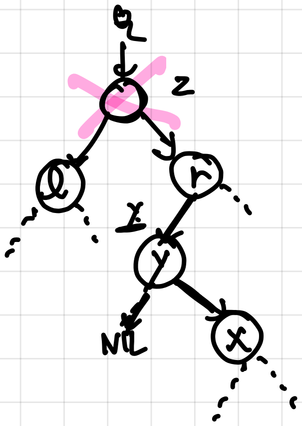
y.right.p = y

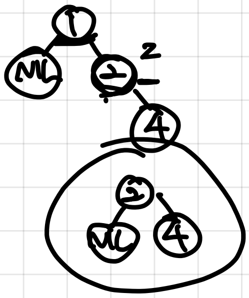


기본적인 경우

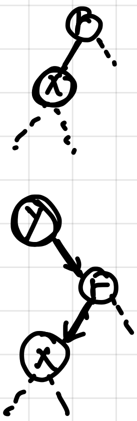
z.left.p = y

y.left = z.left





$y.p \neq z$

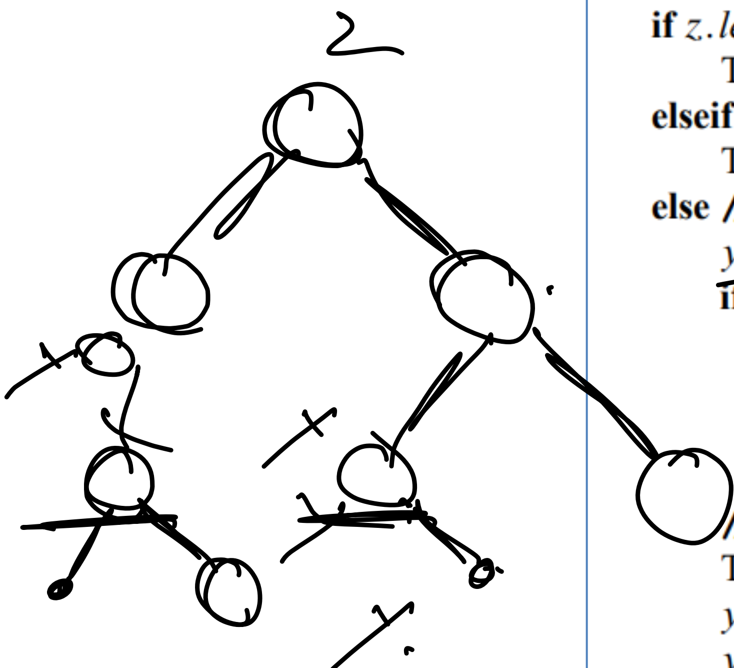


Tree delete (T, z)

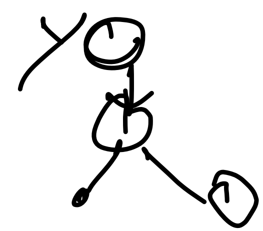
if z.left == NIL
    Transplant (T, z, z.right)
elif z.right == NIL
    transplant (T, z, z.right)
else # two-child
    y = tree-min (z.right)
    if y.p ≠ z
        transplant (T, y, y.right)
        y.right = z.right
        y.right.p = y
    transplant (T, z, y)
    y.left = z.left
    y.left.p = y

Deletion-pseudocode

- **Time**
  - Transplant: 상수
  - 각행 또한 상수시간
  - 트리의 높이만큼 O(h)가 덜린다

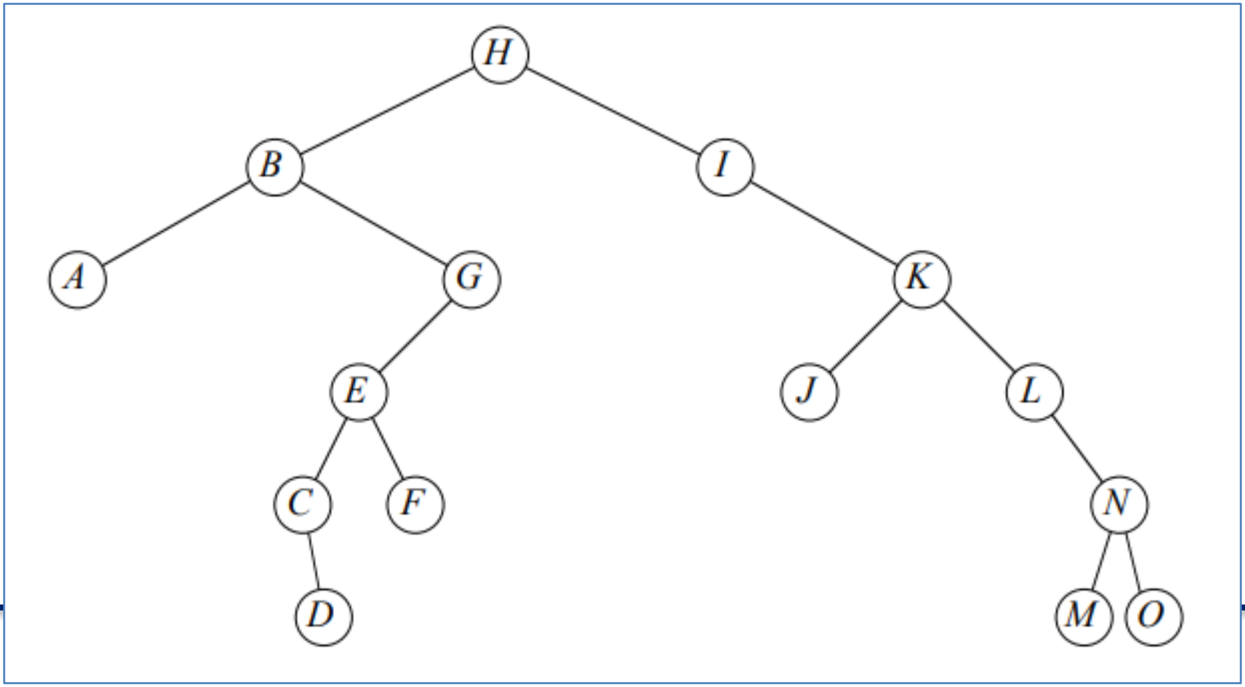


```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```



Deletion-pseudocode

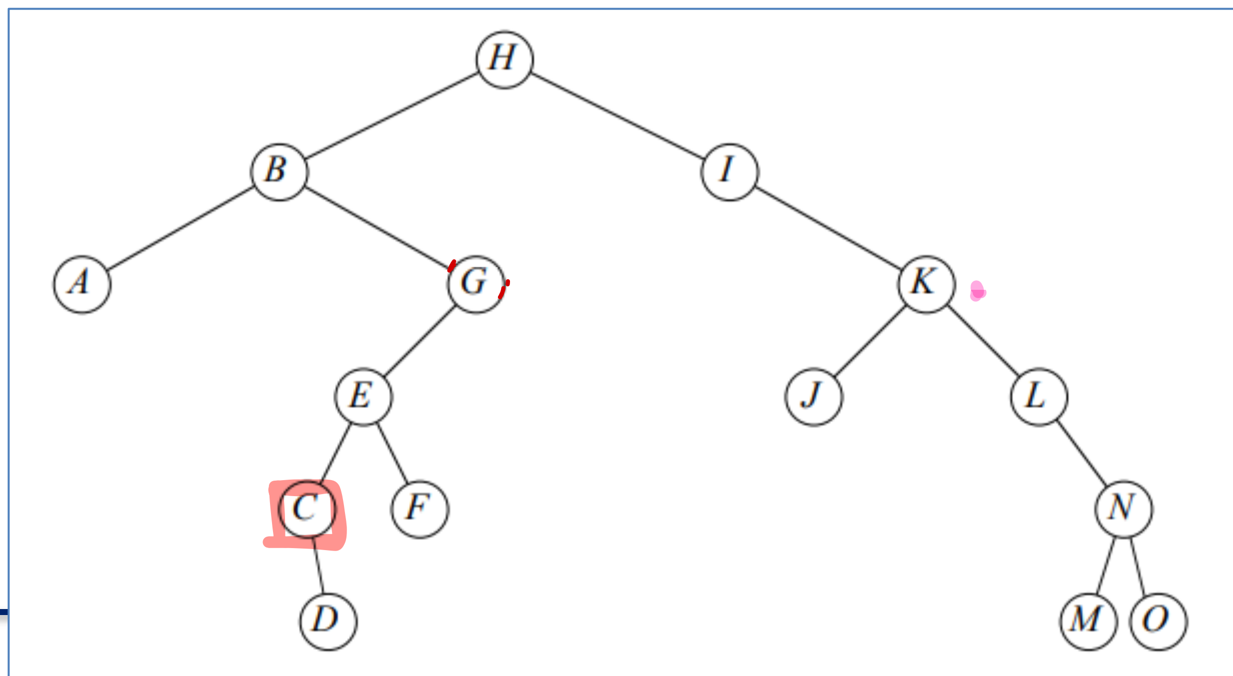
- **Example)**
  - TREE-DELETE(T, I) shows the case in which the node deleted has no left child.
  - TREE-DELETE(T, G) shows the case in which the node deleted has a left child but no right child.
  - TREE-DELETE(T, K) shows the case in which the node deleted has both children and its successor is its right child.
  - TREE-DELETE(T, B) shows the case in which the node deleted has both children and its successor is not its right child.



Binary search tree

- 이진 검색 트리의 한 노드가 두 자식을 가지면, 이 노드의 직후원소는 왼쪽자식을 갖지않고 직전원소는 오른쪽 자식을 갖지 않음을 보여라

1 2 4 5 6 7



Binary search tree –exercise

- 이진 검색 트리의 한 노드가 두 자식을 가지면, 이 노드의 직후원소는 왼쪽자식을 갖지않고 직전원소는 오른쪽 자식을 갖지 않음을 보여라
  - x를 두 개의 자식이 있는 노드라고 가정.
  - 중위 트리 워크에서 x의 왼쪽 하위 트리에 있는 노드는 x 바로 앞에 있고 x의 오른쪽 하위 트리에 있는 노드는 x 바로 뒤에 있음.
  - 따라서 x의 predecessor 은 왼쪽 하위 트리에 있고 successor은 오른쪽 하위 트리에 있음
  - s가 x의 successor라고 가정.
  - 그러면 s는 왼쪽 자식을 가질 수 없다.
  - s의 왼쪽 자식은 inorder walk에서 x와 s 사이에 오기 때문
  - inorder walk에서 x와 s 사이에 노드가 있으면 s는 우리가 생각한 것처럼 x의 후속 노드가 아니다

Binary search tree –exercise

- **N개의 숫자가 주어졌을때, 이를 모두 포함하는 이진검색트리를 만들고 여기에 중위트리순회를 출력하여 정렬 가능하다.**
- **이 알고리즘의 최악 및 최적의 수행시간은?**

- ***Tree-Sort(A):***

  For i=1 to n

  Tree-Insert(T, A[i])

  Inorder-tree-walk(T.root)

  $\Theta(n)$

- **최악의 케이스: O(n^2), 한쪽으로 비대칭적으로 나오는 linear chain 형태의 노드가 반복될시**
- **최선의 케이스 : O(nlgn), 이진트리의 높이가 O(lgn)일때**

Binary search tree –exercise

- 이진 검색트리의 노드 개수를 세는 함수?
- 이진 검색트리에서 높이를 계산하는 함수?

Binary search tree –exercise

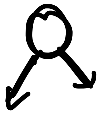
- 이진 검색트리의 노드 개수를 세는 함수?
  - ➢ *def count_node(root):*

```
If root == None:
        Return 0
Else:
        Return 1+count_node(root.left)+count_nodes(root.right)
```

tree-dp

- 이진 검색트리에서 높이를 계산하는 함수?
  - ➢ *def height_count(root):*

```
If root == None:
        Return 0
Else:
        left_height = height_count(root.left)
        right_height = height_count(root.right)
        Return max(left_height, right_height)
```



+1

Binary search tree –exercise

- 데이터베이스 시스템
  - ➢ 데이터베이스에서는 검색 속도가 매우 중요합니다. 이진 검색트리는 데이터가 정렬되어 저장되기 때문에, 데이터베이스에서 빠른 검색 속도를 제공하는 데 활용됩니다.
- 검색 엔진
  - ➢ 인터넷 검색 엔진에서도 이진 검색트리가 활용됩니다. 검색 엔진에서는 매우 큰 데이터 세트에서 검색을 수행해야 하므로, 빠른 검색 속도가 매우 중요합니다.
- 알고리즘
  - ➢ 이진 검색트리는 다양한 알고리즘에서 활용됩니다. 예를 들어, 다익스트라 알고리즘에서는 이진 검색트리를 활용하여 최단 경로를 계산합니다.
- 파일 시스템
  - ➢ 파일 시스템에서도 이진 검색트리가 활용됩니다. 파일 시스템은 파일이나 디렉토리를 검색하고, 삽입하고, 삭제하는 데 이진 검색트리를 활용합니다.
- 네트워크 라우팅
  - ➢ 네트워크에서 데이터를 전송할 때는 최적 경로를 계산해야 합니다. 이진 검색트리는 네트워크 라우팅에서 최적 경로를 계산하는 데 활용됩니다.