Data Structures

Lecture 20: Binary Search Trees (cont.)

Nopadon Juneam
Department of Computer Science
Kasetsart university

Outlines

- Update operations on binary search trees
  - Insertion & deletion

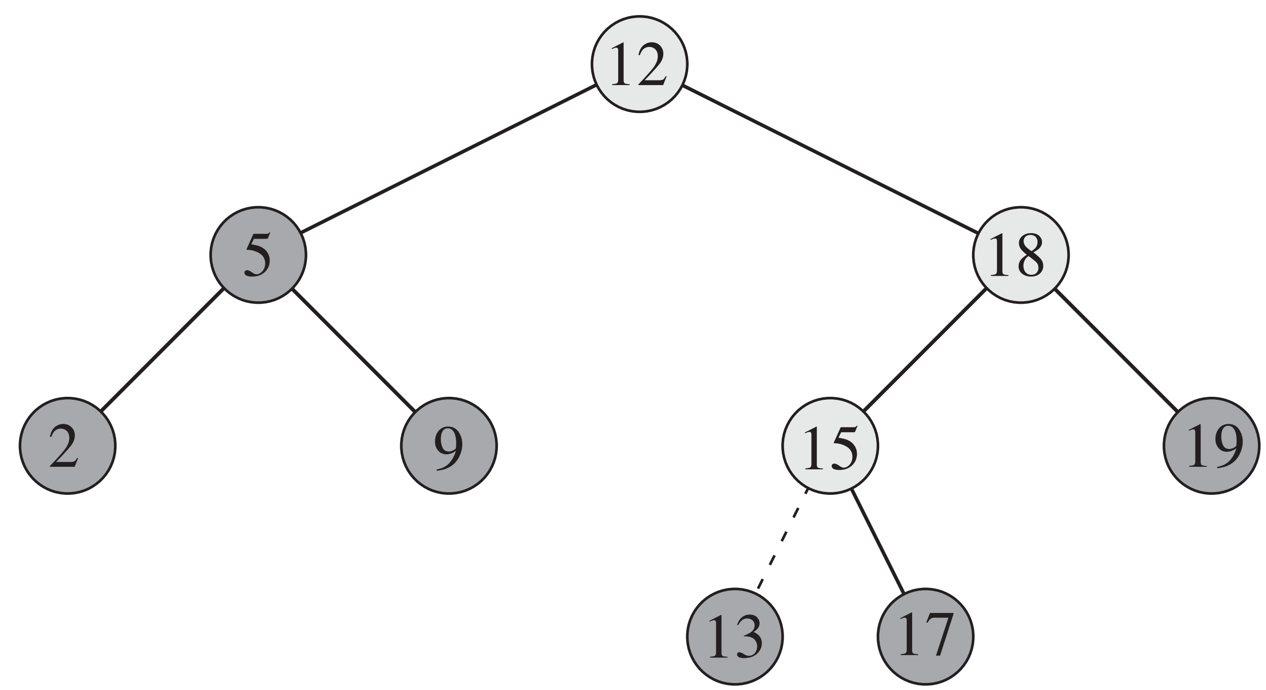
More Operations on Binary Search Trees

- Update operations performed a binary search tree *T*:
  - `Tree-Insert(z, `*T*`):` insert a new node *z* for which *z.key* = *v*, z.*parent* = NULL*, z*.*left* = NULL, and *z*.*right* = NULL into an appropriate position in the search tree *T*
  - `Tree-Delete(z, `*T*`):` delete an existing node *z* from the search tree *T*

Operation: Tree-Insert (1)

- `Tree-Insert(z, T):` insert a new node *z* for which *z.key* = *v*, z.*parent* = NULL, *z.left* = NULL, and *z.right* = NULL into an appropriate position in the tree *T*

```
Tree-Insert(z, T):
   y = NULL
   x = T.root
   while (x != NULL):
       y = x
       if z.key < x.key:
          x = x.left
       else
          x = x.right
   z.parent = y
   if y != NULL:
      if z.key < y.key:
          y.left = z
      else:
          y.right = z
```



Operation: Tree-Insert (2)

- Like `search,` we begin at the root of the tree and the pointer *x* traces a simple path downward looking for a NULL to replace with the input item *z*
    - We maintain the trailing pointer *y* as the parent of *x*
    - After initialization, the while loop in causes these two pointers to move down the tree, going left or right depending on the comparison of *z.key* with *x.key*, until *x* becomes NULL.
    - This NULL occupies the position where we wish to place the input item *z*
    - We need the trailing pointer *y*, because by the time we find the NULL where *z* belongs, the search has proceeded one step beyond the node that needs to be changed

```
Tree-Insert(z, T):
   y = NULL
   x = T.root
   while (x != NULL):
       y = x
       if z.key < x.key:
          x = x.left
       else
          x = x.right
   z.parent = y
   if y != NULL:
      if z.key < y.key:
          y.left = z
      else:
          y.right = z
```

Operation: Tree-Delete (1)

- `Tree-Delete(z, T):` delete an existing node *z* from the search tree *T*
  - The overall strategy for deleting a node *z* from the search tree T has three basic cases:
    - **Case A.1**: If *z* has no children, then we simply remove it by modifying its parent to replace *z* with NULL as its child
    - **Case A.2**: If *z* has just one child, then we elevate that child to take *z*'s position in the tree by modifying *z*'s parent to replace *z* by *z*'s child
    - **Case A.3**: If *z* has two children, then we find *z*'s successor *y*—which must be in *z*'s right subtree—and have *y* 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. This case is the tricky one because it matters whether *y* is z's right child

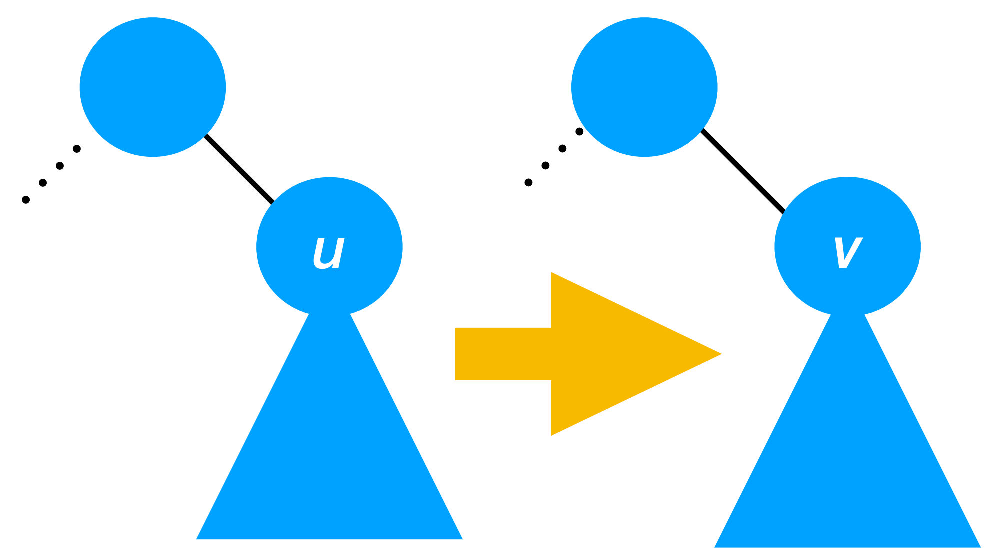
Operation: Tree-Delete (2)

- However, we organize the cases a bit differently from the three cases outlined previously:
  - **<u>Case B.1</u>**: If *z* has no left child, then we replace *z* by its right child, which may or may not be NULL.
    - If *z*'s right child is NULL, this case deals with the situation in which z has no children
    - If *z*'s right child is not NULL, this case handles the situation in which *z* has just one child, which is its right child
  - **<u>Case B.2</u>**: If *z* has just one child, which is its left child, then we replace *z* by its left child
  - **<u>Case B.3</u>**: If *z* has both a left and a right child. We find *z*'s successor *y*, which lies in *z*'s right subtree and has no left child. We want to splice *y* out of its current location and have it replace *z* in the tree
    - **<u>Case B.3.1</u>**: If *y* is *z*'s right child, then we replace *z* by *y*, leaving *y*'s right child alone
    - **<u>Case B.3.2</u>**: If *y* lies within *z*'s right subtree but is not *z*'s right child. In this case, we first replace *y* by its own right child, and then we replace *z* by *y*

Operation: Tree-Delete (3)

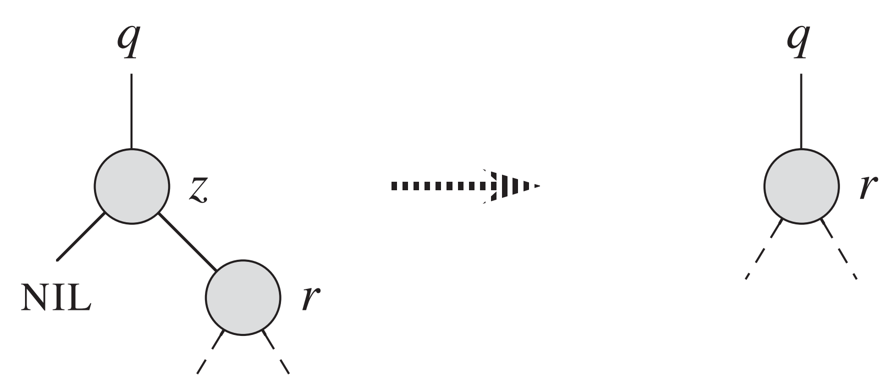
- In order to move subtrees around within the binary search tree, we define a subroutine `Transplant` which replaces one subtree as a child of its parent with another subtree
- `Transplant(T, u, v):` Replace the subtree rooted at node *u* with the subtree rooted at node *v*, node *u*'s parent becomes node *v*'s parent, and *u*'s parent ends up having *v* as its appropriate child

```
Transplant(T, u, v):
   if (u.parent != NULL):
      if u == u.parent.left:
         u.parent.left = v
      else:
         u.parent.right = v
   if (v != NULL):
      v.parent = u.parent
```



Operation: Tree-Delete (4)

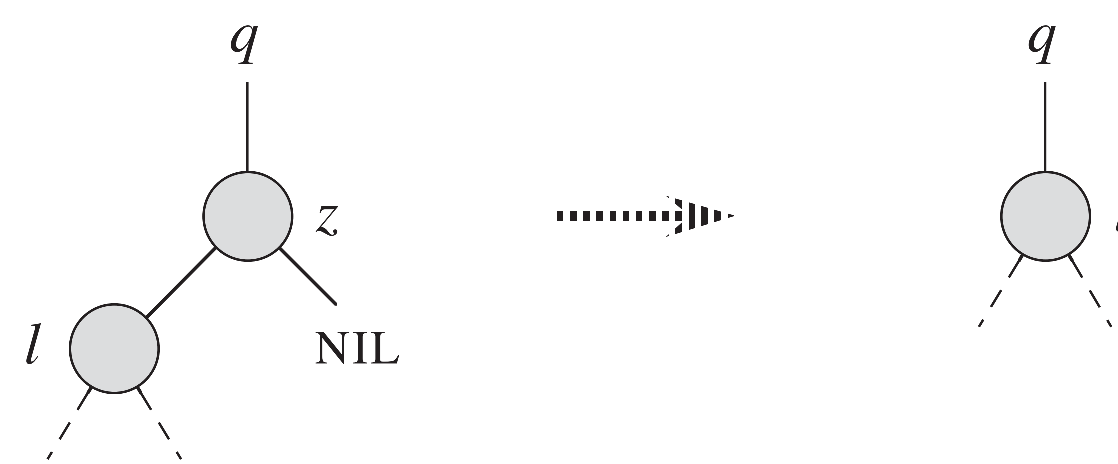
- **<u>Case B.1</u>**: If *z* has no left child, then we replace *z* by its right child, which may or may not be NULL
    - If *z*'s right child is NULL, this case deals with the situation in which z has no children
    - If *z*'s right child is not NULL, this case handles the situation in which *z* has just one child, which is its right child




```
//Case B.1
if z.left == NULL:
   Transplant(T,z,z.right)
```

Operation: Tree-Delete (5)

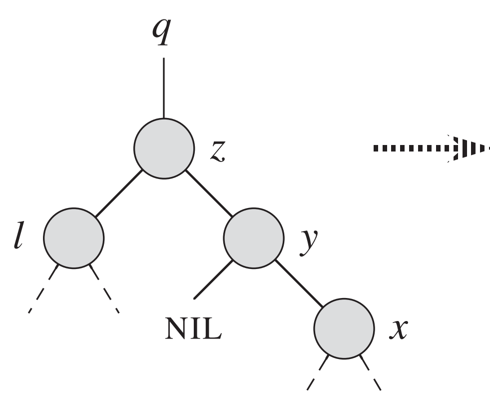
- **Case B.2**: If *z* has just one child, which is its left child, then we replace *z* by its left child



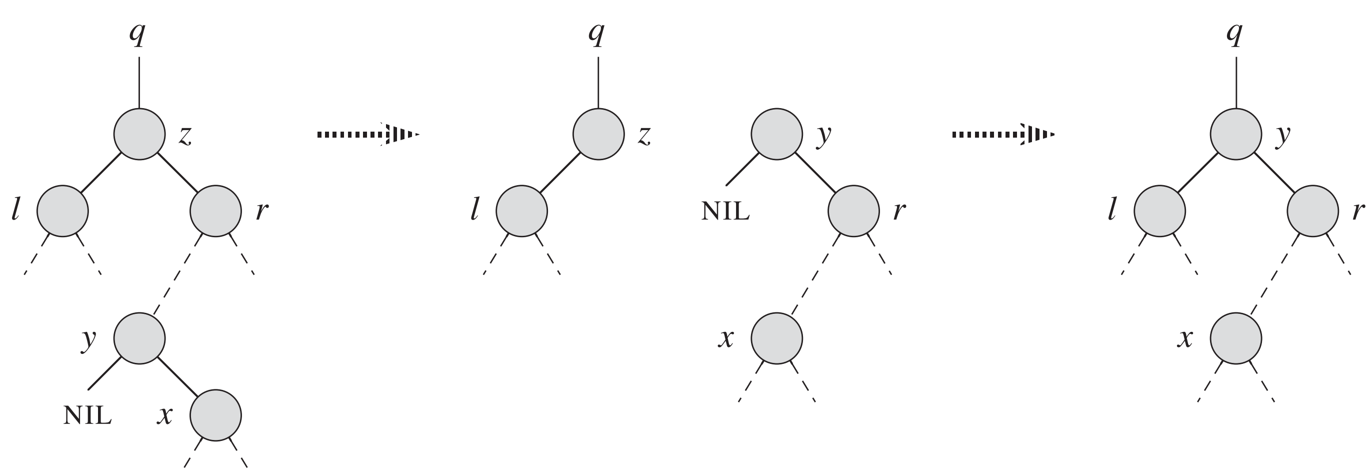

```
//Case B.2
if z.right == NULL:
   Transplant(T,z,z.left)
```

Operation: Tree-Delete (6)

- **Case B.3**: If *z* has both a left and a right child. We find *z*'s successor *y*, which lies in *z*'s right subtree and has no left child. We want to splice *y* out of its current location and have it replace *z* in the tree
    - **Case B.3.1**: If *y* is *z*'s right child, then we replace *z* by *y*, leaving *y*'s right child alone



    - **Case B.3.2**: If *y* lies within *z*'s right subtree but is not *z*'s right child. In this case, we first replace *y* by its own right child, and then we replace *z* by *y*




```
//Case B.3
y = Minimum(z.right)
if (y.parent != z):
    Transplant(T, y, y.right)
    y.right = z.right
    y.right.parent = y
Transplant(T, z, y)
y.left = z.left
y.left.parent = y
```

Operation: Tree-Delete (7)

- `Tree-Delete(z, T):` delete an existing node *z* from the search tree *T*

```
Tree-Delete(z, T):
  if z.left == NULL:
     Transplant(T,z,z.right)
  else:
     if z.right == NULL:
        Transplant(T,z,z.left)
     else:
        y = Minimum(z.right)
        if (y.parent != z):
           Transplant(T, y, y.right)
           y.right = z.right
           y.right.parent = y
        Transplant(T, z, y)
        y.left = z.left
        y.left.parent = y
```

Complexity of Operations on Binary Search Trees

Operations	Complexity
`search`	$O(h)$
`minimum`	$O(h)$
`maximum`	$O(h)$
`successor`	$O(h)$
`predecessor`	$O(h)$
`tree-insert`	$O(h)$
`Tree-delete`	$O(h)$
	<u>Remark:</u> h is the hight of a binary tree. - At worst, h can be n-1. - At best, h can be $\log(n+1)-1$.