CS590 - Algorithms

Lecture 6 – Binary Search Trees

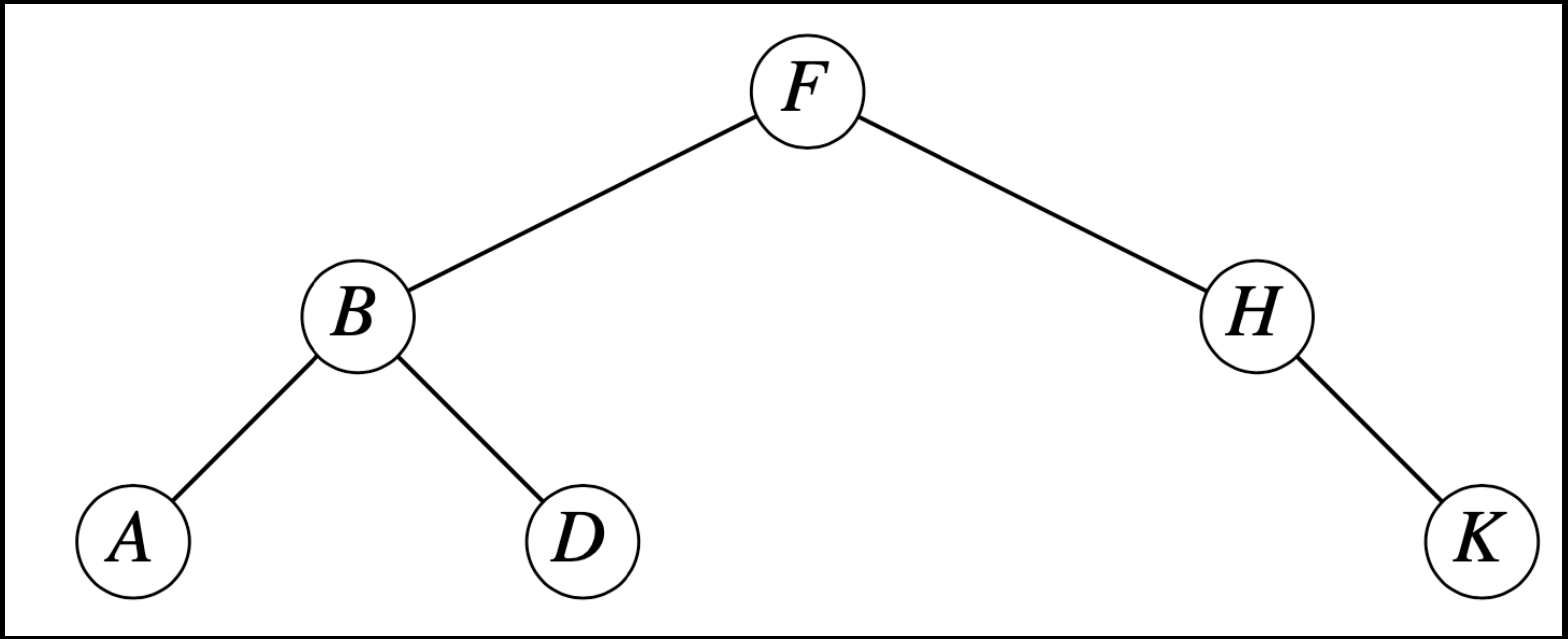
Close your laptop

6.1. Binary search trees

- **Binary search trees** (**BSTs**) are an important data structure for dynamic sets.
- They accomplish many dynamic set operations in $O(h)$ time, where h = height of tree.
- We represent a **binary tree** by a linked data structure in which each node is an object.
- $T.root$ points to the root of tree T .
- Each node contains the fields
  - *Key*: (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - *p*: points to parent. $\Rightarrow T.root.p$ = 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 $y.key \geq x.key$.

6.1. Binary search trees



The ***binary-search-tree property*** allows us to print keys in a binary search tree in order, recursively, using an algorithm called an ***inorder-tree-walk***.
Elements are printed in monotonically increasing order.

6.2. BST – In order tree walk

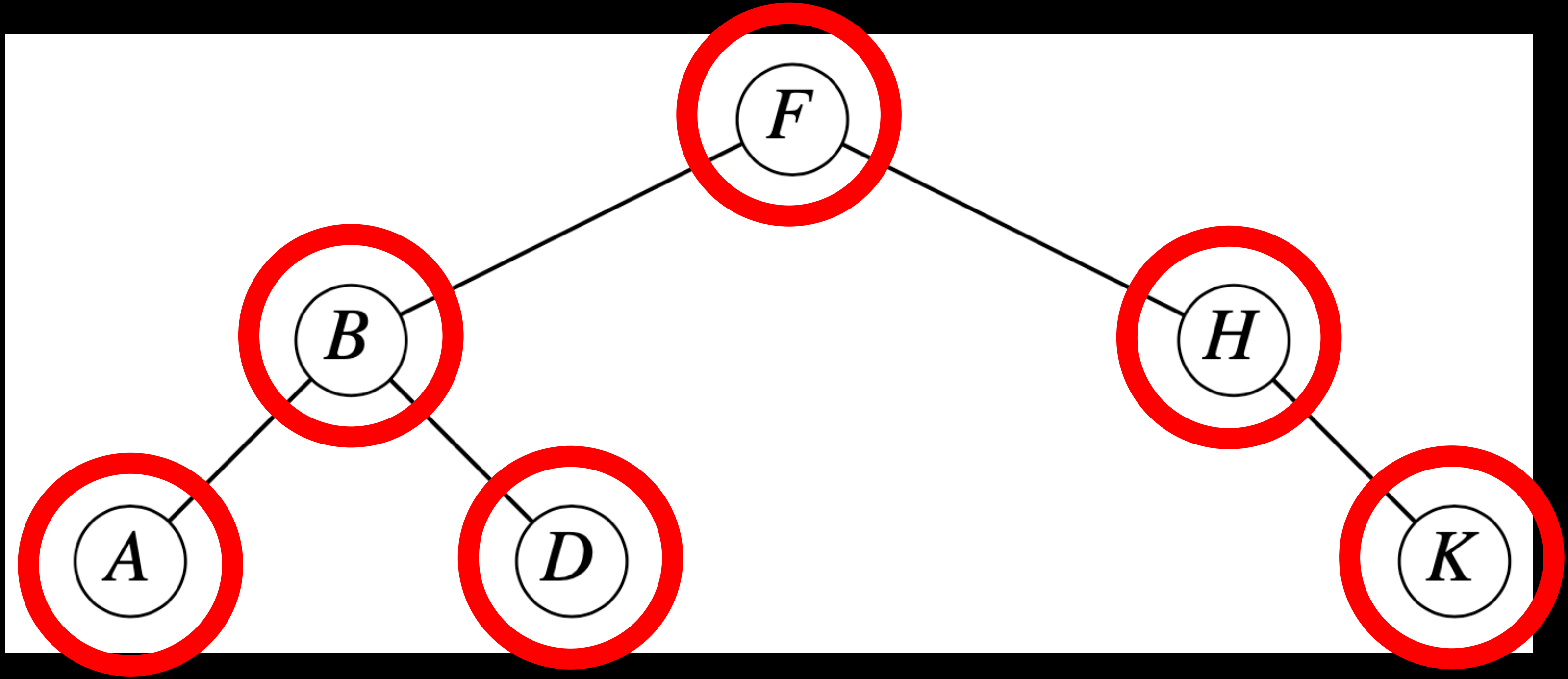
Algorithm (INORDER-TREE-WALK(x))

(1) **if** $(x \neq NIL)$ **then**
(2) INORDER-TREE-WALK(x.left)
(3) **print** x.key
(4) INORDER-TREE-WALK(x.right)
(5) **fi**

How INORDER-TREE-WALK works?

- 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.

6.2. BST – In order tree walk



BST prints: A, B, D, F, H, K

Correctness: Follows by induction directly from the binary-search-tree property.

Time: Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once.

6.2. BST – In order tree walk

Recursion Running Time Equation Construction:

- Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree.
- We have $T(n) = \Omega(n)$ since it visits all n nodes of the subtree.
- The amount of time on an empty subtree is $T(0) = c$ for $c > 0$.
- **Inductive Step**:
  - For $n > 0$, suppose that a node x on the left subtree has k nodes.
  - Then the right subtree has $n - k - 1$ nodes.
  - The performance time is bounded by $T(n) \leq T(k) + T(n - k - 1) + d$ for some constant $d > 0$.
- We need to show that $T(n) = O(n)$!

6.2. BST – In order tree walk

- $T(n) \leq T(k) + T(n-k-1) + d$ for some constant $d > 0$.
- Using the substitution method with guessing that $T(n) \leq (c+d)n + c$.
- For $n = 0$, we see that $T(0) = c$.
- For $n > 0$, we have

$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &\leq \big((c+d)k + c\big) + \big((c+d)(n-k-1) + c\big) + d \\ &= (c+d)n + c - (c-d) + c + d \\ &= (c+d)n + c \end{aligned}$$

6.3. Querying a BST

Searching

- Need to search for a key sorted in BST.
- BST can support queries such as MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR.

Algorithm (TREE-SEARCH(x,k))

(1) **if** $(x = NIL \mid k = x.key)$ **then**
(2)   **return** x
(3) **if** $(k < x.key)$ **then**
(4)   **return TREE-SEARCH**(x.left,k)
(5) **else**
(6)   **return TREE-SEARCH**(x.right,k)

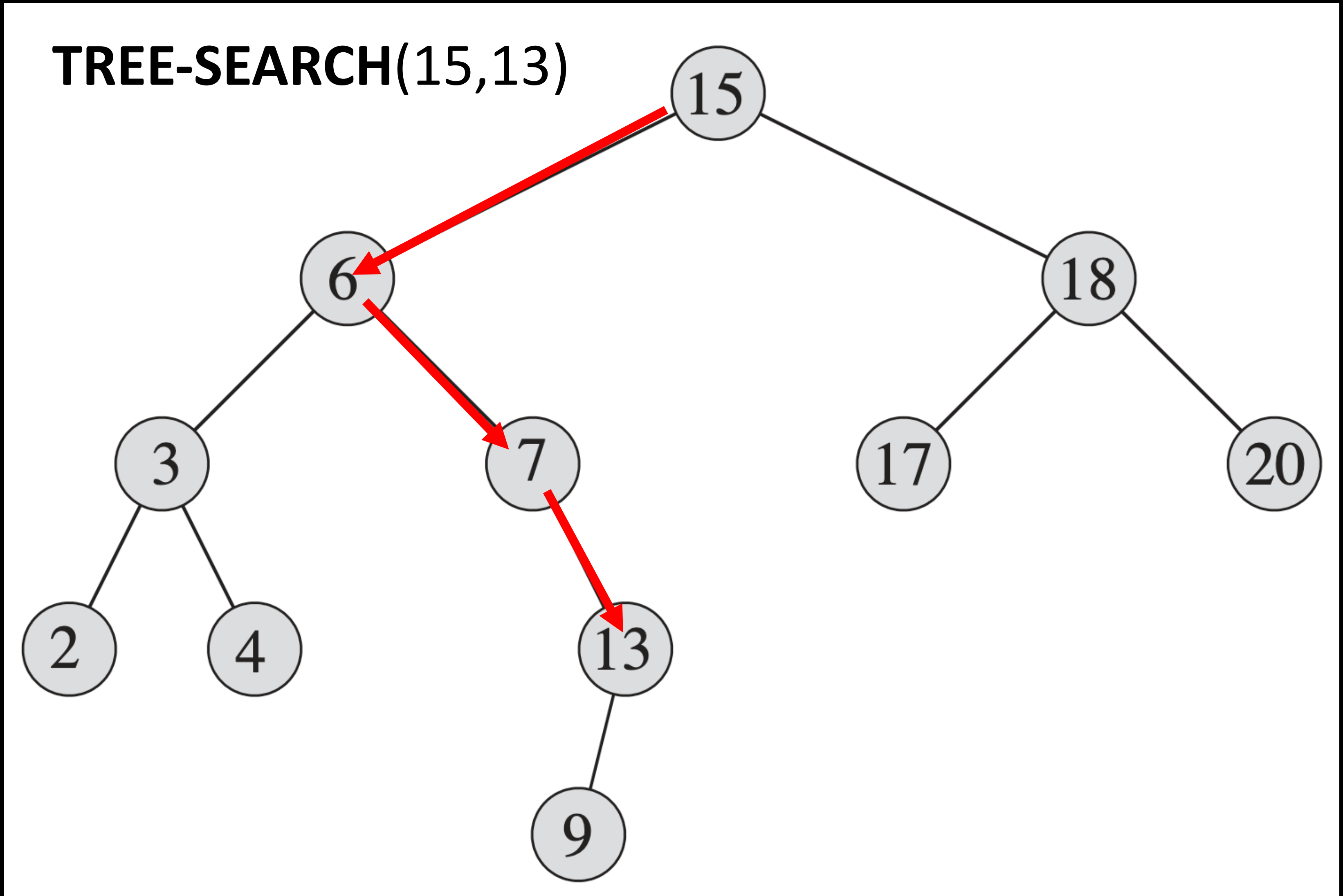
- Initial call is TREE-SEARCH(*T.root*, *k*).
- ***Time:*** The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

6.3. Querying a BST

Algorithm (**ITERATIVE-TREE-SEARCH**(x,k))

while x $\neq NIL \,\&\, k \neq x.key$
  if $k < x.key$
    x=x.left
  else x = x.right
return x

6.3. BST – Tree Search



6.3. Querying a BST

Minimum and maximum

Algorithm (TREE-MINIMUM(x))

(1) **while** $x.\text{left} \neq NIL$ **do**
(2) x=x.left
(3) **return** x

Algorithm (TREE-MAXIMUM(x))

(1) **while** $x.\text{right} \neq NIL$ **do**
(2) x=x.right
(3) **return** x

The BST property guarantees that

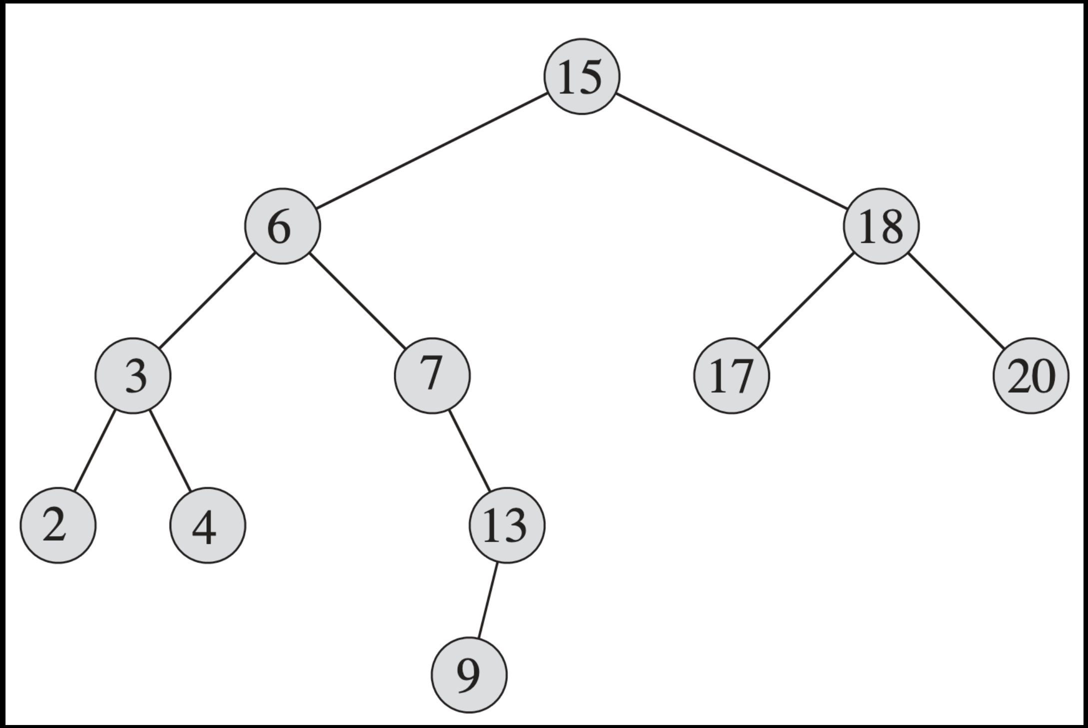
- The minimum key of a BST is located at the leftmost node.
- The maximum key of a BST is located at the rightmost node.

- ***Time:*** Both procedures visit nodes that form a downward path from the root to a leaf.
- Both procedures run in $O(h)$ time, where h is the height of the tree.
- Traverse the appropriate pointers (*left* or *right*) until NIL is reached.

6.3. Querying a BST
Successor and predecessor

Assuming that all keys are distinct,

- the **successor** of a node x is the node y such that $y.key$ is the smallest $key > x.key$.
- the **predecessor** of a node x is the node y such that $y.key$ is the largest $key < x.key$.



6.3. Querying a BST
Successor and predecessor

- We can find x's successor based entirely on the tree structure.
- No key comparisons are necessary.
- If x has the largest /smallest key in the binary search tree, then successor/predecessor is NIL.
- There are two cases:

  1. If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
  2. If node x has an empty right subtree, notice that:
     - As long as we move to the left up the tree (move up through right children), we're visiting smaller keys.
     - x's successor y is the node that x is the predecessor of y (x is the maximum in y's left subtree).

6.3. Querying a BST
Successor and predecessor

Algorithm (TREE-SUCCESSOR(x))

(1) **if** x. right $\neq NIL$ **then**
(2) return TREE-MINIMUM(x.right)
(3) y = x.p
(4) while (y ≠ *NIL* and x = y.right) do
(5) x = y
(6) y = y.p
(7) return y

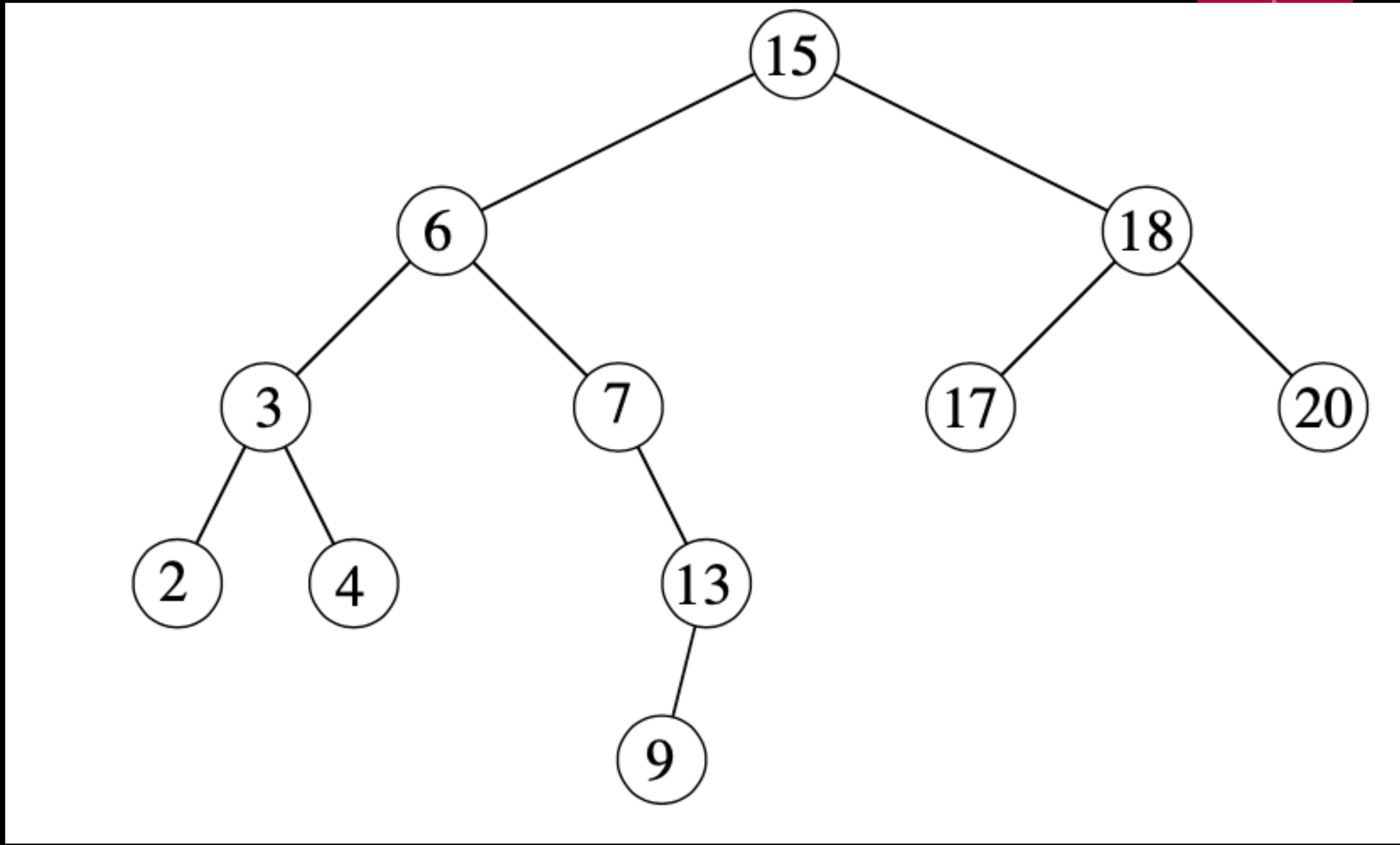
6.3. Querying a BST
Successor and predecessor

Algorithm (TREE-PREDECESSOR(x))

(1) **if** $\text{x.left} \neq NIL$ **then**
(2)   return TREE-MAXIMUM(x.left)
(3) y = x.p
(4) while (y $\neq$ *NIL* and x = y.left) do
(5)   x = y
(6)   y = y.p
(7) return y

- ***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.

6.3. Querying a BST
Successor and predecessor



- 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.

6.4. Insertion

- Insertion and deletion allows the dynamic set represented by a binary search tree to change.
- The binary-search-tree property must hold after the change.
- To insert value v into the binary search tree, we create a node z, with $z.key=v$, $z.left$=NIL, and $z.right$=NIL.
- Find the position for z by tracing downward path from the root. Two points must be maintained:
    - Pointer x: traces downward path.
    - Pointer y: “trailing pointer” to keep track of parent of x.
- Traverse downward by comparing x.key with v (or z.key) then move to the left or right child accordingly.
- The correct position for z if x=*NIL*.
- Compare z’s value with y’s value, and insert z at either y’s *left* or *right*, appropriately.

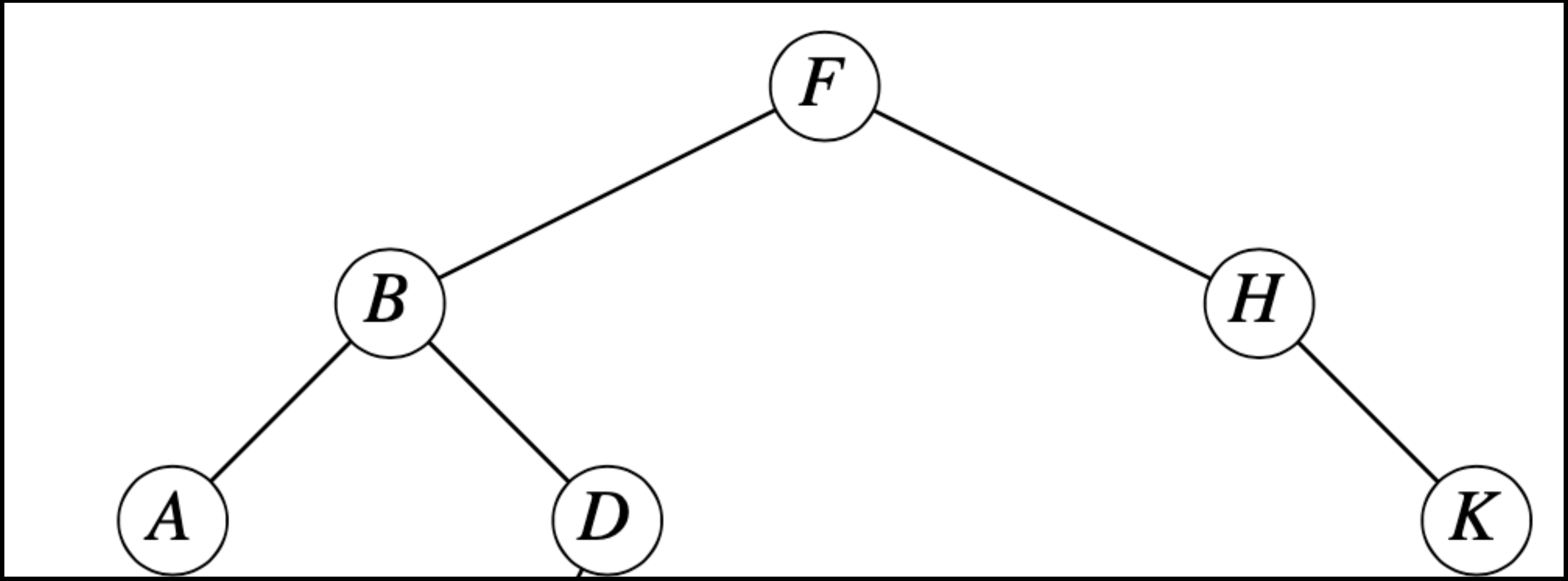
6.4. Insertion and Deletion

Algorithm (TREE-INSERT(T,x))

```
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
```

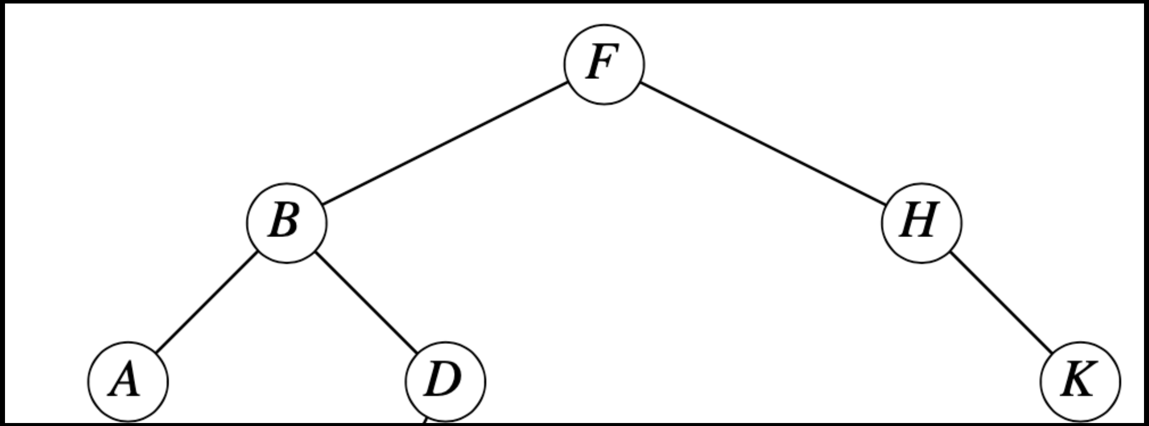
6.4. Insertion and Deletion

TREE-INSERT (T, C)



6.4. Insertion and Deletion

TREE-INSERT (T, C)



Start: y=NIL, x = F

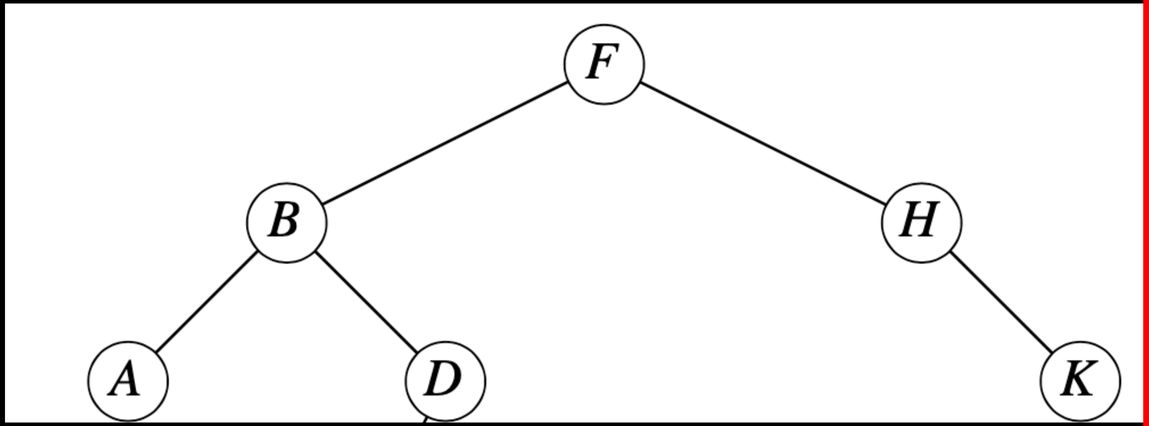
1. x = F, y = F
2. C < F: x = B
3. y = B, C < B, x = D
4. y = D, C < D, x = NIL
5. While-loop terminates: y=D, x = NIL

Algorithm (TREE-INSERT(T,x))

```
y = NIL, x = T.root
while (x ≠ NIL) do
  y = x
  if (z.key < x.key) then
    x = x.left
  else x = x.right
```

6.4. Insertion and Deletion

TREE-INSERT (T, C)



z.p = D

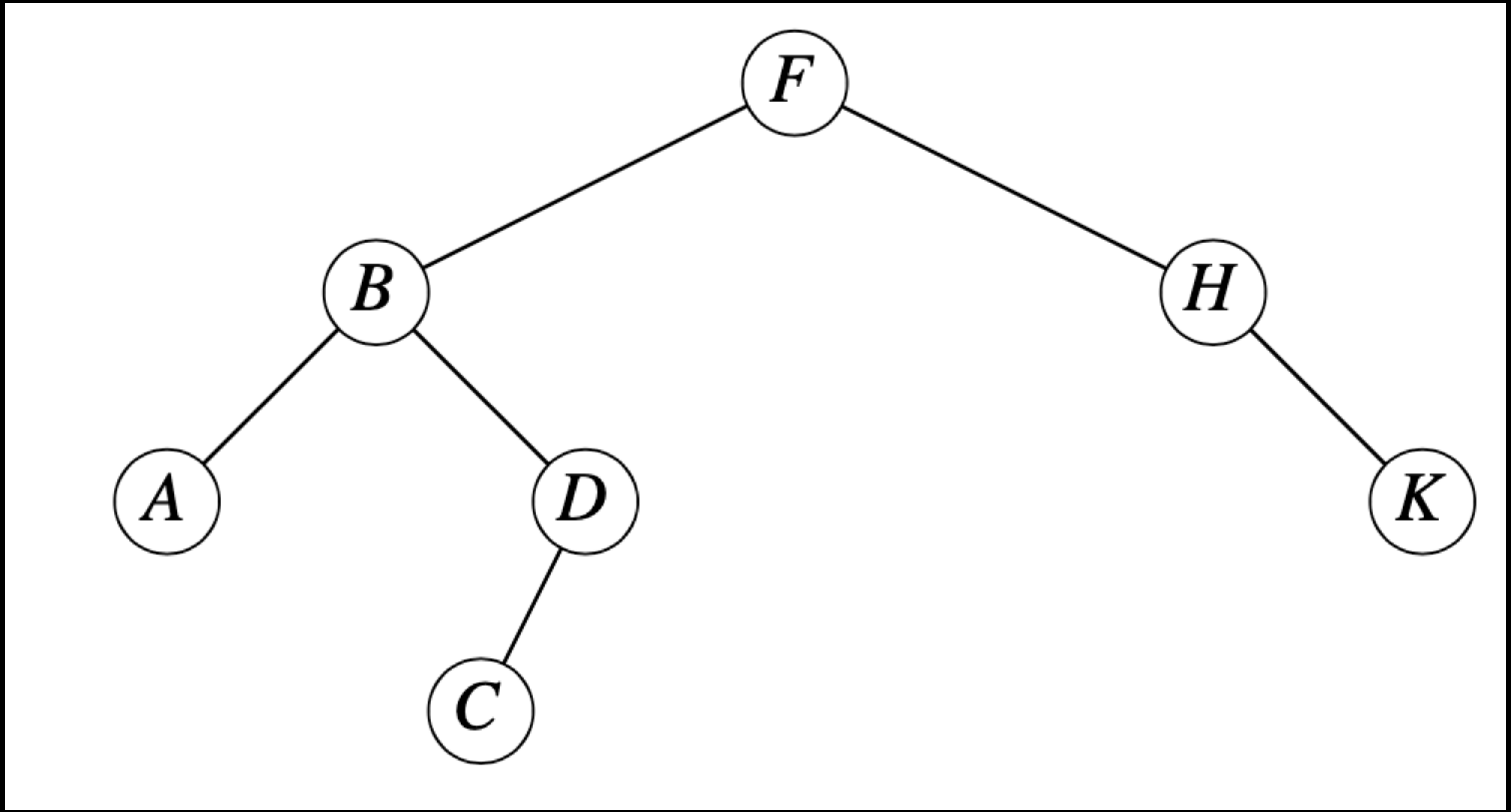
y =D, C < D, D.left = C

Algorithm (TREE-INSERT(T,x))

(1) … (6)
(7) z.p = y
(8) if (y = *NIL*) then
(9) T.root = z
(10) else if (z.key < y.key) then
(11) y.left = z
(12) else y.right = z

6.4. Insertion and Deletion

TREE-INSERT (T, C)



6.4. Insertion

- TREE-INSERT can be used with INORDER-TREE-WALK to sort a given set of numbers.

Algorithm (TREE-SORT(A))
(1) let T be an empty binary search tree
(2) **for** $i \leftarrow 1$ **to** n
(3) **do** TREE-INSERT(T, $A[i]$)
(4) INORDER-TREE-WALK($root[T]$)

- Worst case: $\Theta(n^2)$ occurs when a linear chain of nodes results from the repeated TREE-INSERT operations.
- Best case: $\Theta(n \lg n)$ occurs when a binary tree of height $\Theta(\lg n)$ results from the repeated TREE-INSERT operations.

6.5. Deletion

For deletion, consider three cases:

Case 1: z has no children.

- Delete z by making the parent of z point to NIL, instead of to z.

Case 2: z has one child.

- Delete z by making the parent of z point to z's child, instead of to z.

Case 3: z has two children.

- Find the successor of z, y, and replace z by the successor.
  - y must be in the right subtree of z and have no left child.
  - The rest of the original subtree of z becomes the new right subtree of y.
  - The left subtree of z becomes the new left subtree of y.

6.5. Deletion

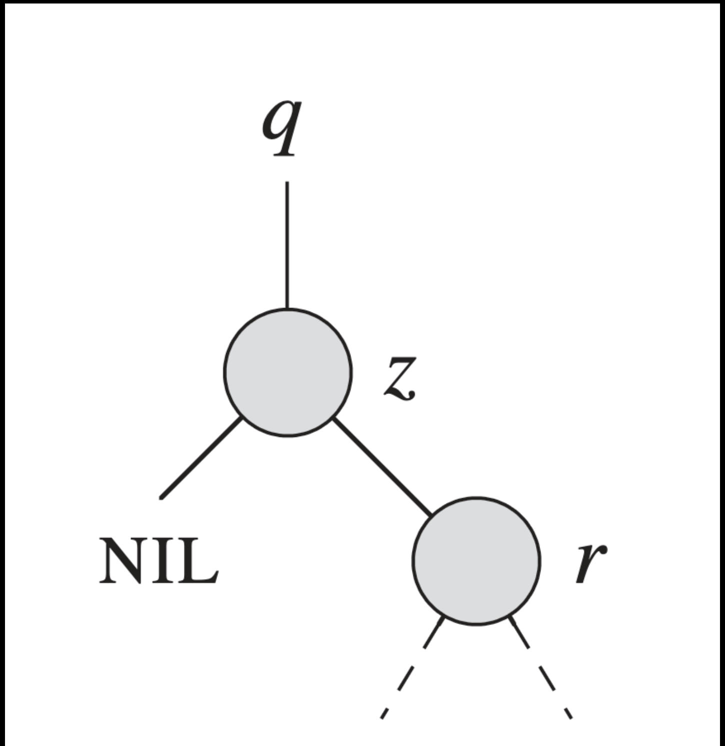
An algorithm **TRNSPLANT** replaces one subtree as the child of its parent by another subtree.

```
Algorithm (TRANSPLANT (T, u, v ))
(1)     if (u.p = NIL) then
(2)         T.root = v
(3)     else
(4)         if (u = u.p.left) then
(5)             u.p.left = v
(6)         else u.p.right = v
(7)         If (v ≠ NIL) then
(8)             v.p = u.p
```

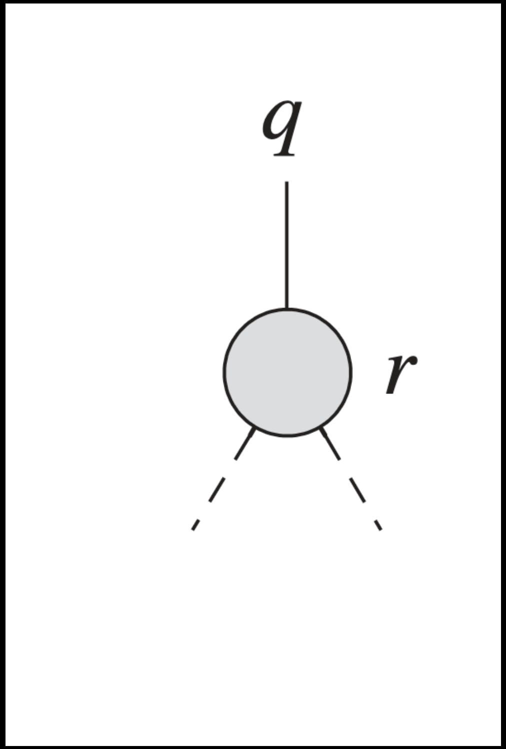
- Replaces the subtree rooted at u by the subtree rotted at v.
- Makes u.p becomes v
  - If u is the root, v becomes the root.
- u.p gets v as either u.left or u.right depending on whether u was u.p.left or u.p.right.
- Does not update v.left or v.right.

6.5. Deletion

- If *z* has no left child, replace by its right child. The right child may or may not be NIL (If NIL -> no child)

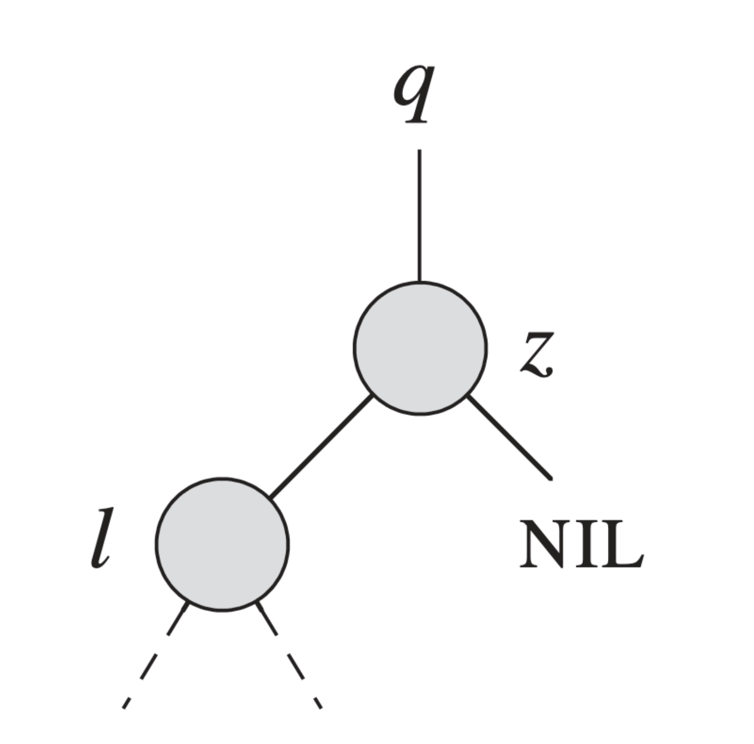


if (z.left = NIL) then
  TRANSPLANT (T, z, r)

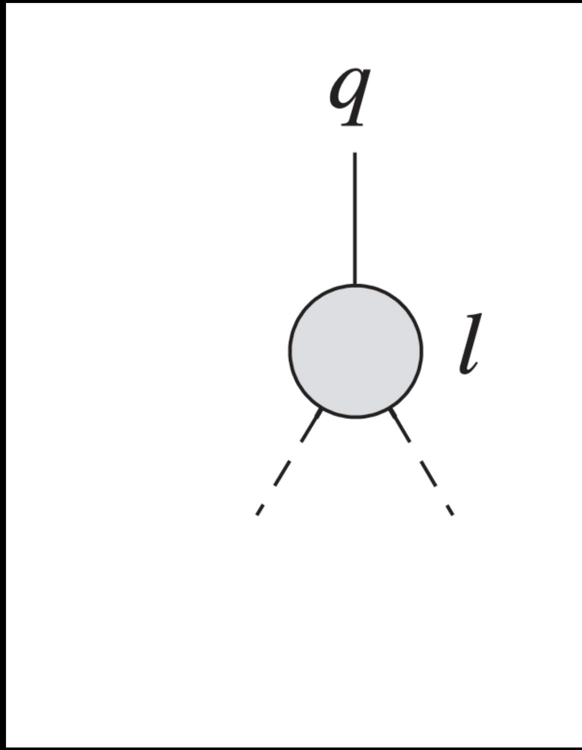


6.5. Deletion

- if *z* has just one child, the left child, then replace *z* by its left child.

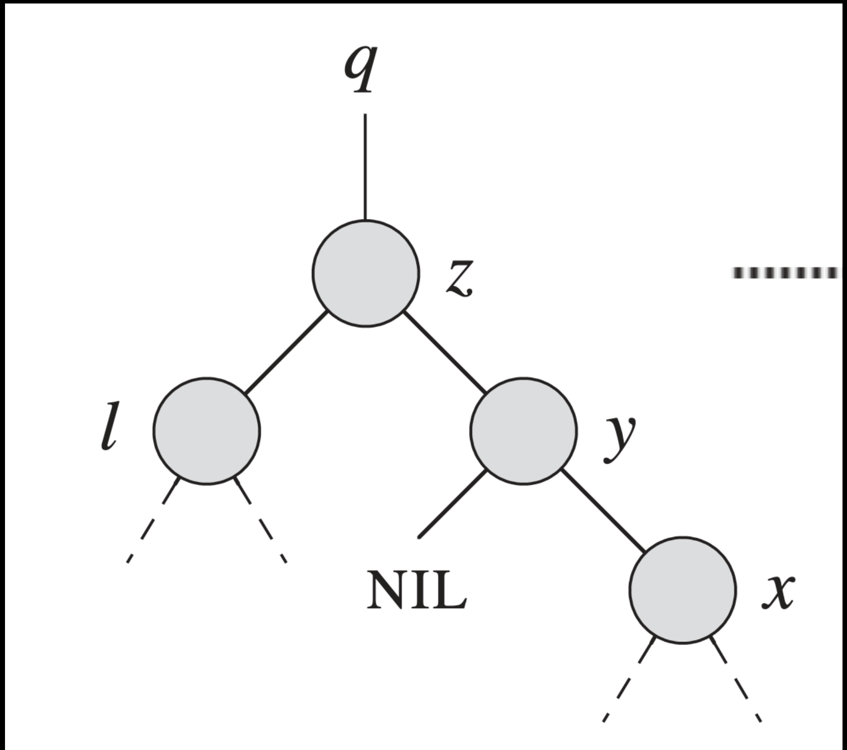



```
if (z.right = NIL) then
  TRANSPLANT (T, z, l)
```

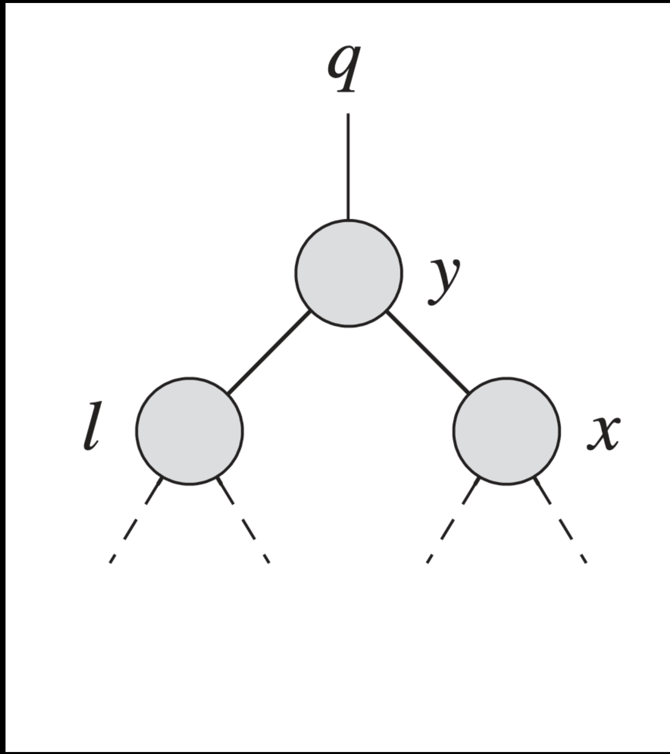


6.5. Deletion

- if *y* is the right child of *z*, then replace *z* by *y* and leave the right child of *y* alone.

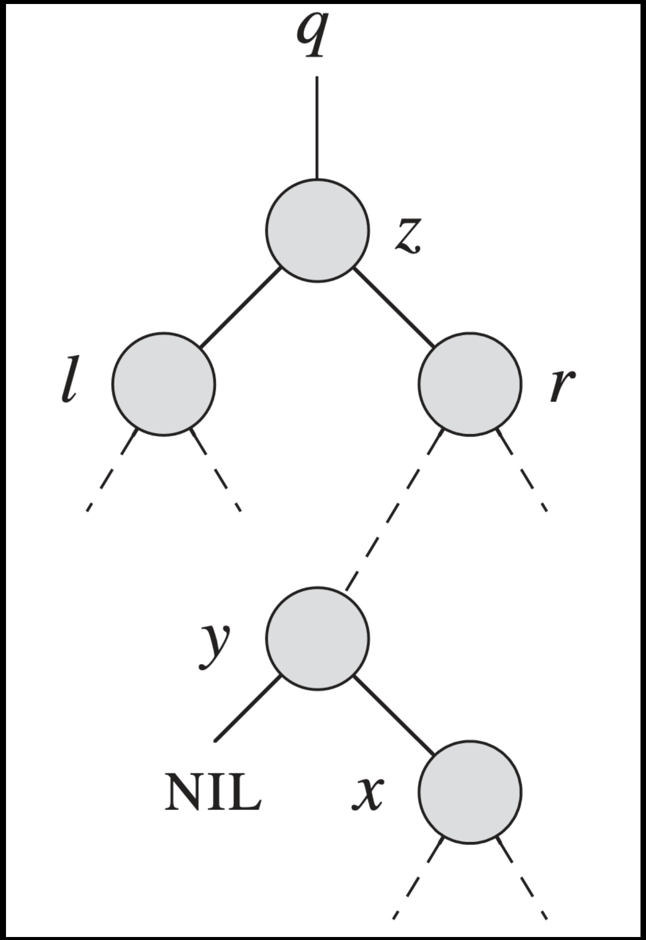



```
y = TREE-MINIMUM(z.right)
if (y.p = z) then
   TRANSPLANT (T, z, y)
```

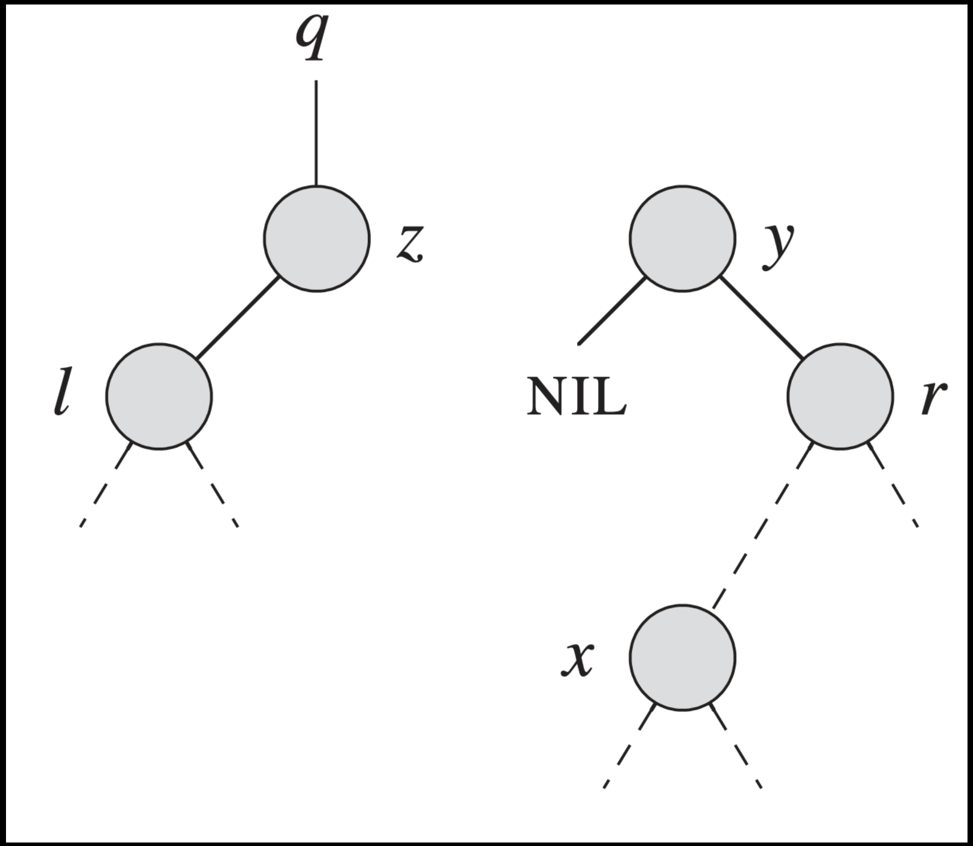


6.5. Deletion

- Otherwise, *y* lie with in the right subtree of z, but it is not the root of this subtree. We replace *y* by its own right child. Then we replace *z* by *y*.

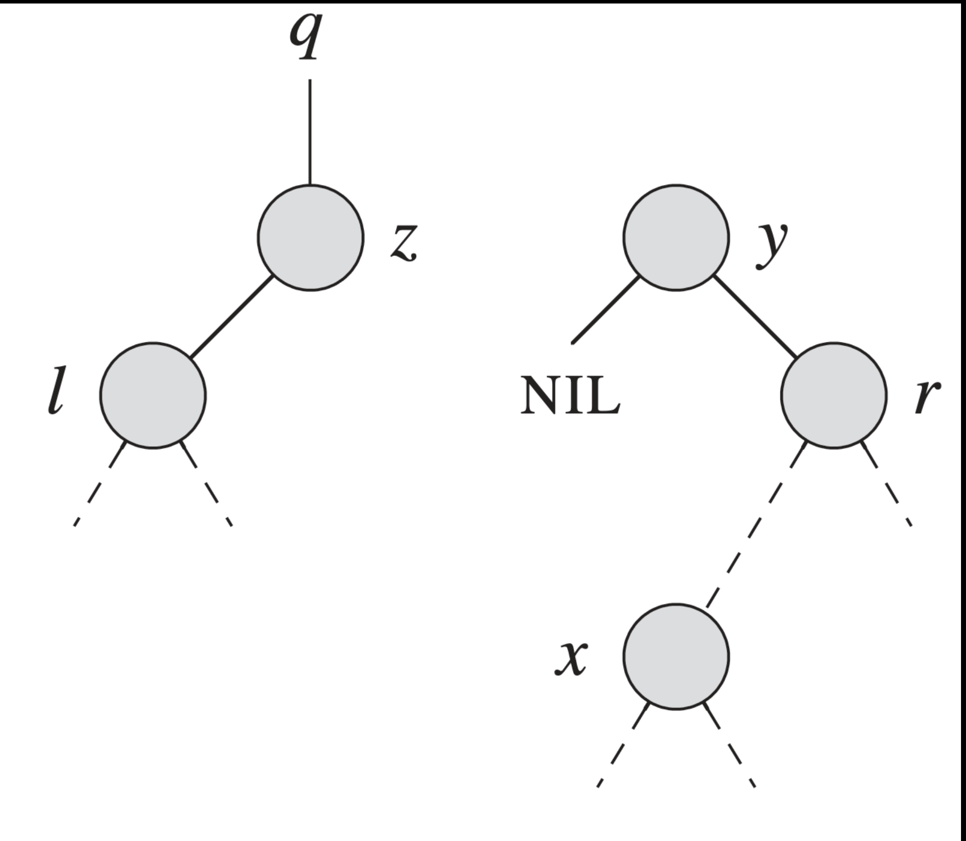



```
y = TREE-MINIMUM(z.right)
if (y.p ≠ z) then
   TRANSPLANT (T, y, y.right)
   y.right = z.right
   y.right.p = y
TRNSPLANT(T, z, y)
y.left = z.left
y.left.p = y
```

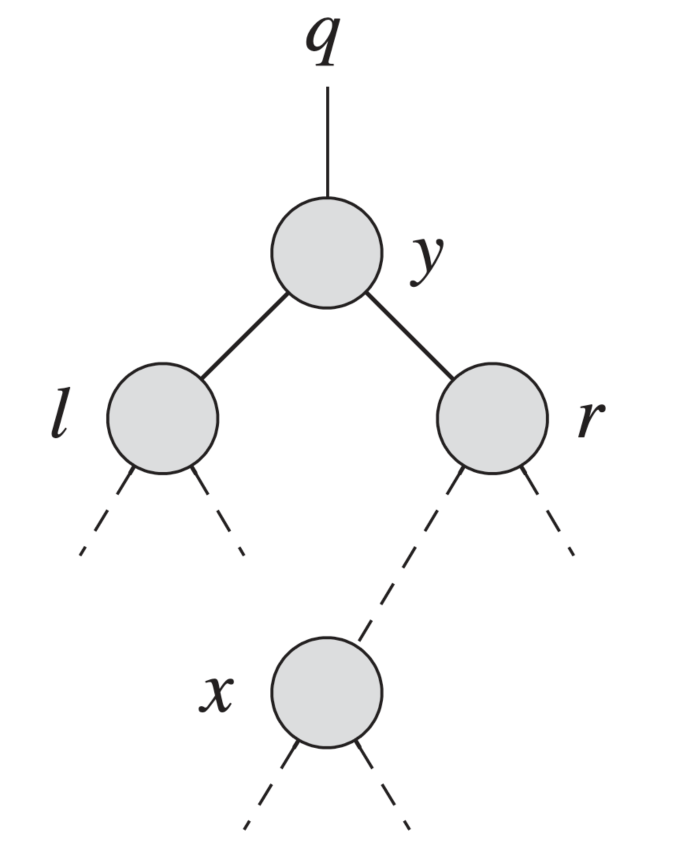


6.5. Deletion

- Otherwise, *y* lie with in the right subtree of z, but it is not the root of this subtree. We replace *y* by its own right child. Then we replace *z* by *y*.

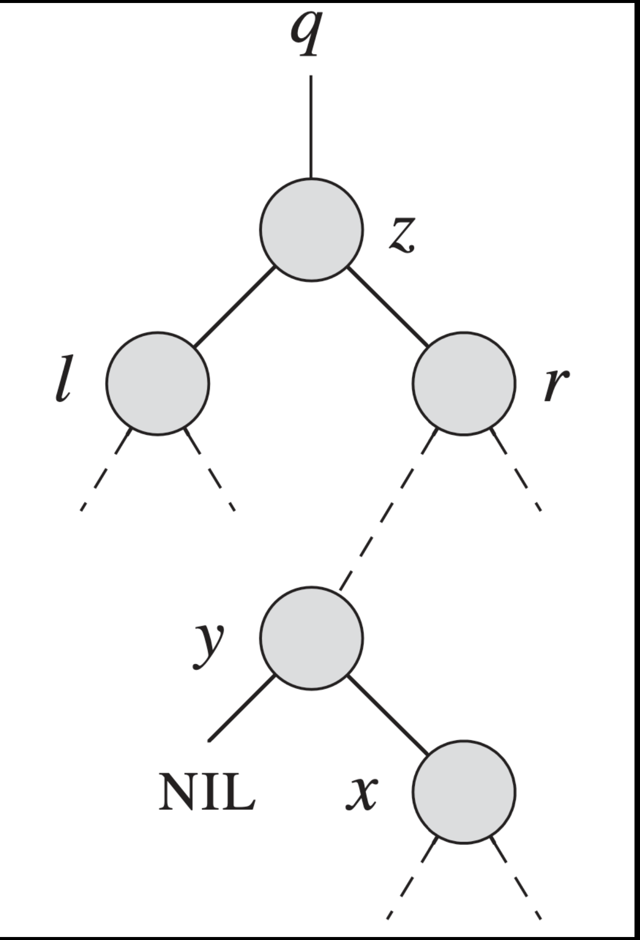


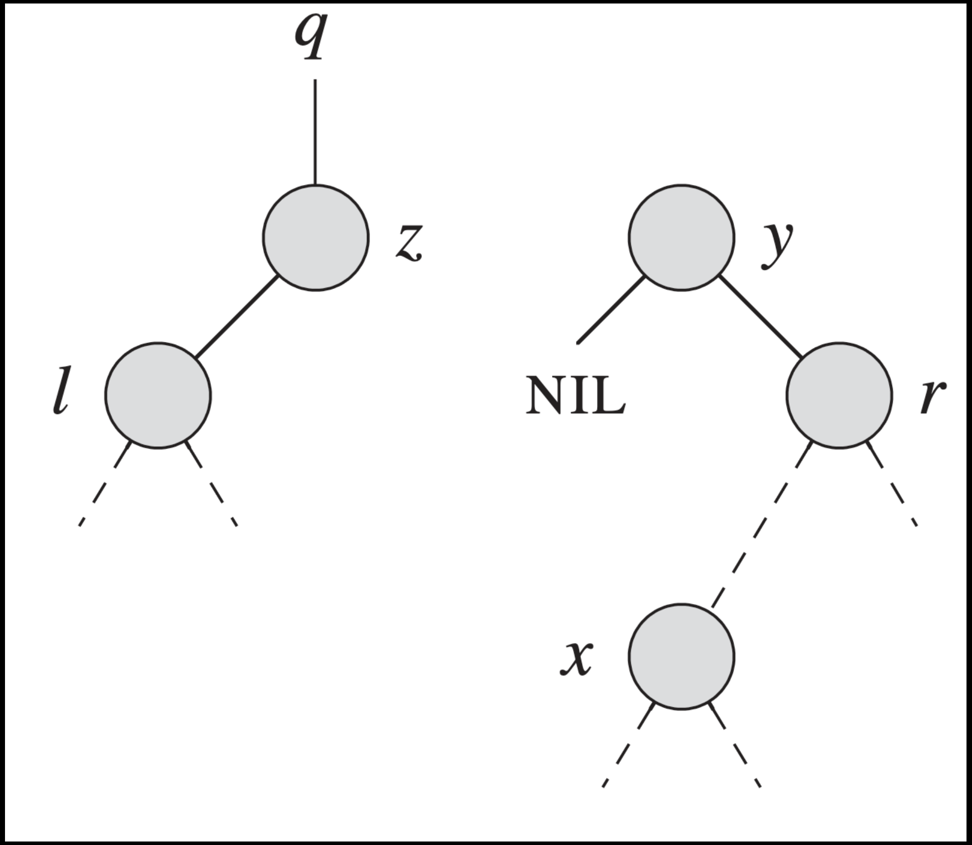

```
y = TREE-MINIMUM(z.right)
if (y.p ≠ z) then
   TRANSPLANT (T, y, y.right)
   y.right = z.right
   y.right.p = y
TRNSPLANT(T, z, y)
y.left = z.left
y.left.p = y
```

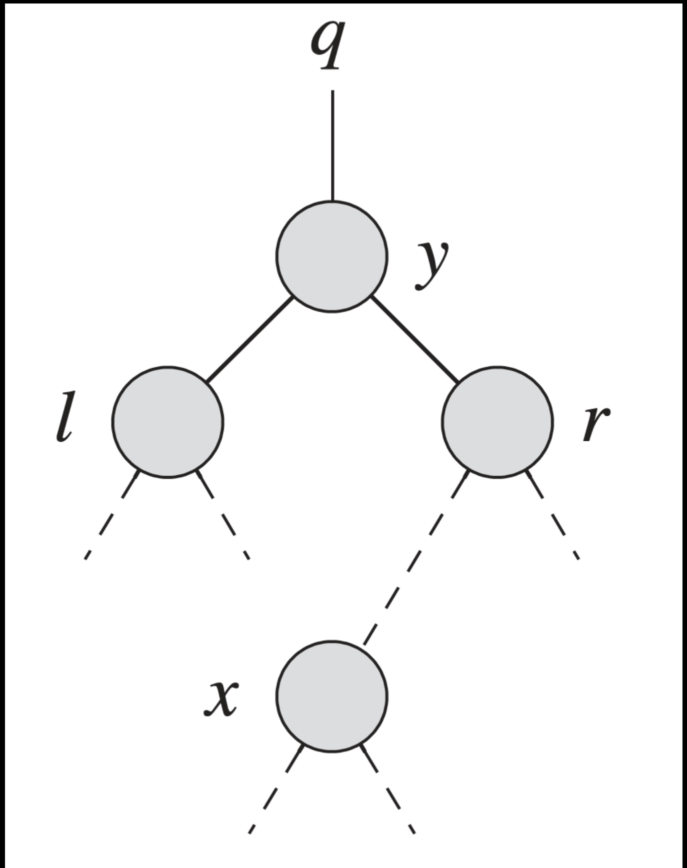


6.5. Deletion

- Otherwise, *y* lie with in the right subtree of z, but it is not the root of this subtree. We replace *y* by its own right child. Then we replace *z* by *y*.







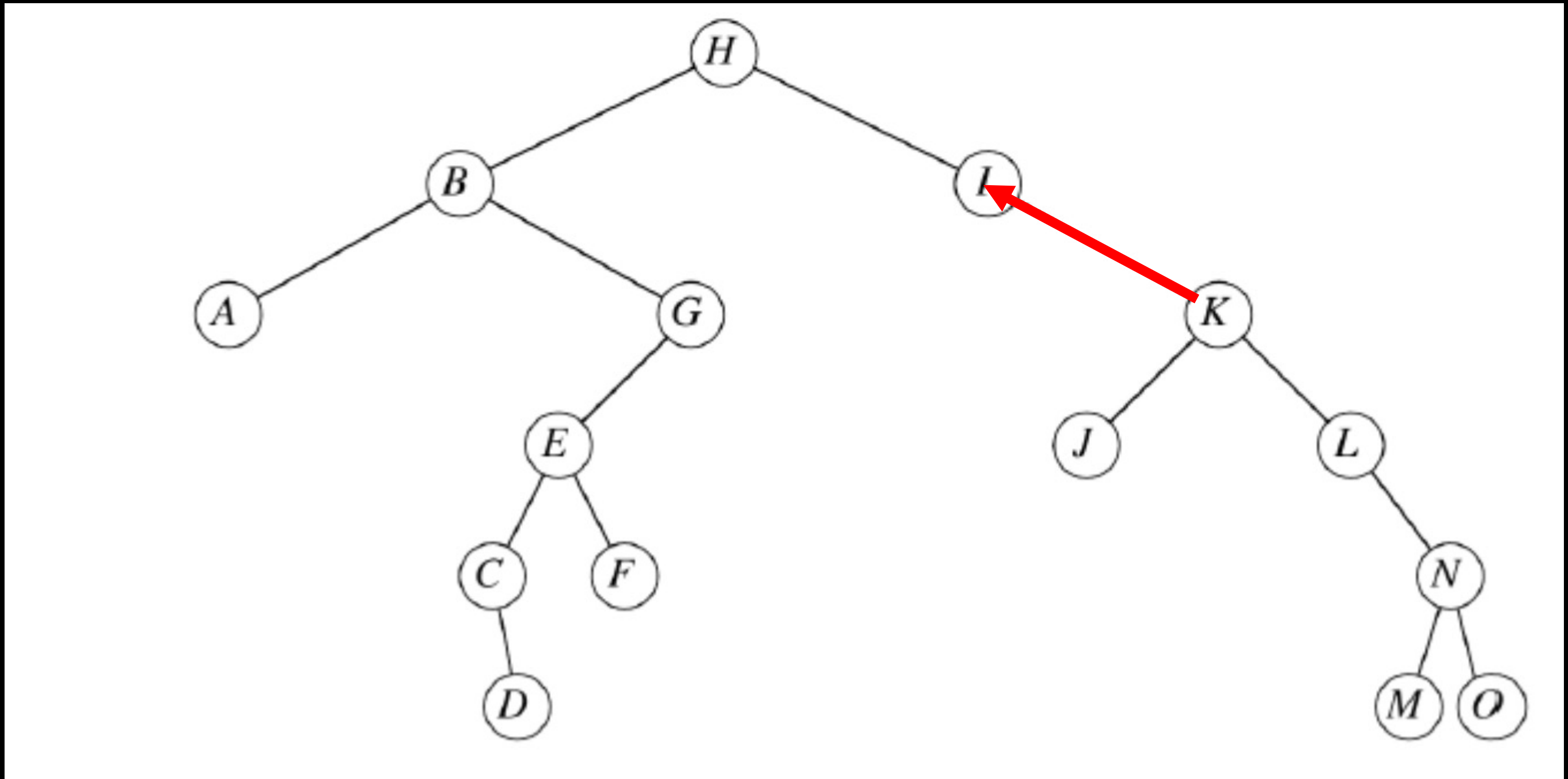
6.5. Deletion

Algorithm (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
```

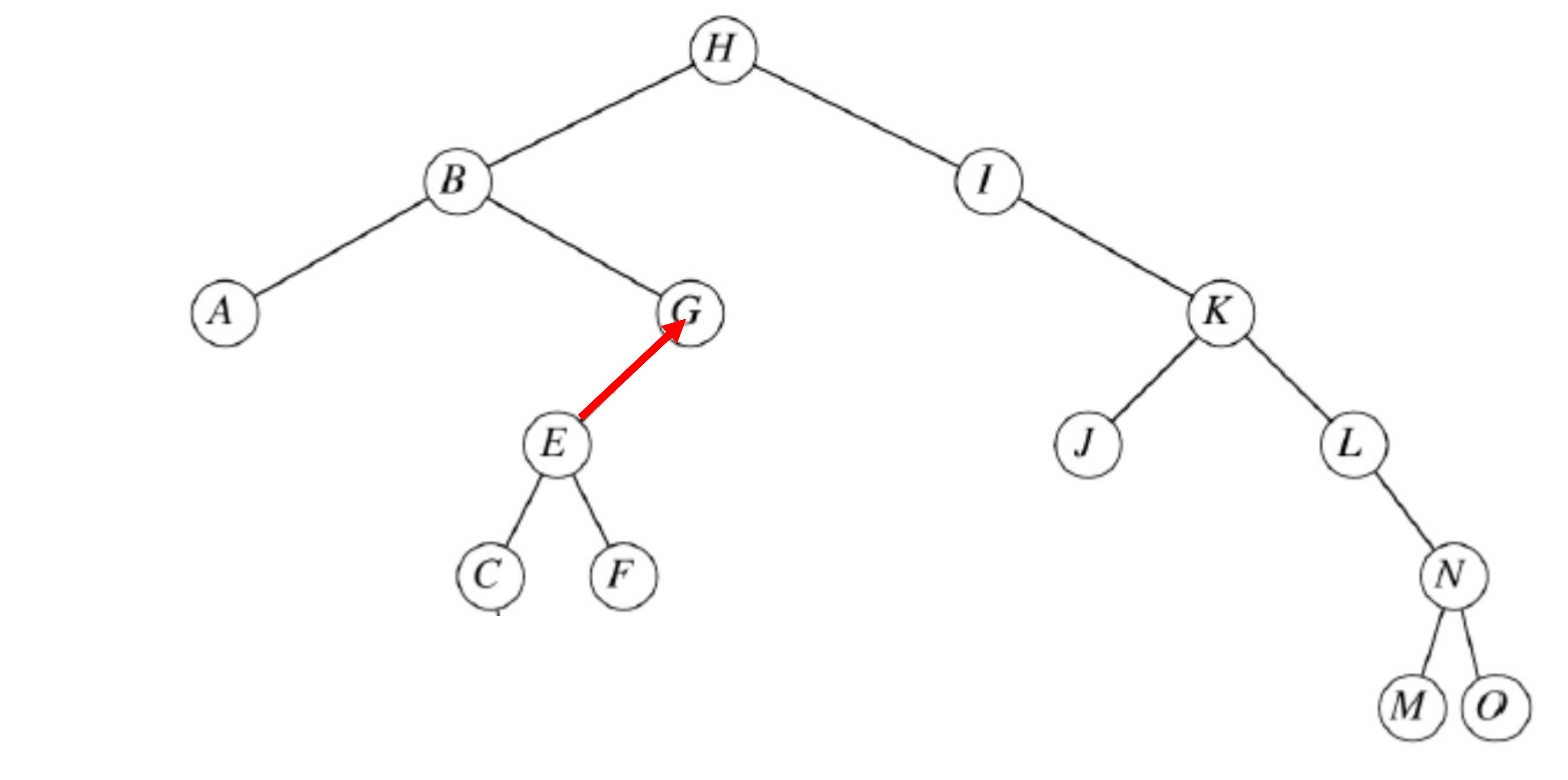
6.5. Deletion

- Tree-delete(T,I): no left child



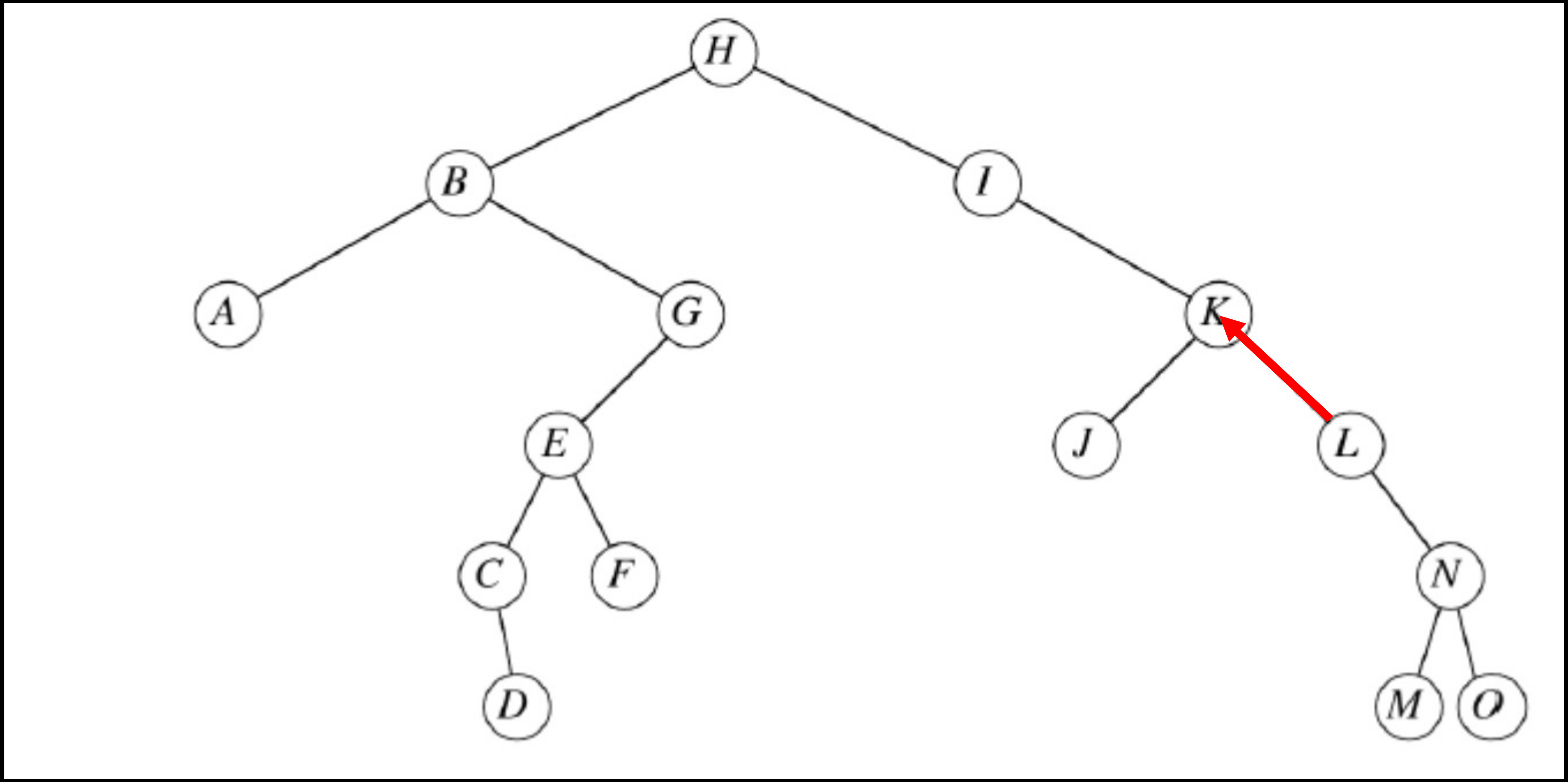
6.5. Deletion

- Tree-delete(T,G): has left, but no right child



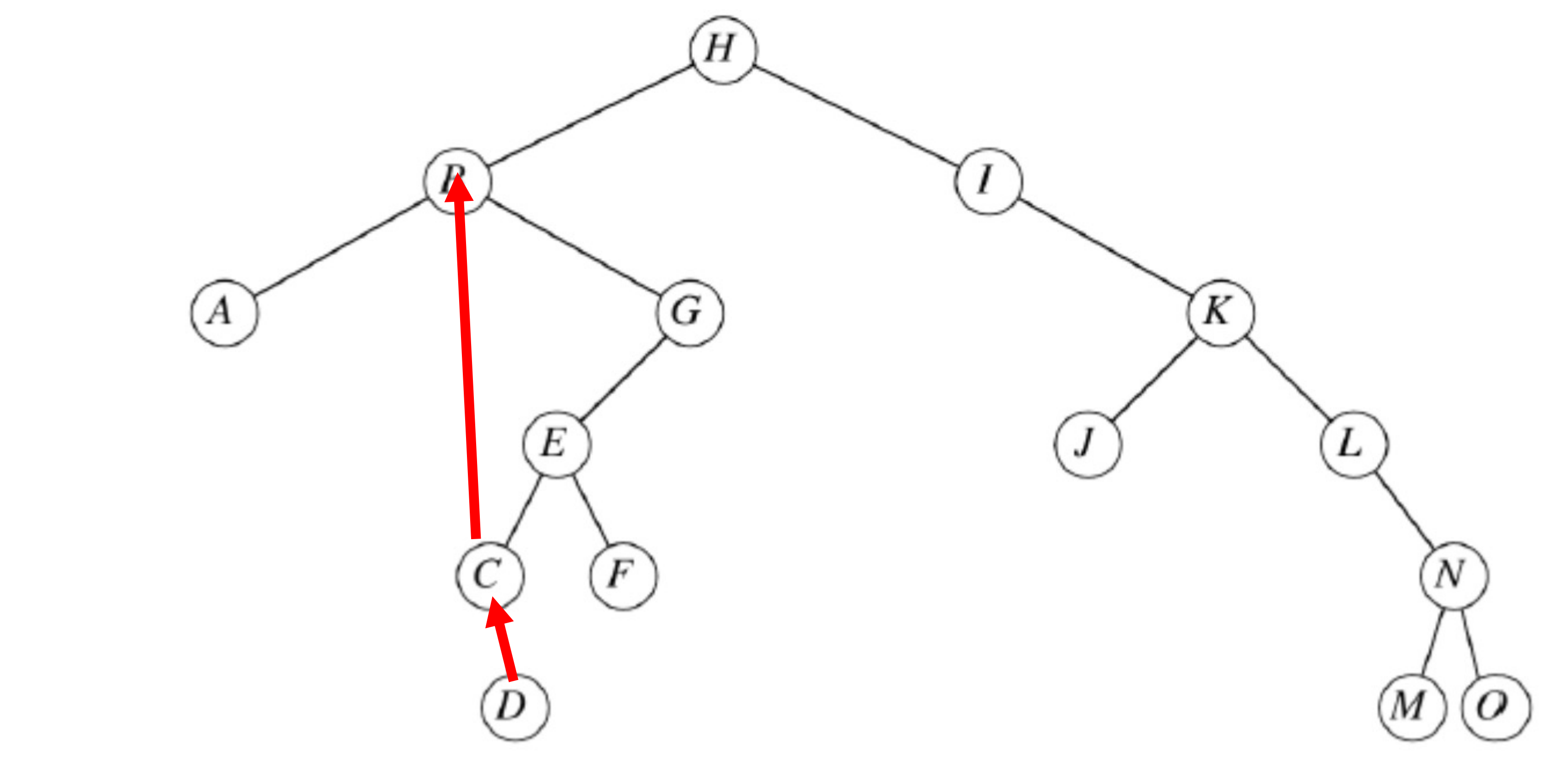
6.5. Deletion

- Tree-delete(T,K): two children, successor is right child



6.5. Deletion

- Tree-delete(T,B): successor is not the right child.



6.5. Deletion

Time: $O(h)$, on a tree of height h.

Minimizing running time

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- Problem: Worst case for binary search tree is $\Theta(n)$ - no better than linked list.
- Solution: Guarantee small height (balanced tree) with $h = O(\lg n)$.
- Method: We restructure the tree. Querying works as before (no adjustments necessary). Insertion or deletion (changing the structure) may require some extra work.

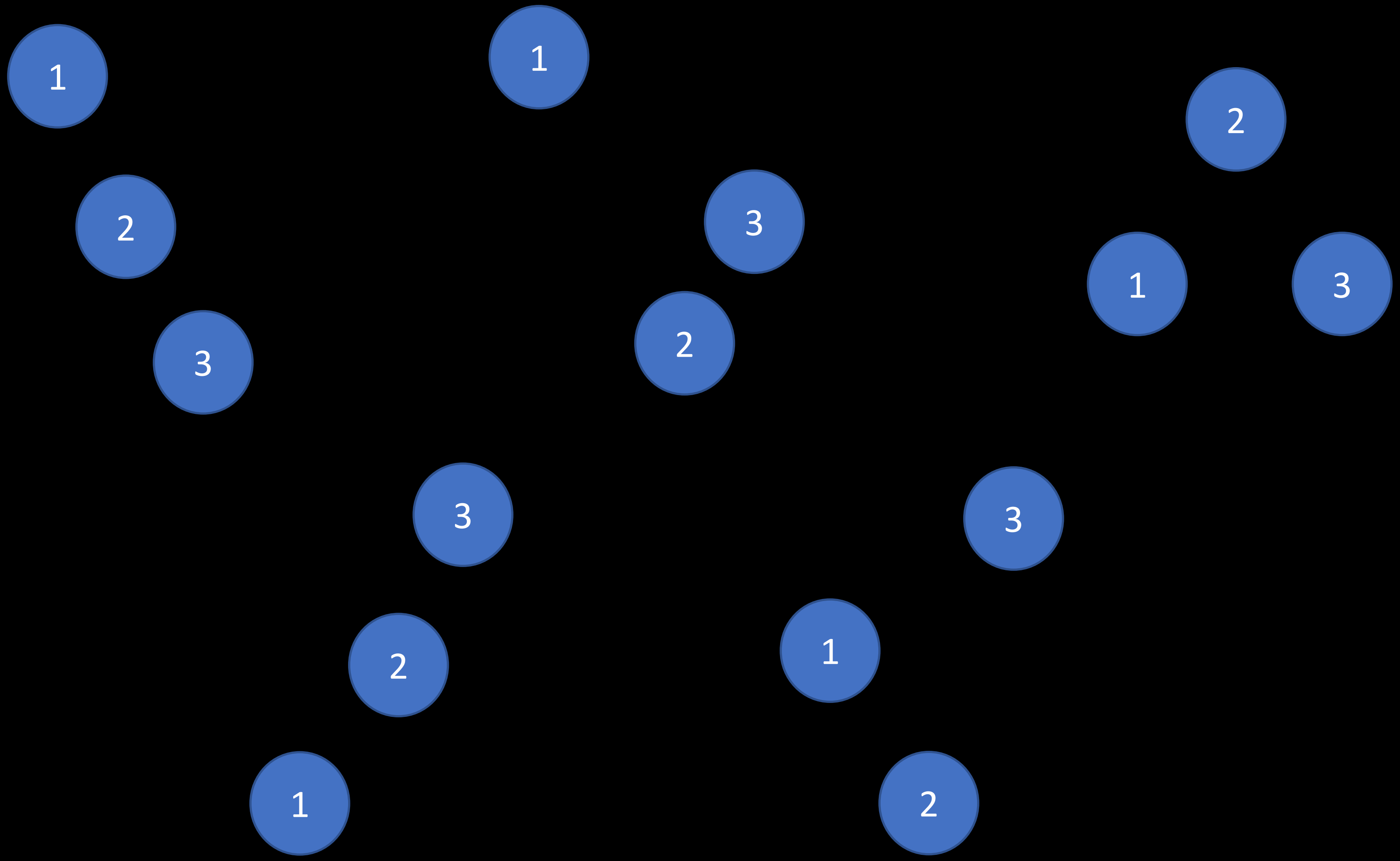
Red-black trees [Next Week] are a special class of binary trees that avoids the worst-case behavior of $O(n)$ like "plain" binary search trees.

6.6. Expected height of a randomly built binary search tree

- Given a set of n distinct keys.
- Insert them in random order into an initially empty binary search tree.
- We assume that each of the $n!$ permutations is equally likely.
- Different from assuming that every binary search tree on n keys is equally likely.
- Try it for $n = 3$. Will get 5 different binary search trees. When we look at the binary search trees resulting from each of the $3!$ input permutations, 4 trees will appear once and 1 tree will appear twice.
- Forget about deleting keys.
- We will show that the expected height of a randomly built binary search tree is $O(\lg n)$.

6.6. Expected height of a randomly built binary search tree

Build all possible BST when A=[1,2,3]



6.6. Expected height of a randomly built binary search tree

Random variables

Define the following random variables:

X_n = height of a randomly built binary search tree on n keys. $Y_n = 2^{X_n} =$ ***exponential height***.

R_n = rank of the root within the set of n keys used to build the binary search tree.

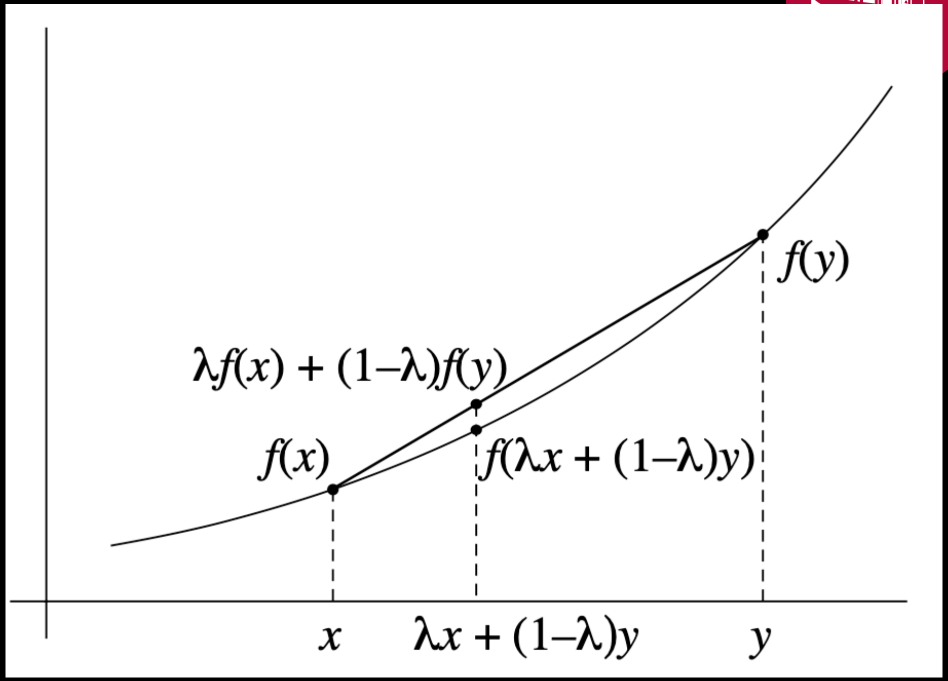
- Equally likely to be any element of $\{1,2,\ldots,n\}$.
- If $R_n = i$, then
  - Left subtree is a randomly-built binary search tree on $i - 1$ keys.
  - Right subtree is a randomly-built binary search tree on $n - i$ keys.

6.6. Expected height of a randomly built binary search tree

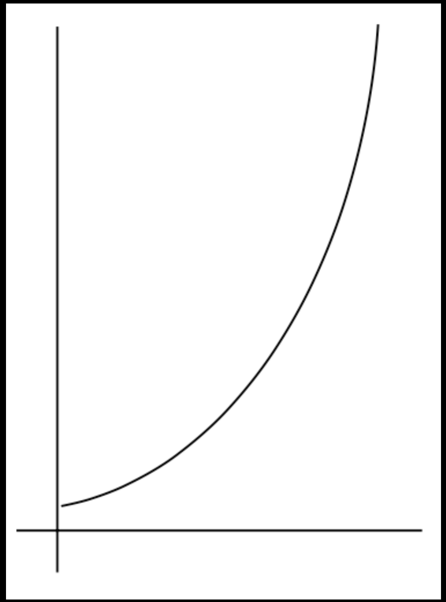
Foreshadowing

We will need to relate E[Yn] to E[Xn].
We'll use ***Jensen's inequality***: $E[f(X)] \geq f(E[X])$ provided

- the expectations exist and are finite, and
  $f(x)$ is ***convex***: for all x, y and all $0 \leq \lambda \leq 1$
- $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$.



- We'll use Jensen's inequality for $f(x) = 2$.
- Since 2^x curves upward, it's convex.



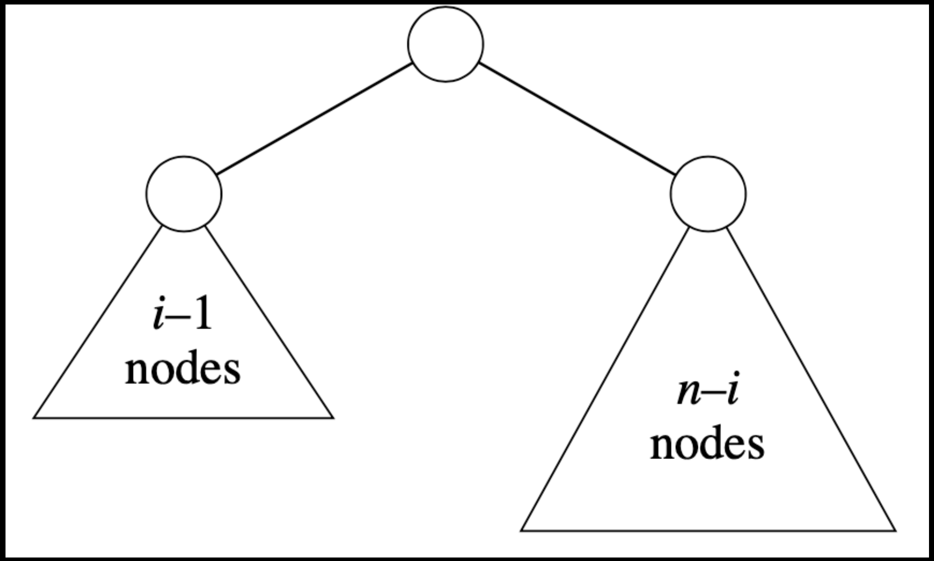
6.6. Expected height of a randomly built binary search tree

Formula for Y_n

Think about Y_n, if we know that $R_n = i$:

Height of root is 1 more than the maximum height of its children:

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}).$$



6.6. Expected height of a randomly built binary search tree

Formula for Y_n

Base cases:
- $Y_1 = 1$ (expected height of a 1-node tree is $2^0 = 1$).
- Define $Y_0 = 0$.

Define indicator random variables $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n} : Z_{n,i} = I\{R_n = i\}$.
R_n is equally likely to be any element of$\{1,2, \ldots, n\}$
$\Rightarrow \Pr\{R_n = i\} = 1/n$
$\Rightarrow E\ [Z_{n,i}]\ =\ 1/n$ (since $E[I\{A\}] = \Pr\{A\}$) [Remember this]
Consider a given n-node binary search tree (which could be a subtree). Exactly one $Z_{n,i}$ is 1, and all others are 0. Hence,

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \cdot (2 \cdot \max(Y_{i-1}, Y_{n-i})) .$$

6.6. Expected height of a randomly built binary search tree

Bounding $E\ [Y_n]$
We will show that $E[Y_n]$ is polynomial in n, which will imply that $E[Xn] = O(\lg n)$.

Claim
$Z_{n,1}$ is independent of Y_{n-1} and Y_{n-i}.
Justification: If we choose the root such that $R_n = i$, the left subtree contains i – 1 nodes, and it's like any other randomly built binary search tree with i – 1 nodes. Other than the number of nodes, the left subtree's structure has nothing to do with it being the left subtree of the root.
Hence, $Y_i - 1$ and $Z_{n,i}$ are independent.
Similarly, Y_{i-1} and $Z_{n,i}$ are independent.

Fact
If X and Y are nonnegative random variables, then $E[\max(X, Y)] \leq E[X] + E[Y]$.

6.6. Expected height of a randomly built binary search tree

Fact

If X and Y are nonnegative random variables, then $\mathrm{E}[\max(X,Y)] \le \mathrm{E}[X]+\mathrm{E}[Y]$.

$$E[Y_n] \le \frac{2}{n}\sum_{i=1}^{n}(E[Y_{i-1}] + E[Y_{n-i}])$$

$$E[Y_0] + E[Y_{n-1}] + E[Y_1] + E[Y_{n-2}] + \cdots + E[Y_{n-1}] + E[Y_0] = 2\sum_{i=0}^{n-1} E[Y_i]$$

$$E[Y_n] \le \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i]$$

6.6. Expected height of a randomly built binary search tree

Solving the recurrence

We will show that for all integers $n > 0$, this recurrence has the solution

$E[Y_n] \leq \frac{4}{n}\binom{n+3}{3}$

Lemma:

$$\sum_{i=0}^{n-1}\binom{i+3}{3} = \binom{n+3}{4}$$

Using Pascal's identity:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n+3}{4} = \sum_{i=0}^{n-1}\binom{i+3}{3}$$

We solve the recurrence by induction on n.

6.6. Expected height of a randomly built binary search tree

Basis: 1 $n = 1$

$$1 = Y_1 = E[Y_1] \le \frac{1}{4}\binom{1+3}{3} = \frac{4}{4} = 1$$

Inductive step: Assume that $E[Y_n] \le \frac{4}{n}\binom{i+3}{3}$ for all $i < n$. Then

$$E[Y_n] \le \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] = \frac{1}{4}\binom{n+3}{3}.$$

6.6. Expected height of a randomly built binary search tree

Bounding E [Xn]

With our bound on E[Yn], we use Jensen's inequality to bound

$$E[Xn]\colon\ 2^{E[X_n]} \le E[2Xn] = E[Yn].$$

Thus,

$$2^{E[X_n]} \le \frac{1}{4}\binom{n+3}{3} = \frac{1}{4}\cdot\frac{(n+3)(n+2)(n+1)}{6} = cn^3$$

Taking logs of both sides gives $E[X_n] = O(\lg n)$.