Data Structures & Algorithms

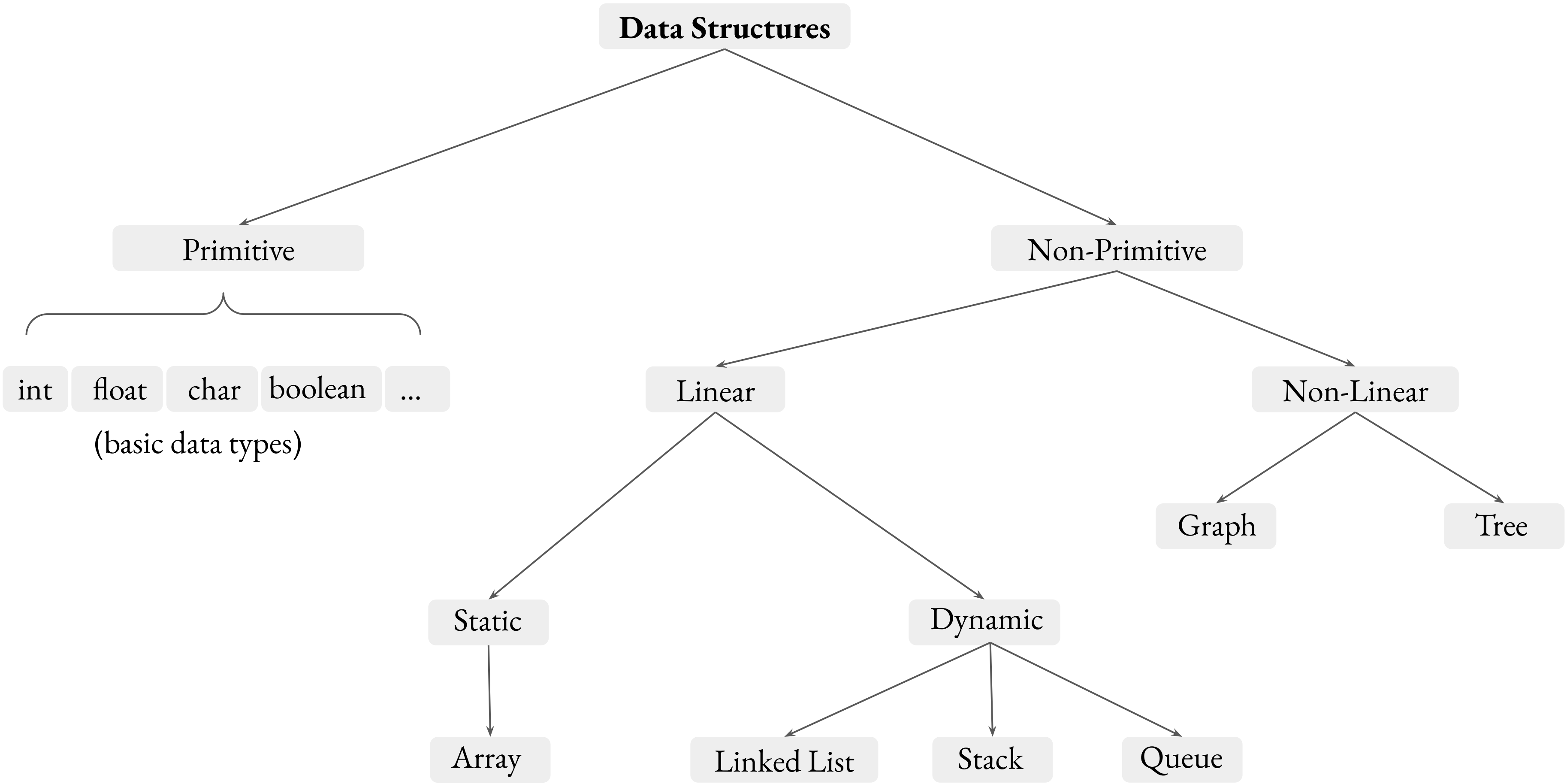
08: Binary Search Trees; Part-II

Dr Ram Prasad Krishnamoorthy

Associate Professor
School of Computing and Data Science

ram.krish@saiuniversity.edu.in

SAI UNIVERSITY

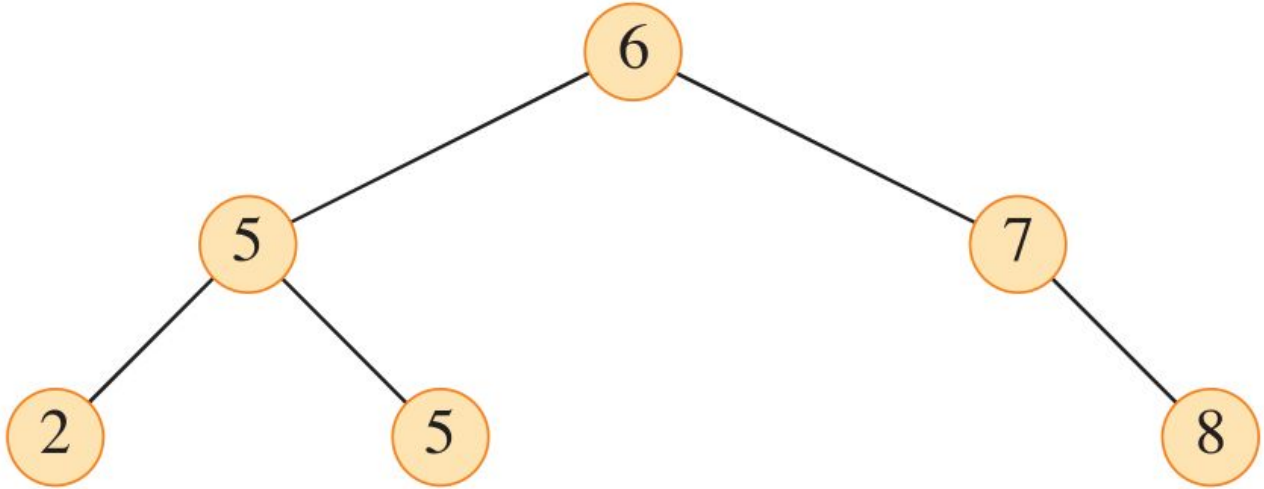

Data Structures
Primitive
Non-Primitive
int
float
char
boolean
...
(basic data types)
Linear
Non-Linear
Graph
Tree
Static
Dynamic
Array
Linked List
Stack
Queue

Binary Search Trees

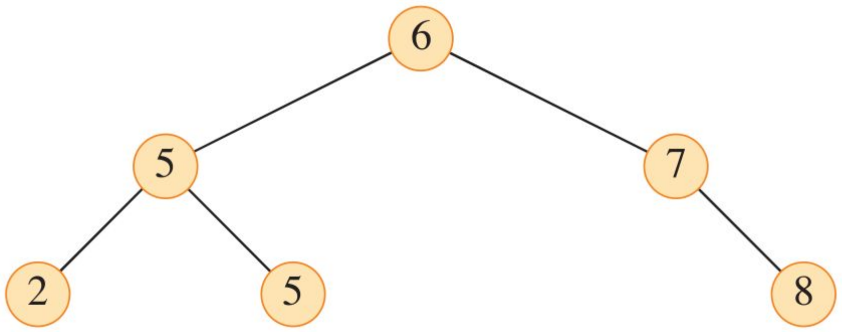
Binary Search Trees (BST) are an important data structure for dynamic sets.

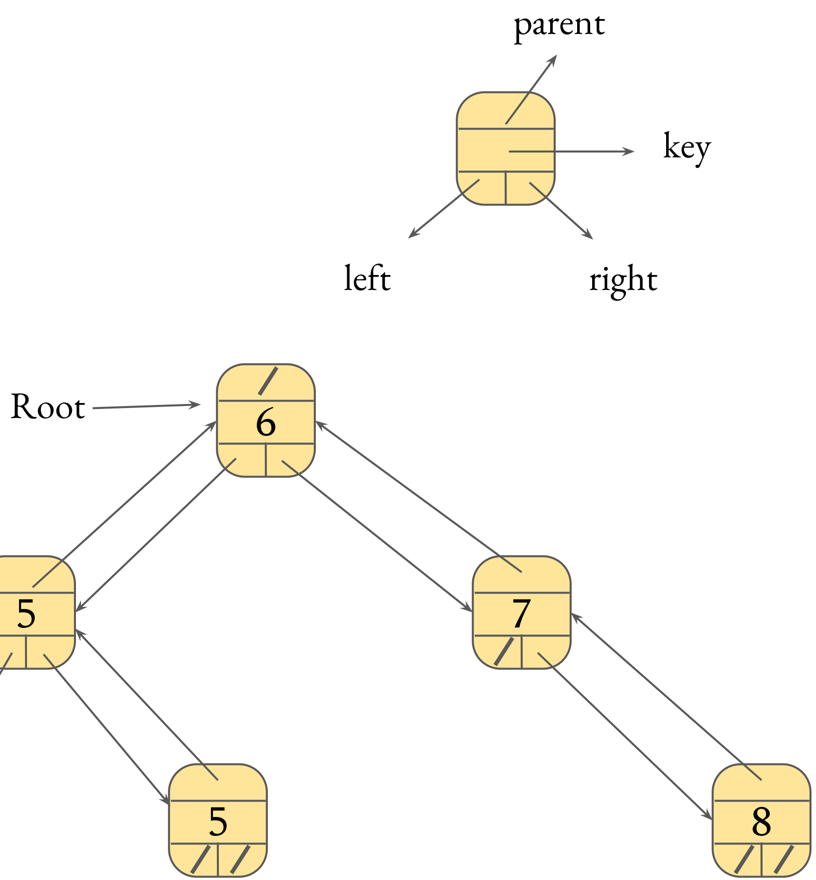
It represent a binary tree by a linked data structure in which each node is an object.

BST is also referred to as an **ordered** or **sorted binary tree**.



Binary Search Trees



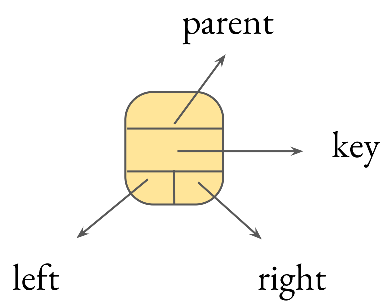


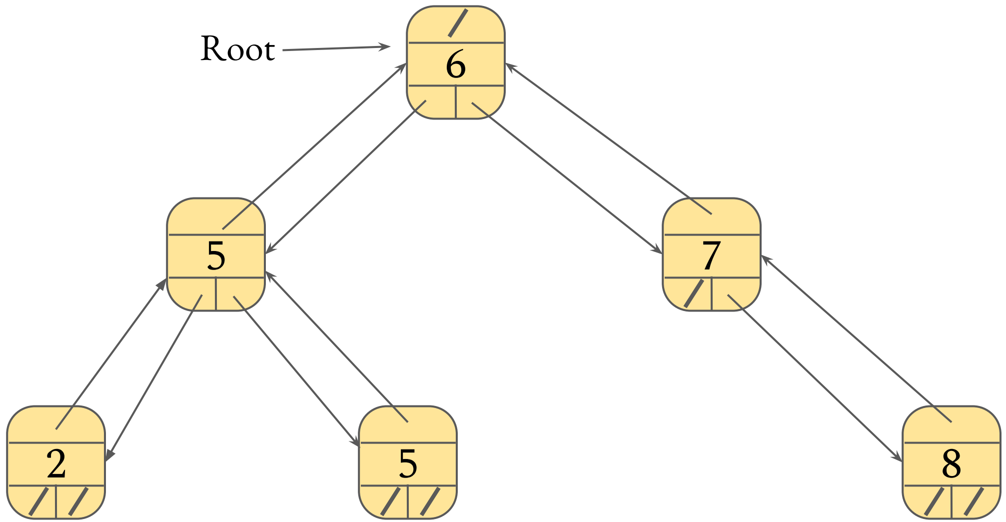
Binary Search Trees

Stored keys must satisfy the binary-search-tree property.

If y is in left subtree of x,
then **y→key** < **x→key**.

If y is in right subtree of x,
then **y→key** >= **x→key**.





Insertion

Binary Search Trees

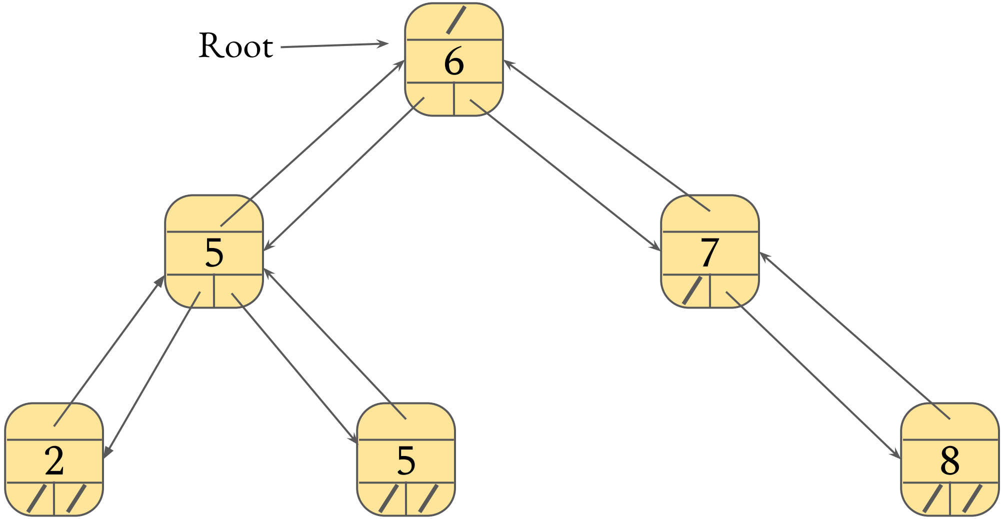
BST Insert Operation

```
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
```

Inorder Walk

Binary Search Trees

BST Inorder Traversal



INORDER-TREE-WALK(x)
  if $x \neq$ NIL
    INORDER-TREE-WALK$(x.left)$
    print $key[x]$
    INORDER-TREE-WALK$(x.right)$

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.

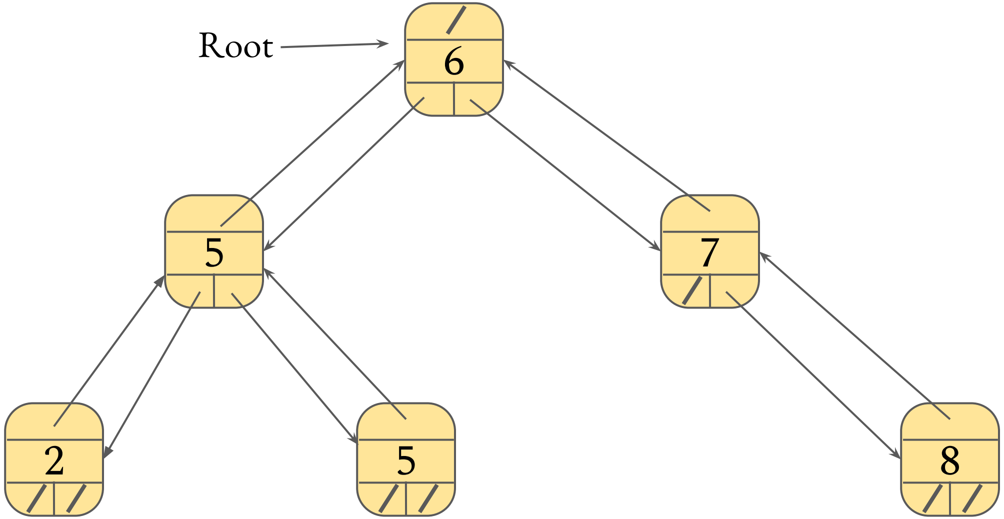
Minimum & Maximum

Binary Search Trees

Minimum and Maximum

The binary-search-tree property guarantees:

- the **minimum** key of a binary search tree is located at the **leftmost** node
- the **maximum** key of a binary search tree is located at the **rightmost** node.



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

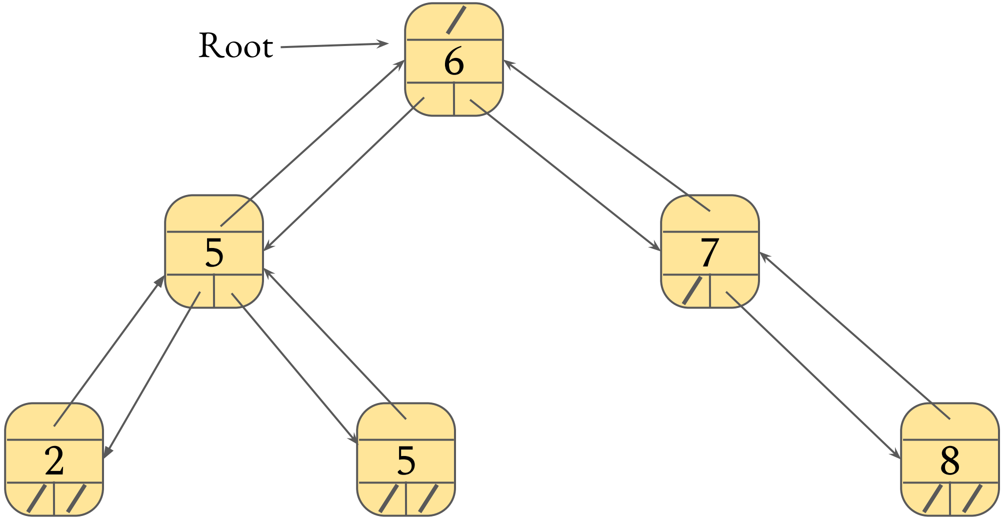
Tree Search

BINARY SEARCH TREES

Tree Search

ITERATIVE-TREE-SEARCH(x, k)

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



- Given a node, this procedure will search in that subnode.
- If we want to search in the entire tree, then start at root.

Transplant

BINARY SEARCH TREES

Transplant

TRANSPLANT(T, u, v)

if $u.p == \text{NIL}$

$\quad T.root = v$

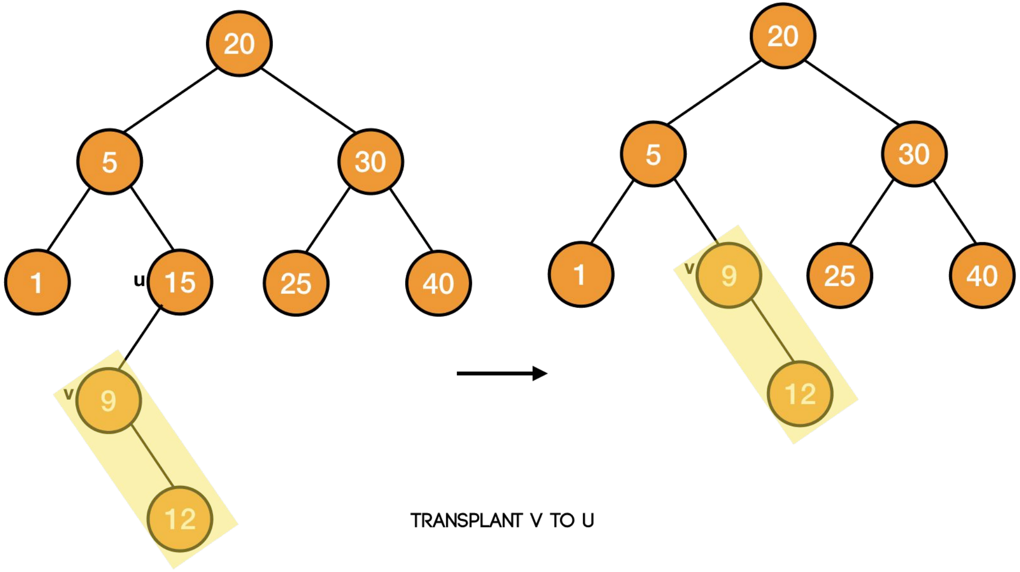
elseif $u == u.p.left$

$\quad u.p.left = v$

else $u.p.right = v$

if $v \neq \text{NIL}$

$\quad v.p = u.p$

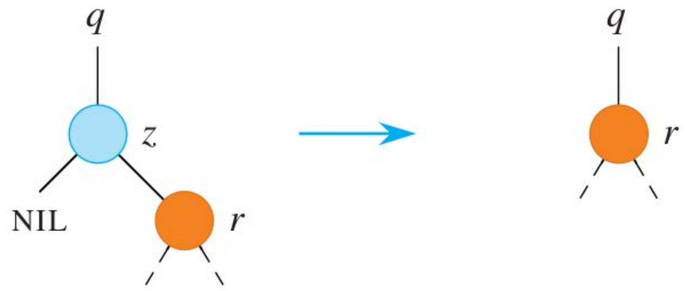


Deletion

Binary Search Trees

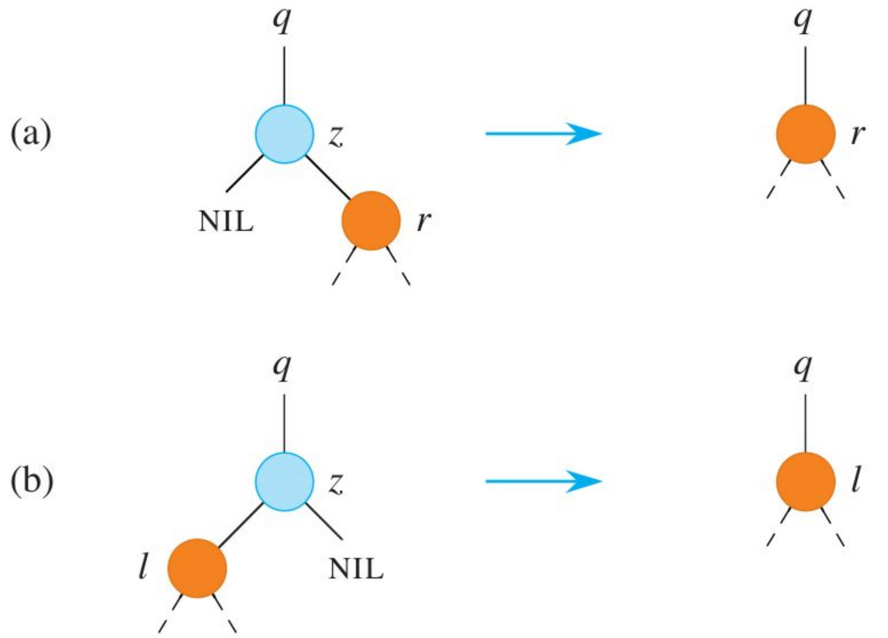
Deletion

(a)



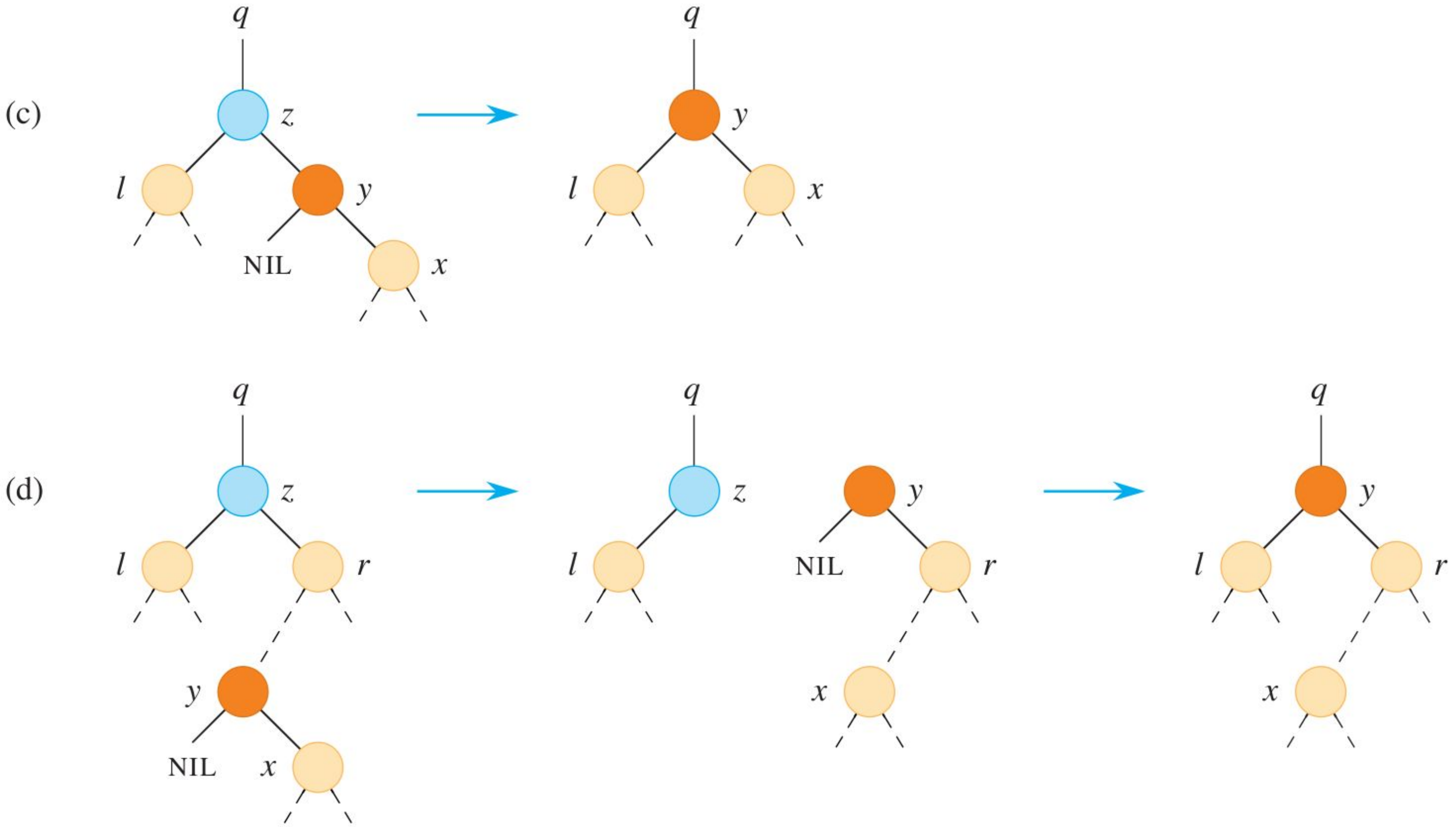
Binary Search Trees

Deletion



Binary Search Trees

Deletion



Binary Search Trees

Deletion

```
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
```