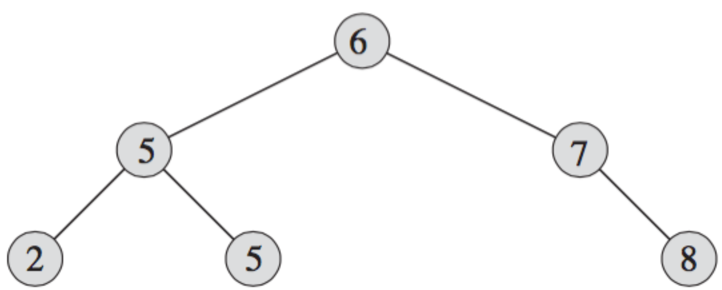
CME 2001
Data Structures and Algorithms

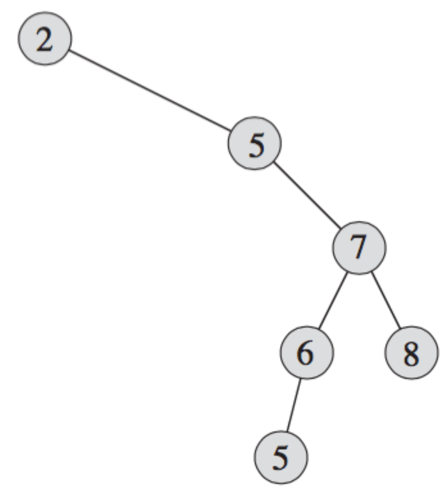
Zerrin Işık
zerrin@cs.deu.edu.tr

Binary Search Trees

What is a binary search tree?

- Important data structure for dynamic sets
  - Search, minimum, maximum, predecessor, successor, insert, delete
- Perform many dynamic-set operations in $O(h)$ time, where h = height of tree.





Representation of a binary search tree

- Represented 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 attributes :
    - *key* (and possibly other satellite 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 $\mathbf{y}.key \leq \mathbf{x}.key$.
  - If y is in right subtree of x, then $\mathbf{y}.key \geq \mathbf{x}.key$.

Inorder tree walk

- The BST property allows us to print keys in a BST in a sorted order recursively

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

Running time: $\Theta(n)$ [for a **n**-node BST]

BST - Search

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)*.
- The algorithm recursively visits nodes on a downward path from the root.
- Running time: *O(h)*, where *h* is the height of the tree.

BST – Minimum & Maximum

TREE-MINIMUM(x)

while $x.left \neq$ NIL
 $x = x.left$
return x

TREE-MAXIMUM(x)

while $x.right \neq$ NIL
 $x = x.right$
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.
- Traverse the appropriate pointers (*left*/*right*) until NIL is reached.
- Running time: $O(h)$, where h is the height of the tree.

BST – Successor & Predecessor

- The **successor** of a node ***x*** is the node ***y*** such that ***y.key*** is the smallest key **>** ***x.key*** (assume all keys are distinct!).

```
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
```

- Running time: *O(h)*, where *h* is the height of the tree.
- TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

BST – Insertion

- The procedure takes a node ***z***, with z.*key*= *v*, z.*left* = NIL, z.*right* = NIL.

- It modifies tree and some attributes of ***z*** in such a way that it inserts ***z*** into a suitable position of the tree.

- Running time: *O(h)*, where *h* is the height of the tree.

```
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
```

BST – Deletion

Deleting node ***z*** from a BST ***T*** has three cases:

1. If ***z*** has no children, just remove it.
2. If ***z*** has one child, then make that child take ***z***'s position in ***T***, elevate the child's subtree along.
3. If ***z*** has two children, then find ***z***'s successor ***y*** and replace ***z*** by ***y*** in ***T***. ***y*** must be in ***z***'s right subtree and have no left child. 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.

BST –Deletion

TRANSPLANT subroutine is used to move subtrees around ***T***.

TRANSPLANT (***T*, *u*, *v***) replaces the subtree rooted at ***u*** by the subtree rooted at ***v***:

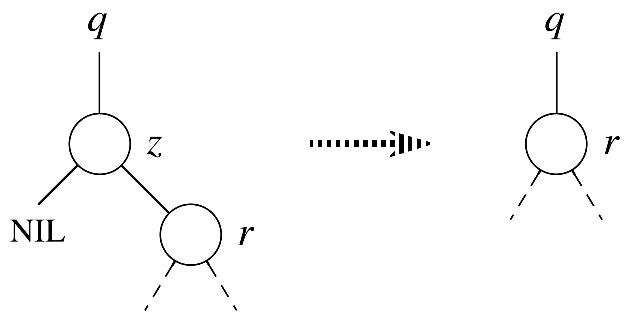
- Make ***u***'s parent become ***v***'s parent.
- ***u***'s parent gets ***v*** as either its left or right child.

TRANSPLANT(T, u, v)

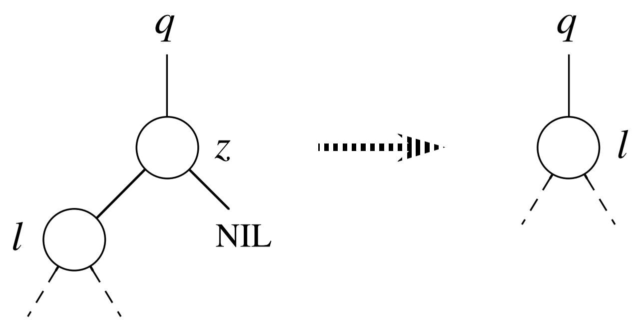
if $u.p$ == NIL
    $T.root = v$
elseif u == $u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$

BST –Deletion cases

- If ***z*** has no left child, replace ***z*** by its right child.



- If ***z*** has just one child, which is its left child, then replace ***z*** by its left child.

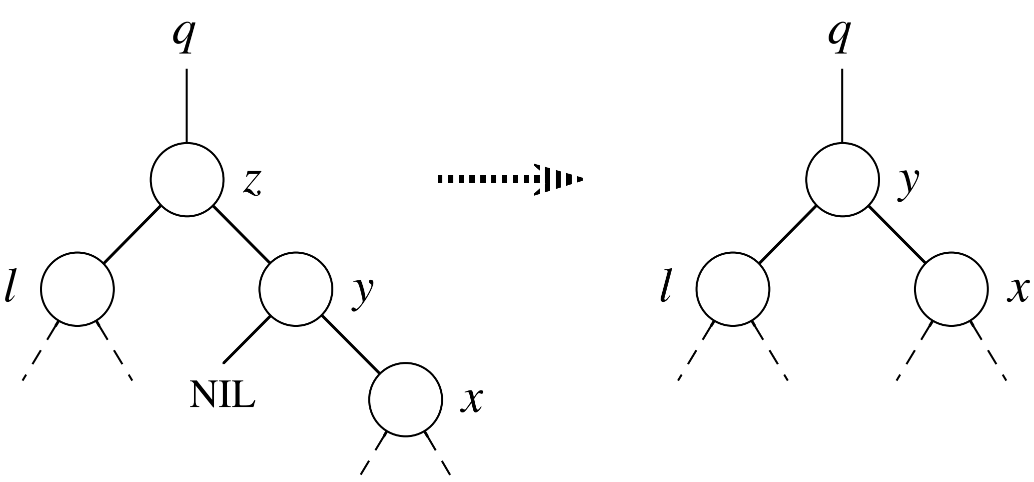


BST –Deletion cases …

- If ***z*** has two children. Find ***z***'s successor ***y***, which must lie in ***z***'s right subtree and have no left child.

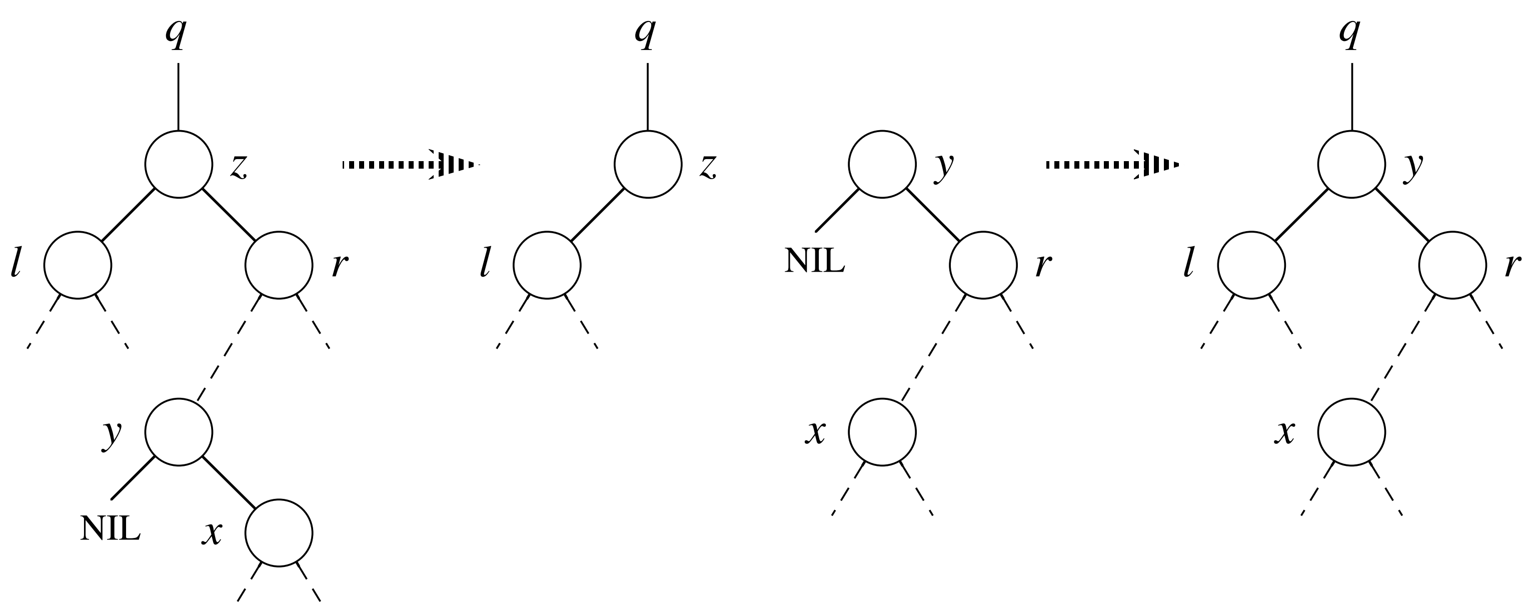
  Aim: Replace ***z*** by ***y***, splicing ***y*** out of its current location:

  1. If ***y*** is ***z***'s right child, replace ***z*** by ***y*** and leave ***y***'s right child alone.



BST –Deletion cases …

2. 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***.



BST – Deletion code

```
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
```

- Running time : *O(h)*, where *h* is the height of the tree.
- Note that everything is O(1) except the call of TREE-MINIMUM.

Next Week …

- Midterm Exam
  - Will cover all chapters of lectures.
  - Start at 9:00 - sharp.
  - No cheating paper, no calculator.