CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 12. Binary Search Trees

Binary Search Tree: Introduction

- A search tree data structure supports each of the following dynamic set operations

SEARCH(S, k)
A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
A modifying operation that adds the element pointed to by x to the set S. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

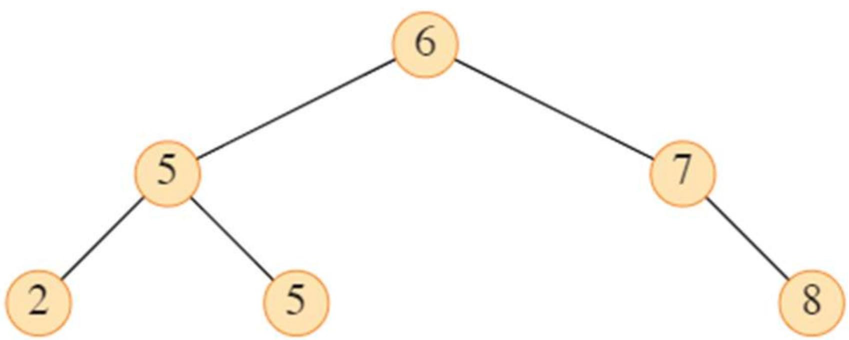
MINIMUM(S) and MAXIMUM(S)
Queries on a totally ordered set S that return a pointer to the element of S with the smallest (for MINIMUM) or largest (for MAXIMUM) key.

SUCCESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

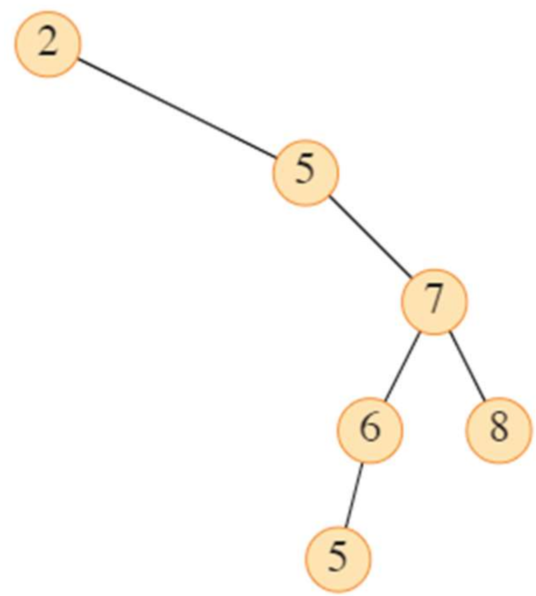
PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Binary Search Tree: Introduction

- Basic operations on a binary search tree take time proportional to the height of the tree.
  - The height of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf.
  - The height of a tree is the height of its root.



height: 2



height: 4

Binary Search Tree: Introduction

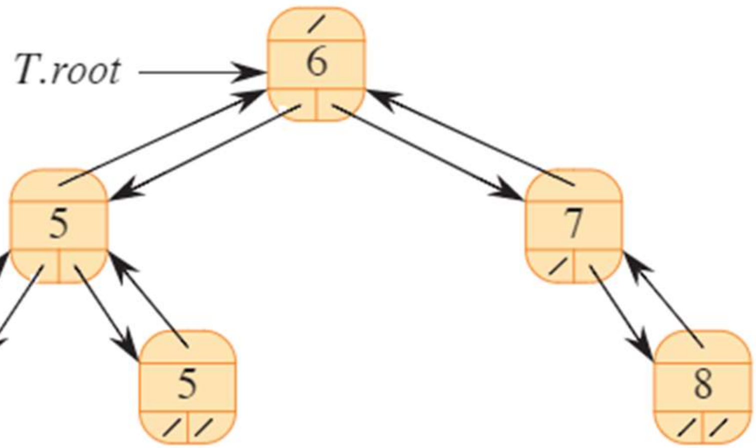
- For a complete binary tree with n nodes, such operations run in $\Theta(\log n)$ worst-case time.
  - A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

- If a tree is a linear chain of nodes, however, the same operations take $\Theta(n)$ worst-case time.

12.1 What is a binary search tree?

Binary Tree Representation

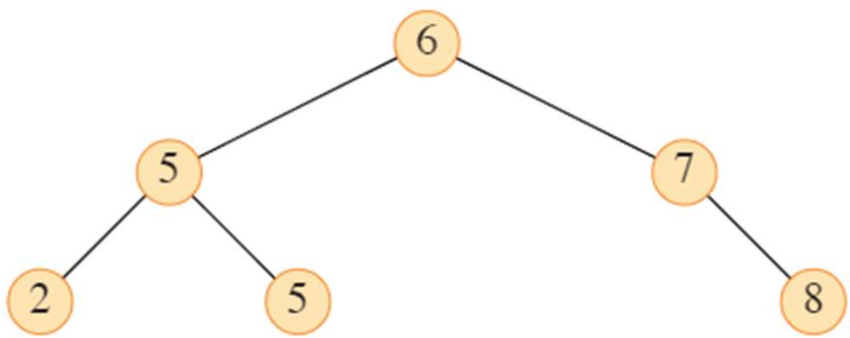
- Representing a binary tree
  - A binary tree can be represented using a linked data structure
  - Each node of a tree has the following attributes
    - *key*
    - *left*: link to left child
    - *right*: link to right child
    - *p*: link to parent
  - If a child or the parent is missing, the appropriate attribute contains NIL.
  - The tree itself has an attribute *root* that points to the root node.



Binary Search Tree: Property

- A binary search tree is a binary tree that satisfies the binary-search-tree property.

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.



Binary Search Tree: Traversal

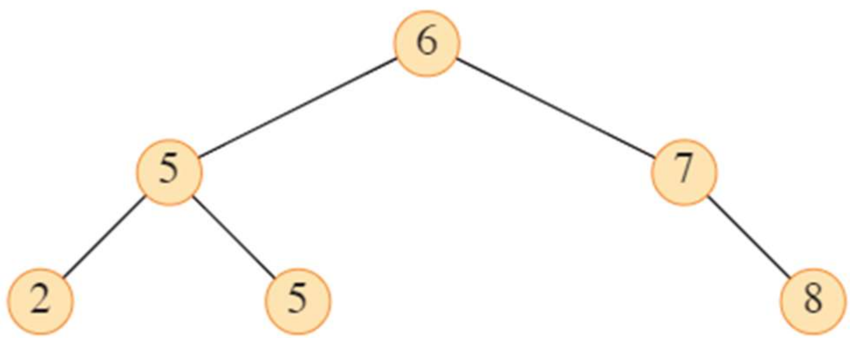
- In order to print all the keys in a sorted order, we can do the in-order tree walk on the tree.
- In-order tree walk is a recursive algorithm where it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.

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

- Other walk algorithms
  - pre-order walk: print root before the values in either subtree
  - post-order walk: print root after the values in either subtree

Binary Search Tree: Example Tree Walk

- In-order tree walk on a binary search tree
  - 2 → 5 → 5 → 6 → 7 → 8
- Pre-order walk
  - 6 → 5 → 2 → 5 → 7 → 8
- Post-order walk
  - 2 → 5 → 5 → 8 → 7 → 6



서강대학교
SOGANG UNIVERSITY

Binary Search Tree: Cost of Walk

- It takes $\Theta(n)$ time to walk an n-node binary search tree
- After the initial call, the procedure calls itself recursively exactly twice for each node in the tree – once for its left child and once for its right child.

- Formal proof that the walk takes $O(n)$ time.
  - Suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n-k-1$ nodes.
  - $T(n) \leq T(k) + T(n-k-1) + d$
  - d is a constant that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK, exclusive of the time spent in recursive calls.
  - Using substitution method, we show that $T(n) \leq (c+d)n + c$. $(T(0) = c)$

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

12.2 Querying a binary search tree

Binary Search Tree: Querying Operations

- Binary search trees can support the queries MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, as well as SEARCH.

- These operations can be supported in $O(h)$ time where h is the height of the tree.

Binary Search Tree: Searching

- Given a pointer x to the root of a subtree and a key k
- TREE-SEARCH(x, k) returns a pointer to a node with key k if one exists in the subtree; otherwise, it returns NIL.

- Recursive implementation

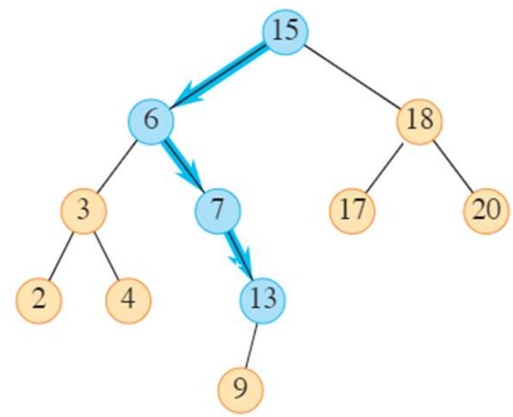
```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

Binary Search Tree: Searching

- Given a pointer x to the root of a subtree and a key k
- TREE-SEARCH(x, k) returns a pointer to a node with key k if one exists in the subtree; otherwise, it returns NIL.

- Recursive implementation

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



$k = 13$

- The nodes encountered during the recursion form a simple path downward from the root of the tree → running time of TREE-SEARCH is $O(h)$.

Binary Search Tree: Searching

- Iterative implementation

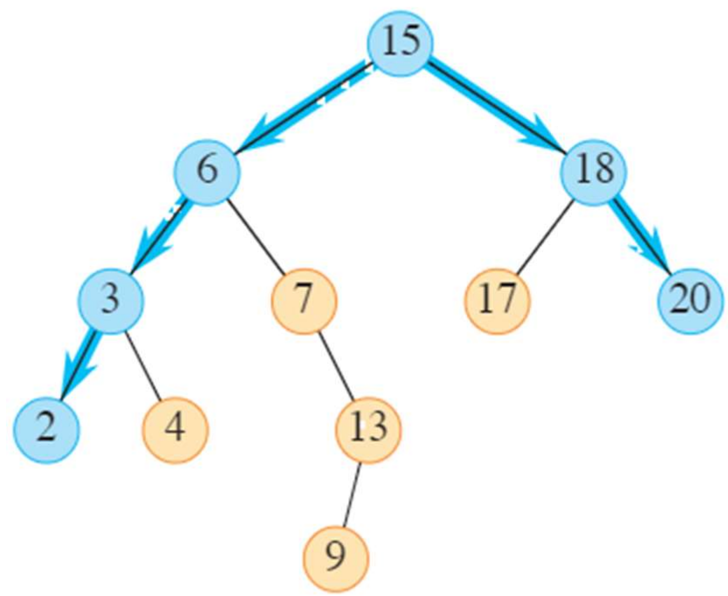
```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Binary Search Tree: Minimum and Maximum

- To find the minimum key in the tree, just follow the left child pointers from the root until you encounter a NIL.
- To find the maximum key in the tree, just follow the right child pointers from the root until you encounter a NIL.

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

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



- Similar to searching, finding minimum or maximum takes $\Theta(h)$ time.

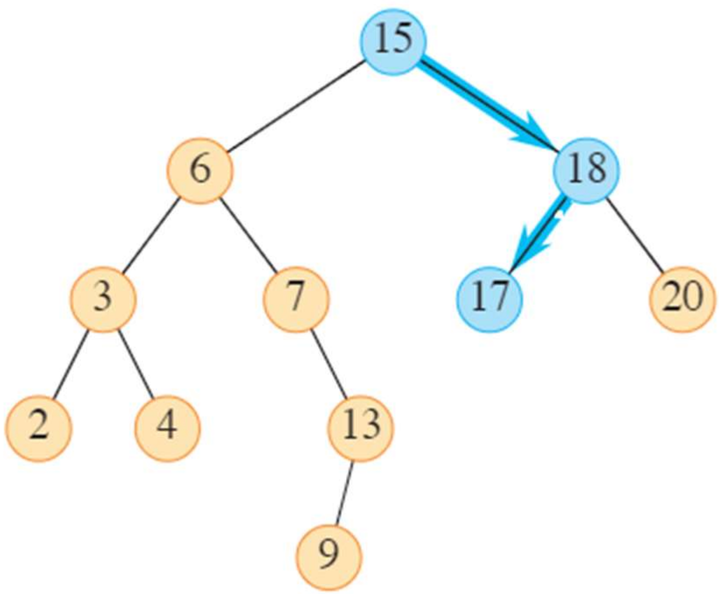
Binary Search Tree: Successor and Predecessor

- If all keys are distinct, the successor of a node x is the node with the smallest key greater than $x.key$.
- In a binary search tree, the successor of a node is the next node visited in an in-order tree walk.

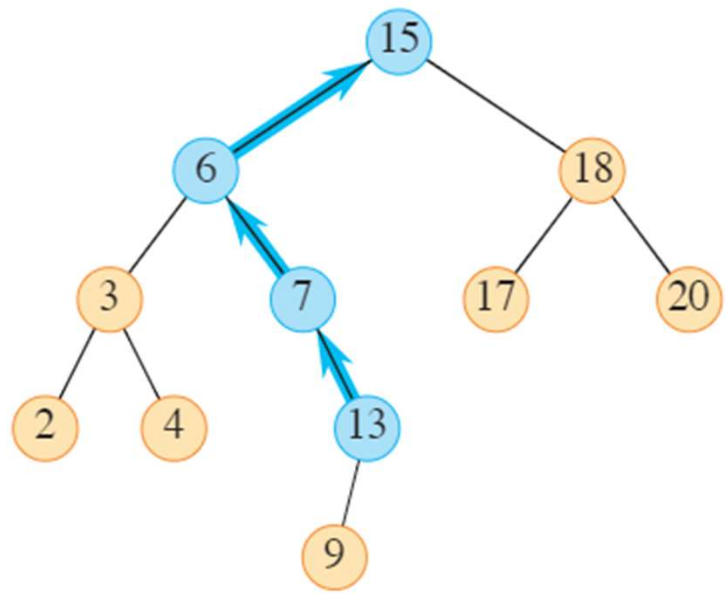
- There are 2 case when finding successor of node x in a binary search tree.
- Case 1: If the right subtree of node x is nonempty, then the successor of x is the leftmost node in x's right subtree.
- Case 2: if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

Binary Search Tree: Successor and Predecessor

- Example



right child is nonempty



right child is empty

Binary Search Tree: Successor and Predecessor

- TREE-SUCCESSOR procedure

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)   // leftmost node in right subtree
else // find the lowest ancestor of x whose left child is an ancestor of x
    y = x.p
    while y ≠ NIL and x == y.right
        x = y
        y = y.p
    return y
```

- The running time of TREE-SUCCESSOR is $O(h)$, since it either follows a simple path up the tree or follows a simple path down the tree.

- TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

12.3 Insertion and deletion

Insertion and Deletion

- When we insert an element to the binary search tree or delete an element from the tree, we modify the binary search tree structure.
- Insertion and deletion must be done in a way that the binary-search-tree property continues to hold after insertion or deletion.

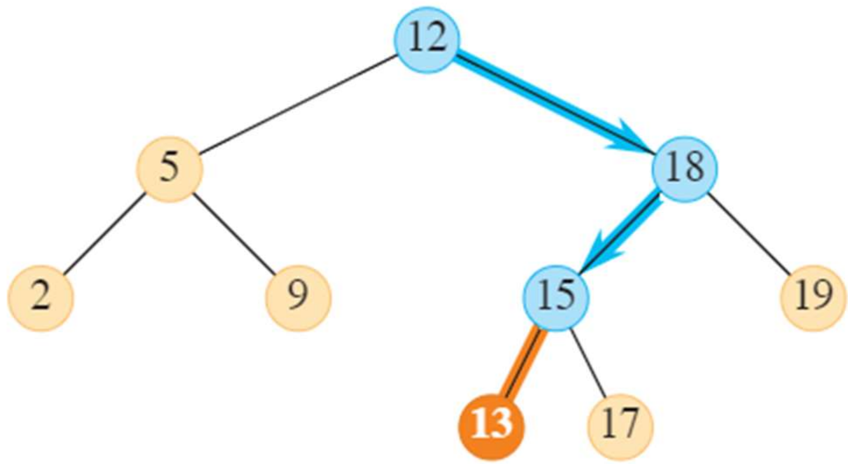
Insertion

- The TREE-INSERT procedure inserts a new node into a binary search tree.
- The procedure takes a binary tree T and a node z for which $z.key$ has already been filled, $z.left = \text{NIL}$ and $z.right = \text{NIL}$.
- z must be inserted into an appropriate position in the tree.

```
TREE-INSERT(T, z)
x = T.root                  // node being compared with z
y = NIL                     // y will be parent of z
while x ≠ NIL               // descend until reaching a leaf
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                     // found the location—insert z with parent y
if y == NIL
    T.root = z              // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Insertion

- The procedure maintains a pointer x that traces a simple path downward looking for a NIL to replace with the input node z.
- The procedure also maintains a trailing pointer y as the parent of x.
- Once x finds NIL, z should be inserted in that position.
- x is inserted into the tree as y's child.



- Like other operations, it is easy to see that TREE-INSERT runs in $O(h)$ time.

Deletion

- The overall strategy for deleting a node z from a binary search tree T has three basic cases.

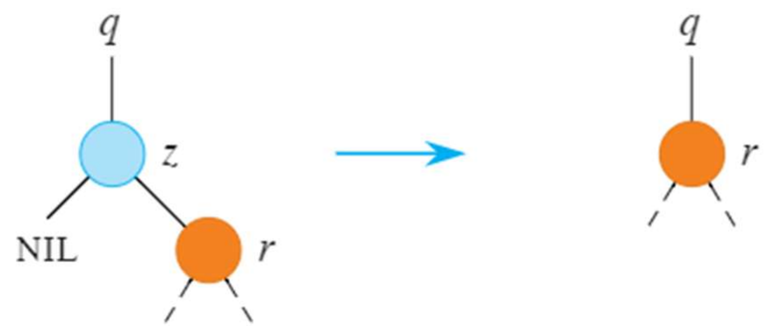
- Case 1: If z has no children, then simply remove it by modifying its parent to replace z with NIL as its child.

- Case 2: If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.

- Case 3: If z has two children, find z's successor y – which must belong to z's right subtree – and move y to take z's position in the tree.
  - The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.
  - Because y is z's successor, it cannot have a left child, and y's original right child moves into y's original position, with the rest of y's original right subtree following automatically.

Deletion: Case 2

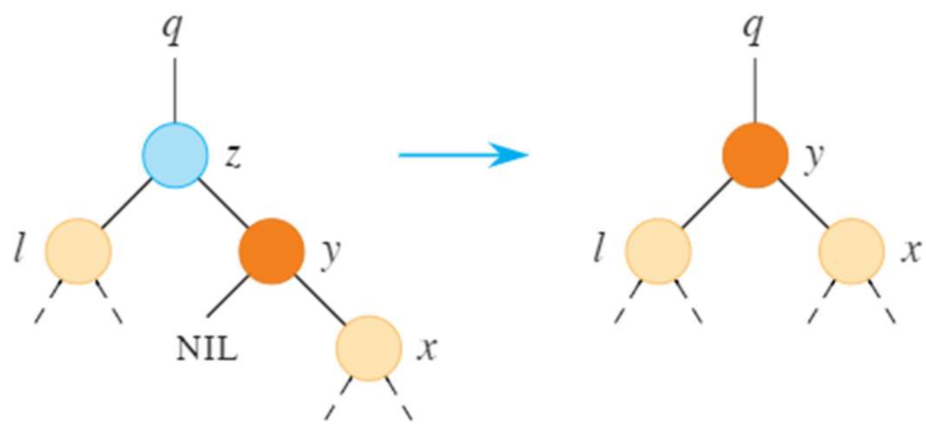
- If z has no left child, replace z by its right child.
  – If right child is NIL, it is case 1.



- If z has no right child, place z by its left child.
  – If left child is NIL, it is case 1.

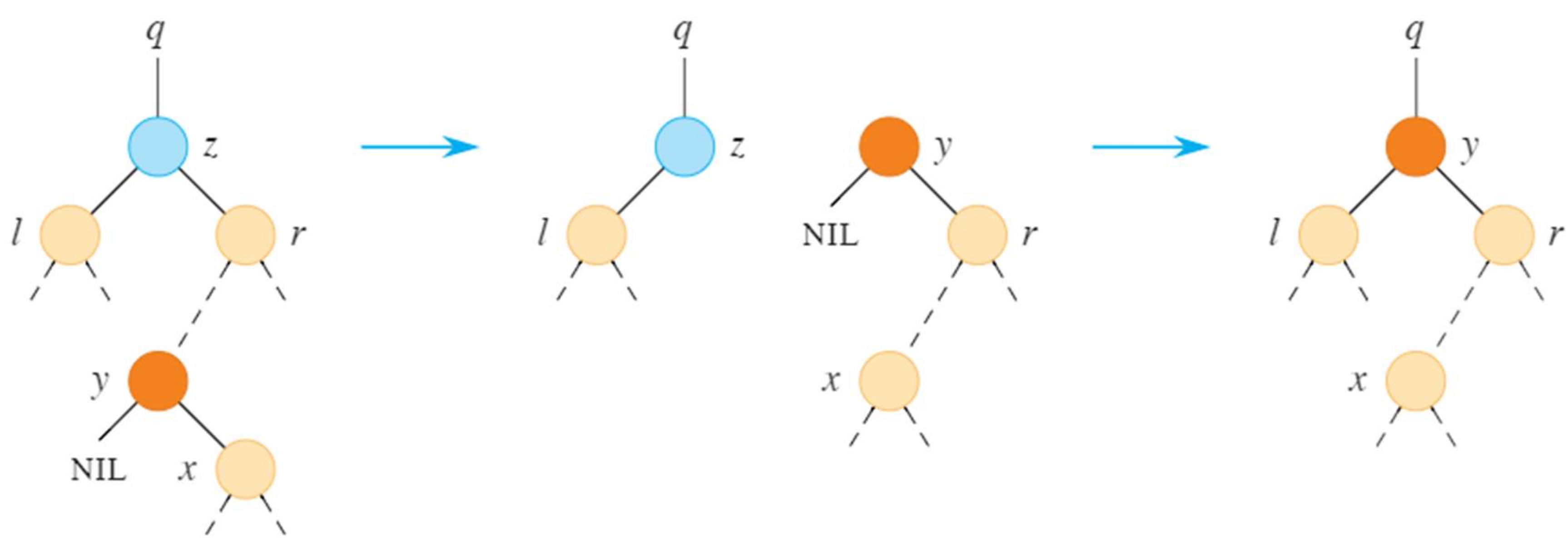
Deletion: Case 3-i

- If the successor y is z's right child, replace z by y, leaving y's child alone.



Deletion: Case 3-ii

- Otherwise, y lies within z's right subtree but is not z's right child. In this case, first replace y by its own right child, and then replace z by y.



Deletion: TRANSPLANT

- The subroutine TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent and u's parent ends up having v as its appropriate child.
  - v can be NIL.

```
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
```

Deletion: TREE-DELETE

- TREE-DELETE removes node z from the binary search tree T.

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)          // replace z by its right child
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)           // replace z by its left child
else y = TREE-MINIMUM(z.right)         // y is z's successor
    if y ≠ z.right                     // is y farther down the tree?
        TRANSPLANT(T, y, y.right)      // replace y by its right child
        y.right = z.right              // z's right child becomes
        y.right.p = y                  //       y's right child
    TRANSPLANT(T, z, y)                // replace z by its successor y
    y.left = z.left                    // and give z's left child to y,
    y.left.p = y                       //       which had no left child
```

- All lines except for the call to TREE-MINIMUM takes constant time.
- Thus, TREE-DELETE runs in $O(h)$ time.

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr