CS590 - Algorithms

Week 6 – Binary Search Trees

Fall 23

Outline

- 6. Binary Search Trees (BST)
- 6.1. Binary Search Trees
- 6.2. BST – In order tree walk
- 6.3. Querying a BST
- 6.4. BST Insertion
- 6.5. BST Deletion

6.1. Binary Search Tree

- **Binary search trees (BSTs)** are an important data structure for dynamic sets.
- They accomplish many dynamic set operations in O(h) time, where h is the tree's height.
- We represent a **binary tree** by a linked data structure in which each node is an object.

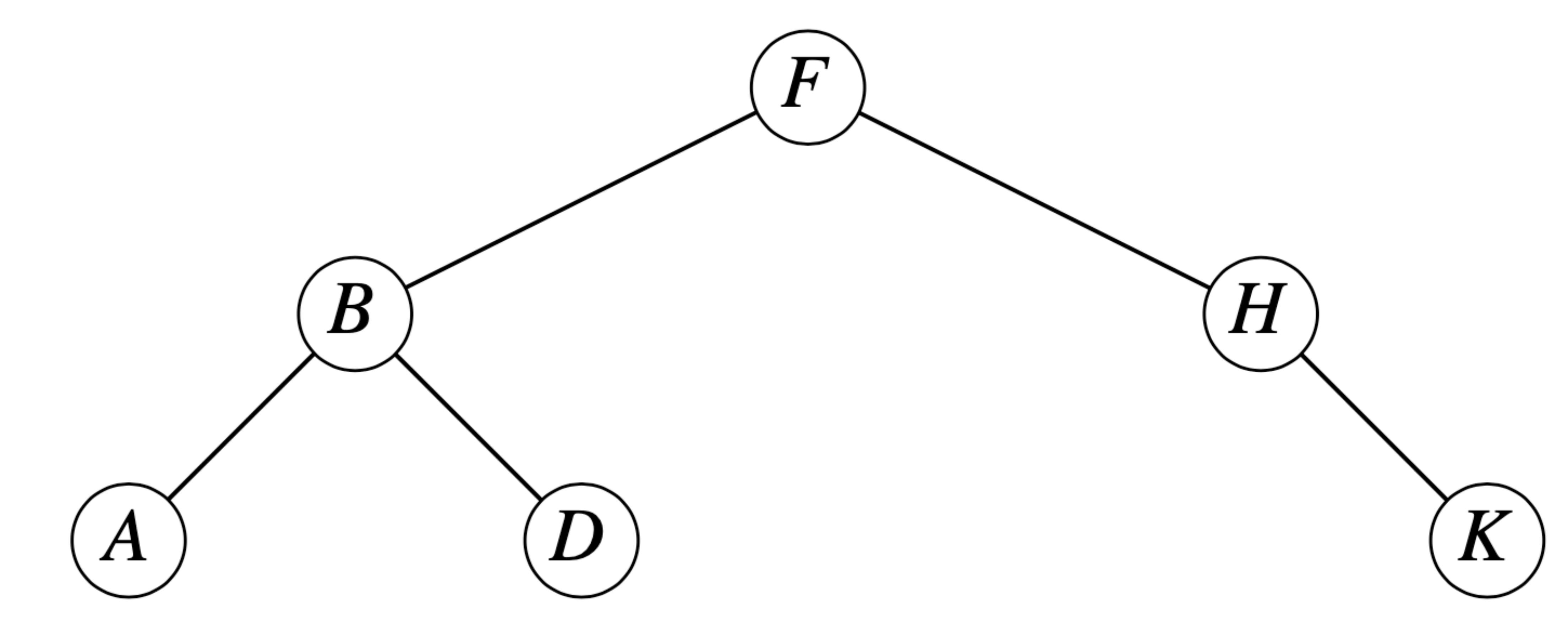
6.1. Binary Search Tree

- *T.root* points to the root of the tree, *T*.
- Each node contains the fields
  - *Key*: (and possibly other satellite data).
  - *left*: points to the left child.
  - *right*: points to the right child.
  - *p*: points to the parent
    - $T.root.p = \text{NIL}$.

6.1. Binary Search Tree

- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in the left subtree of x, then $y.key \leq x.key$.
  - If y is in the right subtree of x, then $y.key \geq x.key$.

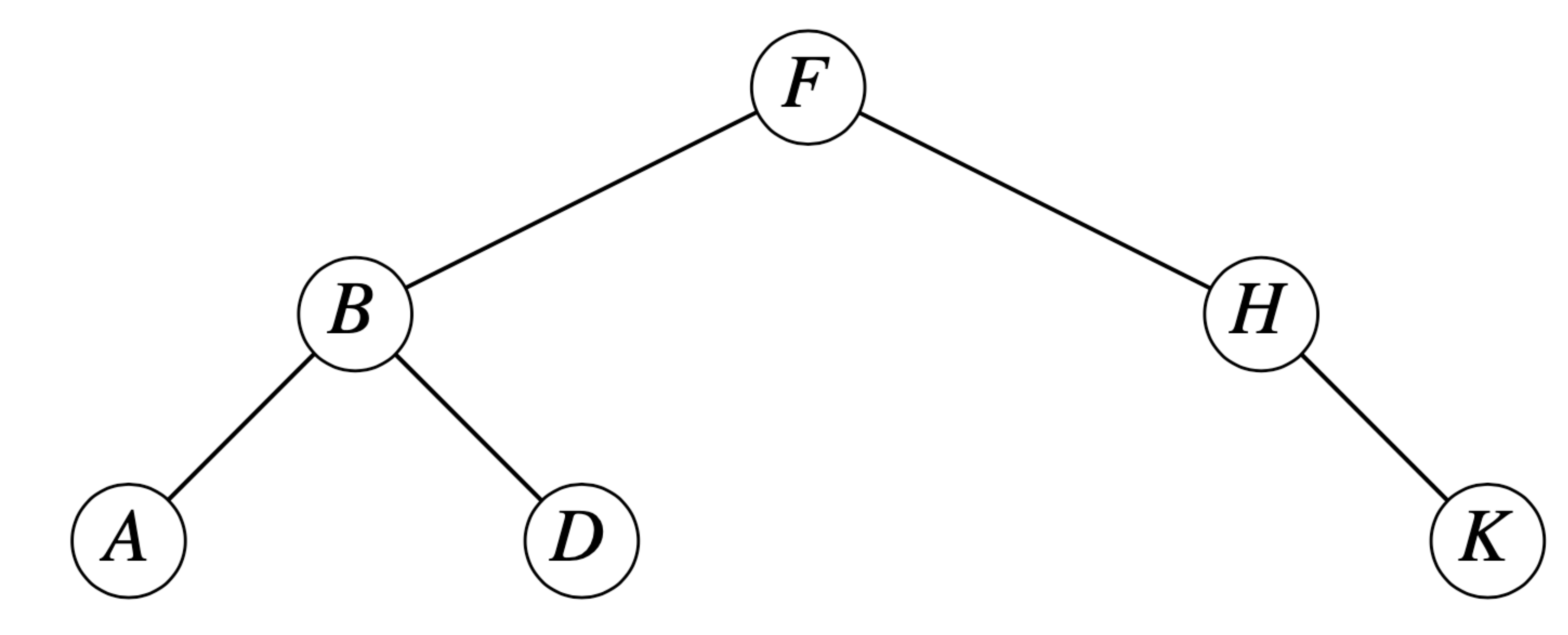
6.1. Binary Search Tree



6.2. BST – Inorder-Tree-Walk

```
INORDER-TREE-WALK(x
    (1)  if  (x!=NIL) then
    (2)      INORDER-TREE-WALK(x.left)
    (3)      print x.key
    (4)      INORDER-TREE-WALK(x.right)
    (5)  fi
```

6.2. BST – Inorder-Tree-Walk



6.2. BST – Inorder-Tree-Walk

- What is the printout for **`INORDER-TREE-WALK(T.F)`**?

- Confirm the correctness.

- What is the running time T(n)?

6.2. BST – Inorder-Tree-Walk

- Construct the recursion equation.
  - Let $T(k)$ be the running time of INORDER-TREE-WALK(k) at any subtree with a k root, assuming there are j many nodes.
  - If the subtree is empty, then $T(k)=0$.
  - Suppose there are x many nodes in the left subtree of T.k.
    - The number of right subtree nodes is

6.2. BST – Inorder-Tree-Walk

- The recursion equation becomes:

6.2. BST – Inorder-Tree-Walk

- Use the substitution method to solve T(k).
- Let the guessing function be:

6.3. Querying a BST

- Consider searching for a key value k in the sorted BST.

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

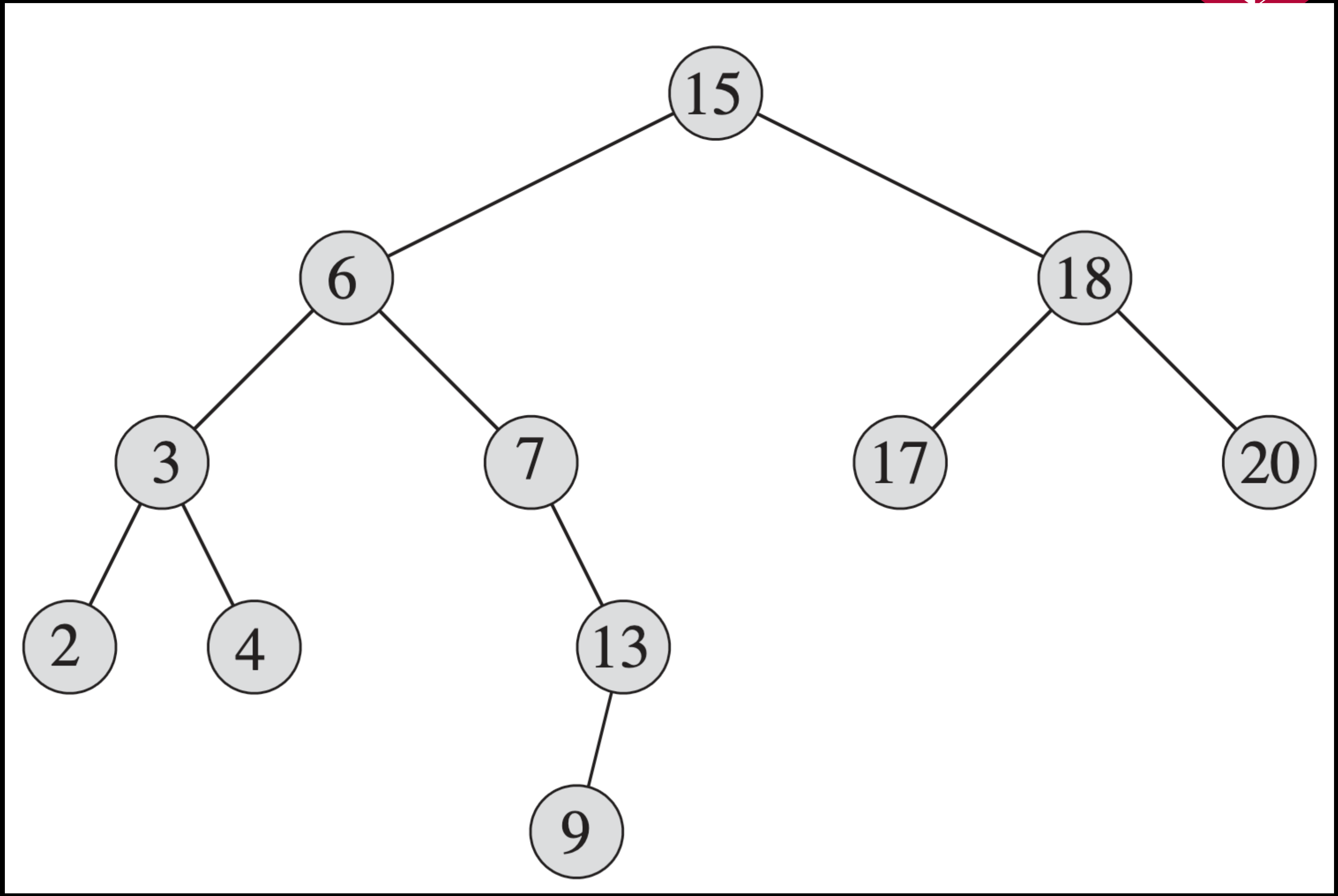
6.3. Querying a BST

- We can use an iterative approach:

```
ITERATIVE-TREE-SEARCH(x,k)
    (1) while x!=NIL and k!=x.key
    (2)   if k<x.key
    (3)       x = x.left
    (4)   else x = x.right
    (5) return x
```

6.3. Querying a BST

TREE-SEARCH(15,13)



6.3. Querying a BST

- How can we search for a minimum or maximum key value in BST?
- Where are they located in BST?
- What would be the time if you started from the root?
- Build pseudo-codes.

6.3. Querying a BST

- The minimum of BST is always at the left-most leaf.
- The maximum of BST is always at the right-most leaf.

```
TREE-MINIMUM(x)
 (1) while x.left!=NIL do
 (2)   x = x.left
 (3) return x
```

- TREE-MAXIMUM(x) is the same except from left to right.

6.3. Querying a BST

- Suppose all keys are distinct in BST.
- The **successor** of a node x is the node y such that y.key is the smallest key but > x.key.
- The **predecessor** of a node x is the node y such that y.key is the largest key but < x.key.

6.3. Querying a BST

- Finding a successor/predecessor is based on the tree structure.
- No key comparison is required.
- What if x.key is a minimum or maximum of the tree?
- What if x.key is not a minimum or maximum?

6.3. Querying a BST

- Consider two cases in a successor search:
  - x has a right subtree.
  - x does not have a right subtree.
    - We need to move left up until we find a smaller key.

6.3. Querying a BST

- TREE-SUCCESSOR(x):

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

6.3. Querying a BST

- Consider two cases in a predecessor search:
  - x has a left subtree.
  - x does not have a left subtree.
    - We need to move right up until we find a bigger key.

6.3. Querying a BST

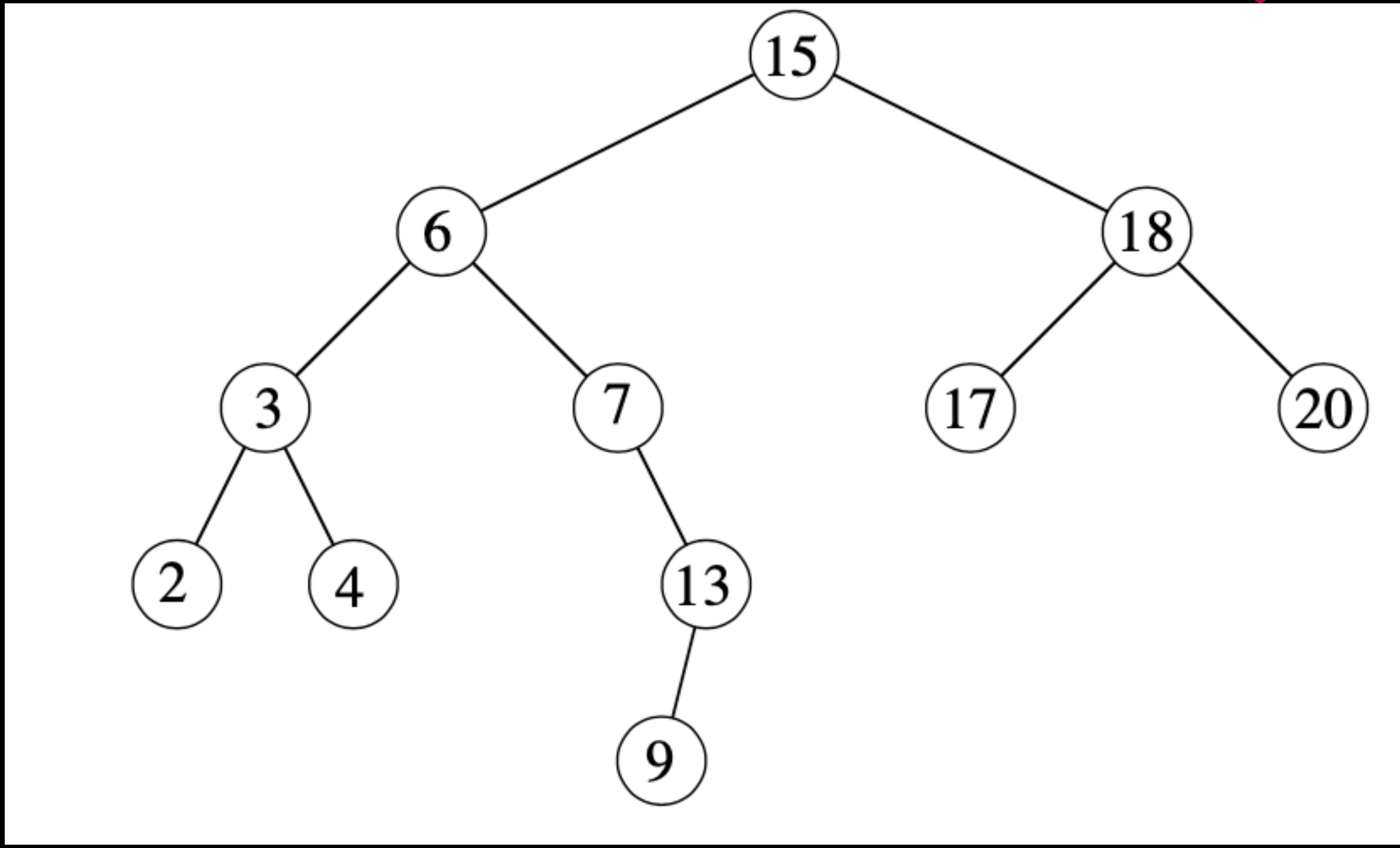
```
TREE-PREDECESSOR(x)
if x.left != NIL then
    return TREE-MAXIMUM(x.left)
y = x.p
while (y !=NIL and  x = y.left) do
    x = y
    y = y.p
return y
```

6.3. Querying a BST

- If y is the successor of x, then x is the predecessor of y.
  - x is the maximum in the left subtree of y.

6.3. Querying a BST

- Find the successor of 15.
- Find the successor of 6.
- Find the successor of 4.
- Find the predecessor of 6.



6.4. BST Insertion

- Consider inserting a node, z, to BST.
- BST property must be held after the insertion.
- Suppose z.key = v.
  - We let z.left = z.right = NIL.
  - Having children NIL makes connecting subtrees to z easier.

6.4. BST Insertion

- Two operations are needed.

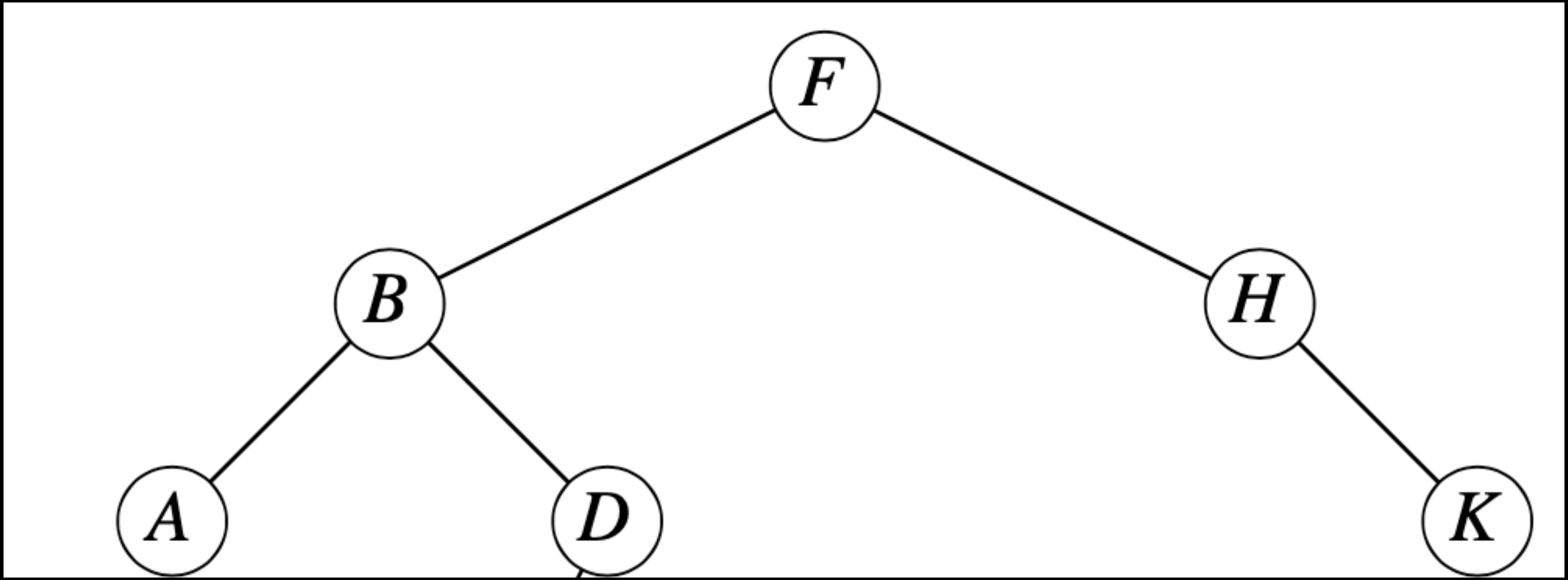
1. Find the position of z from the root using two pointers.
   - A pointer x will trace the path.
   - A pointer y will keep track of x.p.
   - x.key will be compared with z.key and move to the correct direction.
2. Insert z as either y.left or y.right accordingly.

6.4. BST Insertion

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

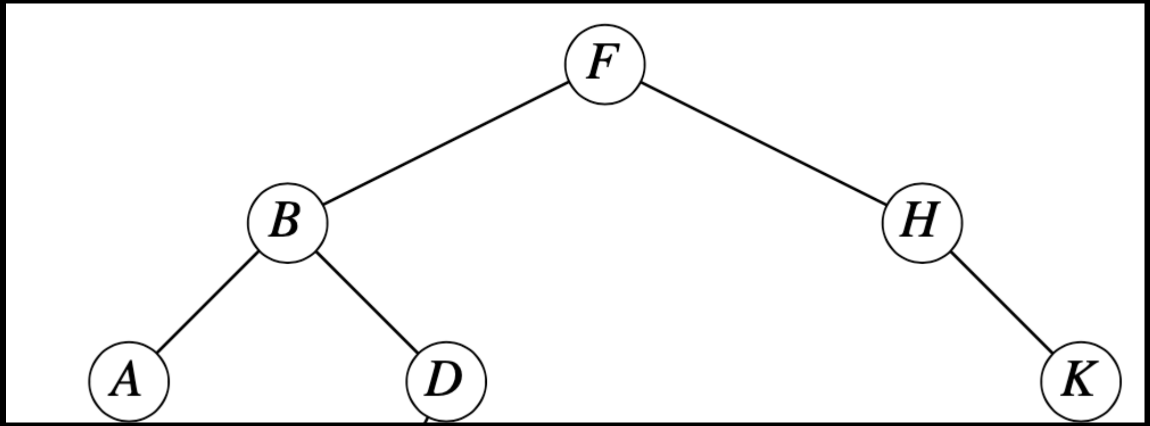
6.4. BST Insertion

TREE-INSERT (T, C)



6.4. BST Insertion

```
TREE-INSERT(T, z)
(1) y = NIL, x = T.root
(2) while (x != NIL) do
(3)     y = x
(4)     if (z.key < x.key) then
(5)         x = x.left
(6)         else x = x.right
(7)  z.p = y
```



6.4. BST Insertion

- Can we sort a set of given numbers using TREE-INSERT( )?

6.5. BST Deletion

- Deleting a node is much more complex than inserting a node.
- Suppose a node z is to be removed from BST.
  - After z is removed, the BST property must be maintained.
  - It is simple if z does not have a child or has a child.
    - z can be removed and z.p.child = NIL.
    - Since z.child is a root of its own subtree, let z.child.p = z.p after removing z.

6.5. BST Deletion

- But… if z has two children, there might be a problem.
    - Suppose we let z.right.p = z.p after removing z.
        - All elements in the z.left subtree are less than z.right.
        - z.left subtree can be a new left subtree of z.right if z.left is NIL.
        - What if z.left is not NIL?
            - Should we add the z.left subtree to the bottom of the z.right left subtree using TREE-INSERT(…)?
            - How about the height of the new BST?

6.5. BST Deletion

- What if we replace z with the successor of z, y?
  - y will be in the right subtree of z with no left child.
  - The original subtree of z can be a new right subtree of y.
  - The left subtree of z becomes a new left subtree of y.
  - Then, the BST property will always be maintained, and the height will be the same.

6.5 BST Deletion

- Before we move on, we will define a method called TRANSPLANT(T, u, v) that replaces one subtree, u, as the child of its parent by another subtree, v.

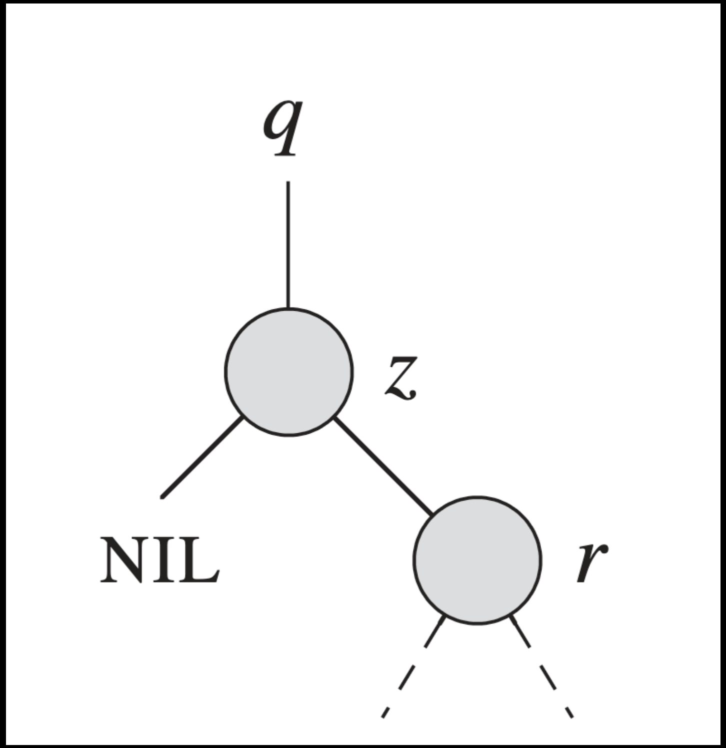
```
TRANSPLANT (T, u, v )
    if (u.p = NIL) then
        T.root = v
    else
        if (u = u.p.left) then
            u.p.left = v
        else u.p.right = v
        If (v ≠ NIL) then
            v.p = u.p
```

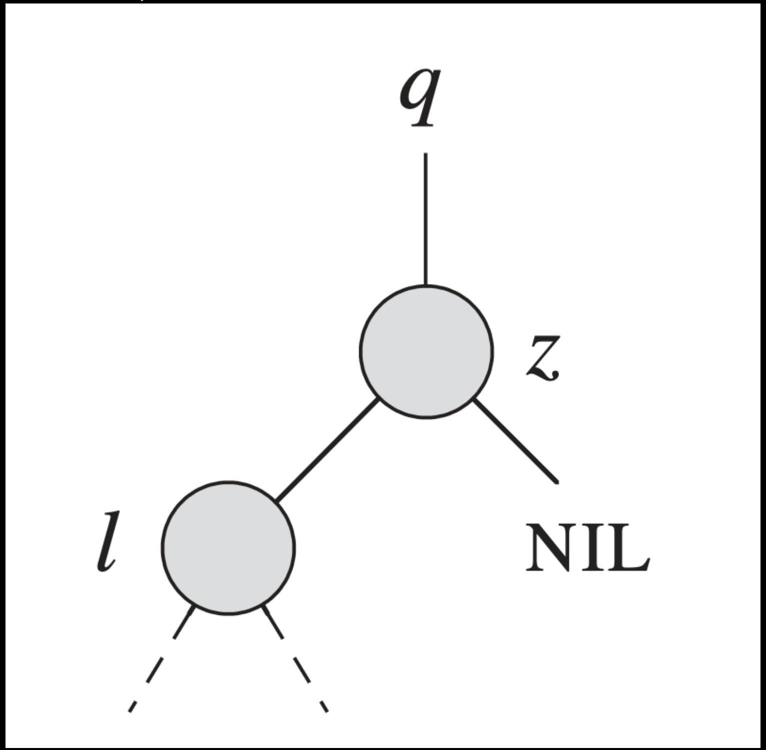
6.5. BST Deletion

- Case I: z does not have children.
  - Delete z by letting z.p point to NIL.
- Case 2: z has a single child.
  - Delete z by letting z.p point to the child.

6.5. BST Deletion

- To handle case 2, we can call TRANSPLANT(T, z, z.child).
- If z.left = NIL, call TRANSPLANT(T, z, r)
- If z.right = NIL, call TRANSPLANT(T, z, l)



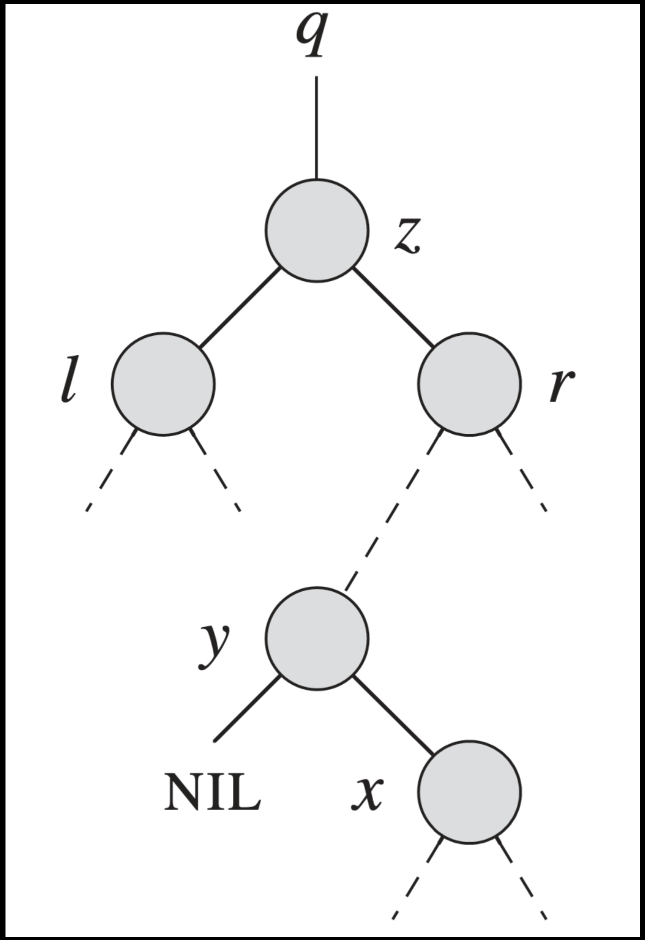


6.5. BST Deletion

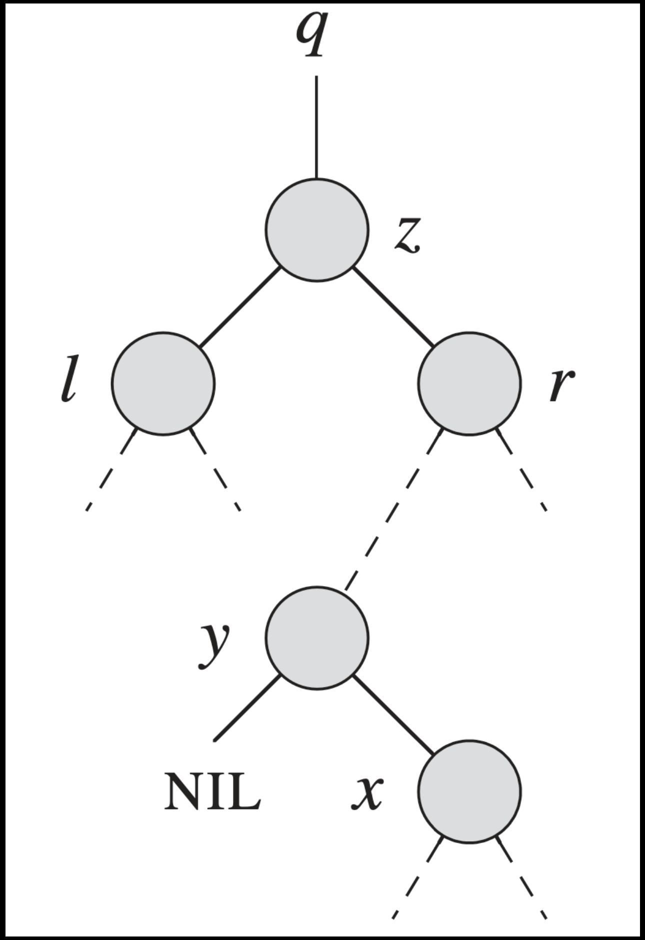
- Case 3: z has two children.
- Find a successor y by calling TREE-MINIMUM(z.right).
- Consider a two scenarios:
  - y != z.right (hard)
  - y = z.right (easy)

6.5. BST Deletion

- y != z.right (hard):
  - Call TRANSPLANT(T, y, y.right)
  - y.right = z.right
  - y.right.p = y

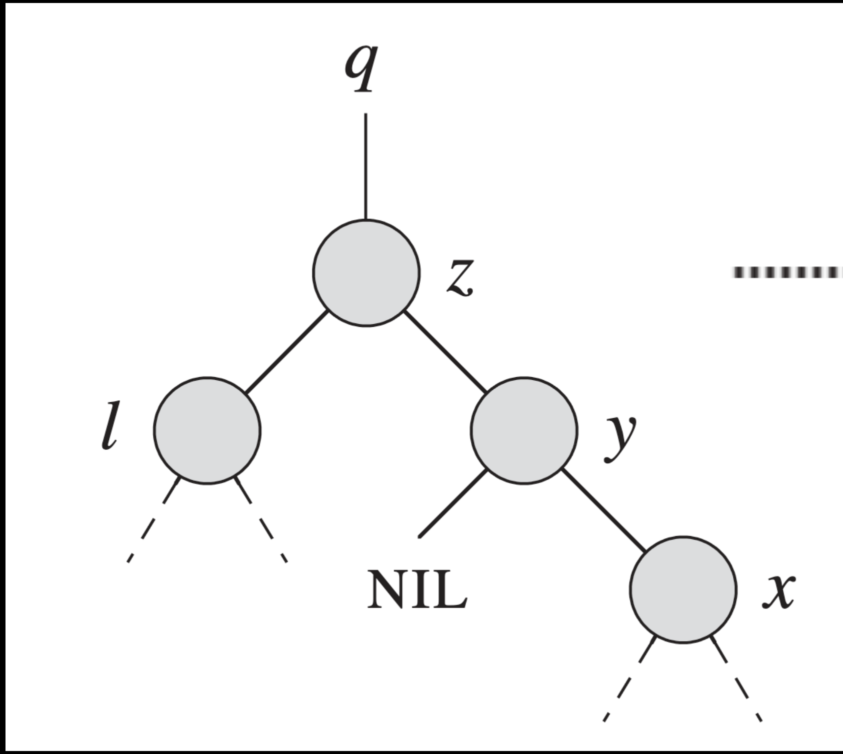


6.5. BST Deletion



6.5. BST Deletion

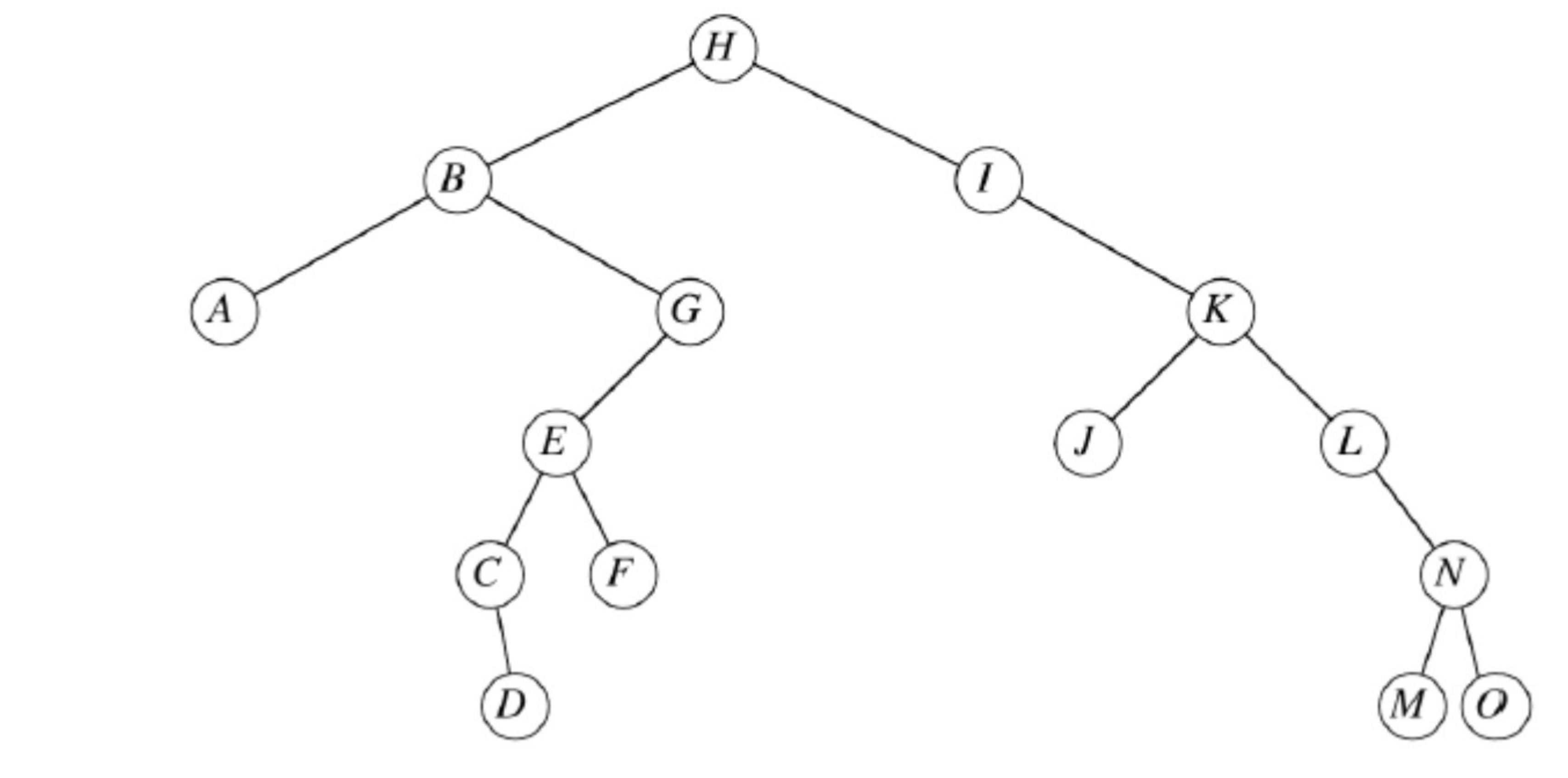
- y = z.right (easy):
  - Call TRANSPLANT(T, z, y)
  - y.left = z.left
  - y.left.p = y



```
TREE-DELETE(T, z)
  if (z.left = NIL) then                        //no left child
      TRANSPLANT(T, z, z.right)
  else
      if (z.right = NIL) then                   //no right child
          TRANSPLANT(T, z, z.left)
      else                                      //two children
          y = TREE_MINIMUM(z.right)
          if (y.p ≠ z) then                     //y is not z.right
              TRANSPLANT(T, y, y.right)
              y.right = z.right
              y.right.p = y
          TRANSPLANT(T, z, y)                   //y is z.right
          y.left = z.left
          y.left.p = y
```

6.5. BST Deletion

- Delete I
- Delete G
- Delete K
- Delete B



6.5. BST Deletion

- Consider analyzing in terms n, not h.
- Best case: when the tree is balanced.
- Worst case: when the tree is skewed. Ways to fix up?
- Need to reconstruct the BST.
- Red-black trees will do it.