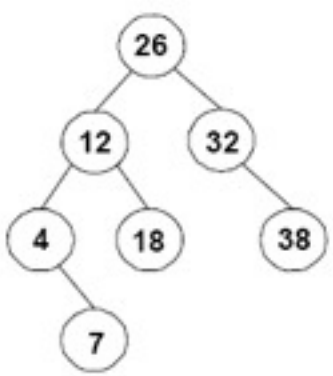
Tree Species
(Binary Trees, Binary Search Trees)

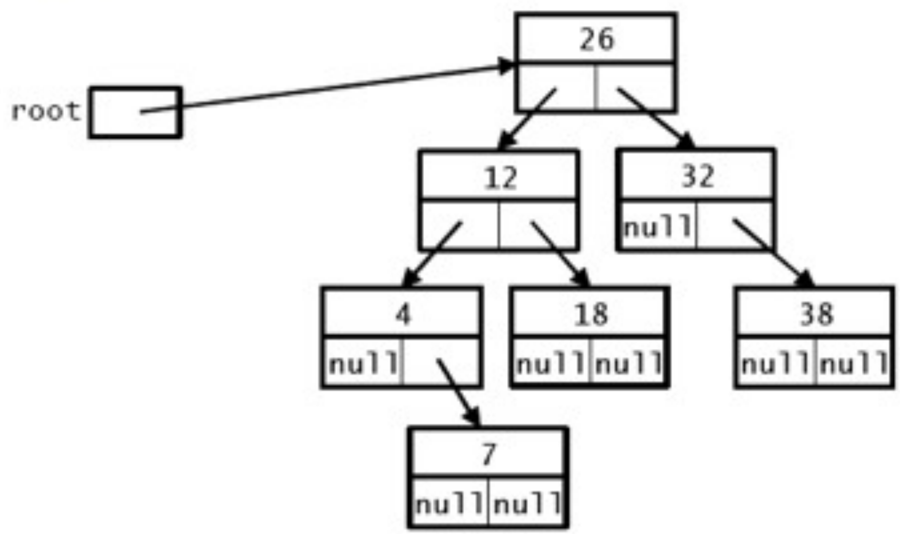
EECS 233

Previous Lecture

- Binary Tree Representation in Java

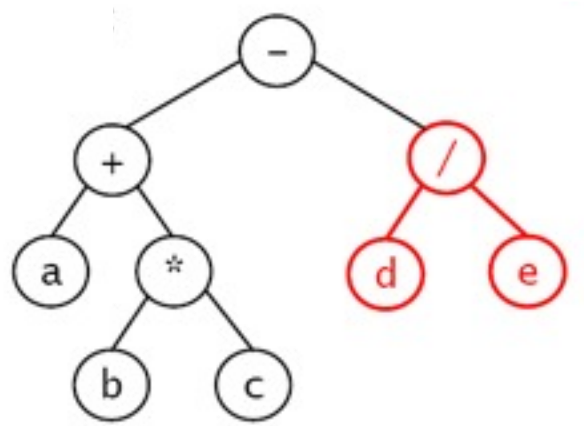
```
public class LinkedTree {
        private class Node {
                private int key;
                private String data;
                private Node left;          // reference to left child
                private Node right;         // reference to right child
                …
        }
        private Node root;
        …
}
```





Binary Trees and Expressions

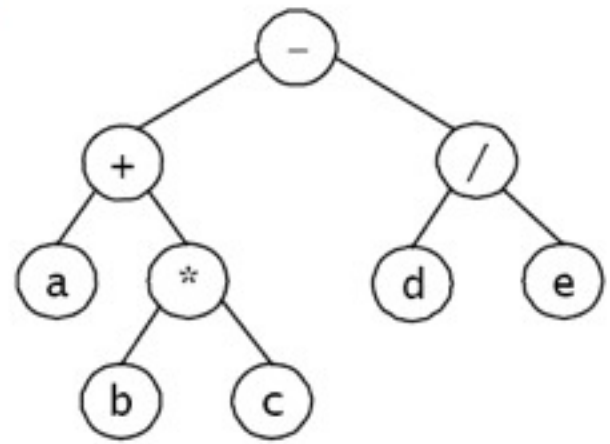
- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: +, –, *, /
  - Example expression: ((a + (b * c)) – (d / e))
  - Tree representation:



- Leaf nodes are variables or constants; interior nodes are operators.
- Because the operators are binary, either a node has two children or it has none.
- How would you generalize it?

Traversing An Expression Tree

- Inorder gives conventional infix expression.
  - print ‘(’ before the recursive call on the left subtree
  - print ‘)’ after the recursive call on the right subtree
  - for tree at right: ((a + (b * c)) – (d / e))
  - parenthesis to avoid ambiguity

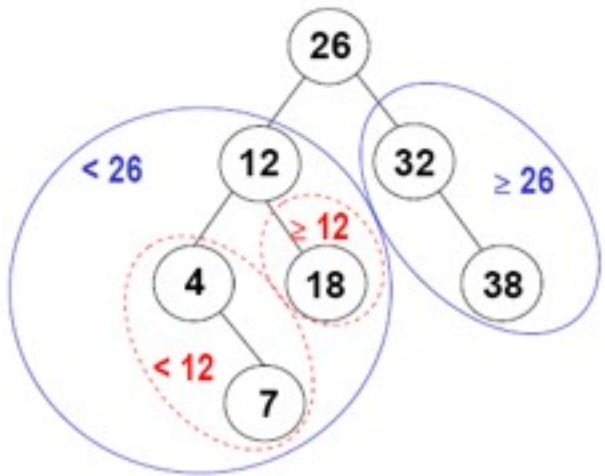


- Preorder gives functional notation.
  - print ‘(’s and ‘)’s as for inorder
  - for tree above: – (+ (a, *(b, c)), / (d, e)), or – + a * b c / d e

- Postorder gives the postfix expression.
  - for tree above: a b c * + d e / –

Binary Search Trees

- Search-tree property: for each node *k:*
  - all nodes in *k*'s left subtree are < *k*
  - all nodes in *k*'s right subtree are >= *k*



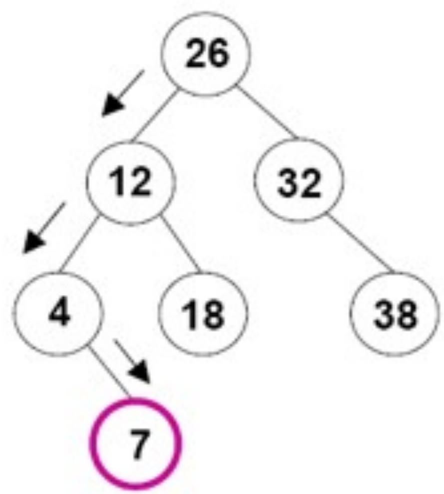
- Performing an inorder traversal of a binary search tree visits the nodes in sorted order.

Searching An Item in A Binary Search Tree

- Algorithm for searching for an item with a key *k*:

  if *k* == the root node's key, you're done
  else if *k* < the root node's key, search the left subtree
  else search the right subtree

- Example: search for 7

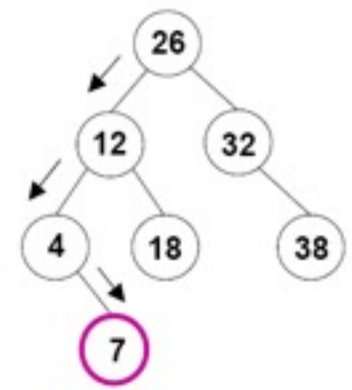


- search for 30?

Implementing Search using Recursion

```
private class Node {
        private int key;
        private String data;
        private Node left;
        private Node right;
        }
```

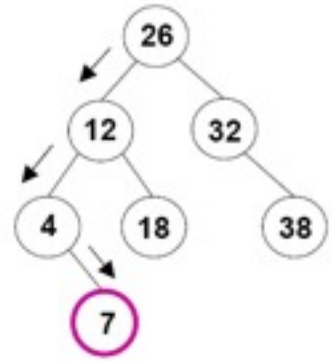
```
public class LinkedTree {
        …
        private Node root;
        public String search(int key) {
                Node n = searchTree(root, key);
                return (n == null ? null : n.data);
        }
        private Node searchTree(Node root, int key) {
                //
                if (root == null)
                        return null;
                else if (key == root.key)
                        return root
                else if (key < root.key)
                        return searchTree(root.left, key);
                else
                        return searchTree(root.right, key);
        }
}
```



- The search() method makes the initial call of the recursive searchTree() method, invoking it on the root of the entire tree.

Implementing Search using Iteration

```
public class LinkedTree {
    …
    private Node root;
    public String search(int key) {
        Node n = searchTree(root, key);
        return (n == null ? null : n.data);
    }
    private Node searchTree(Node root, int key) {
        Node trav = root;
        while (trav != null) {
            if (key == trav.key)
                return trav;
            else if (key < root.key)
                trav = trav.left;
            else
                trav = trav.right;
        }
        return null;
    }
}
```

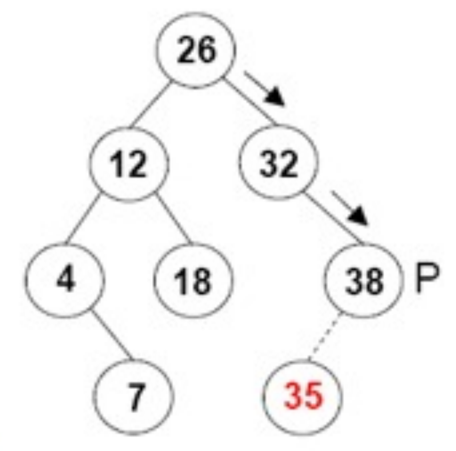


Inserting An Item in A Binary Search Tree

- We want to insert an item whose key is *k*.
- First, we find the node P that will be the parent of the new node:
  - we traverse the tree as if we were searching for *k*, but we don't stop if we find it – we continue until we can't go any further
- Next, we add the new node to the tree:

  if *k* < P's key, make the node P's left child

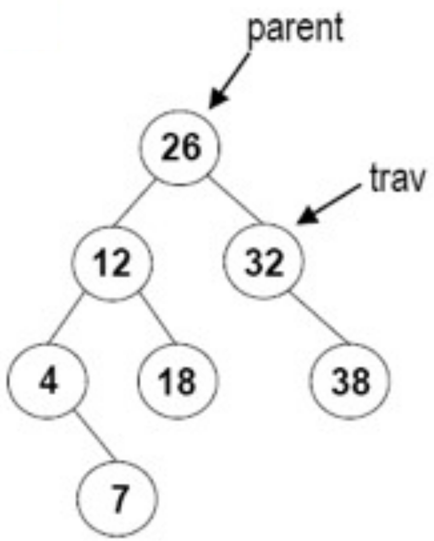
  else make the node P's right child



- *Special case:* if the tree is empty, make the new node the root of the tree

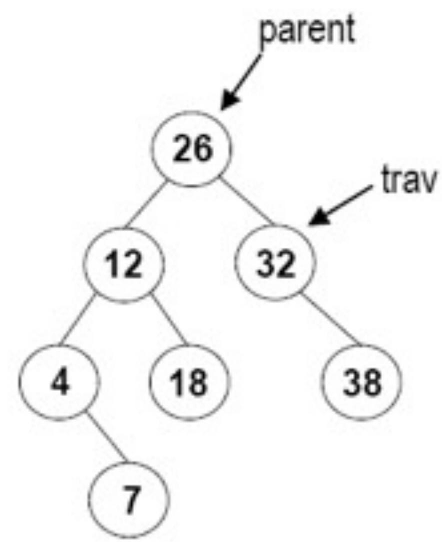
Implementing Insert

- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
  - trav: performs the traversal down to the point of insertion
  - parent: stays one behind trav
  - when we're done with the traversal:
    - ✓ trav will be null;
    - ✓ parent will point at the a leaf node that will be the parent of the new node



Let's Do An Exercise

```
public void insert(int key, String data) {
        // Find the parent of the new node.
        Node parent = null;
        Node trav = root;
        while (trav != null) {
                parent = trav;
                if (key < trav.key)
                        trav = trav.left;
                else
                        trav = trav.right;
        }
        // Insert the new node.
        Node newNode = new Node(key, data);
        if (parent == null)      // the tree was empty
                root = new Node(key,data);
        else if (key < parent.key)
                parent.left = new Node(key,data);
        else
                parent.right = new Node(key,data);
}
```

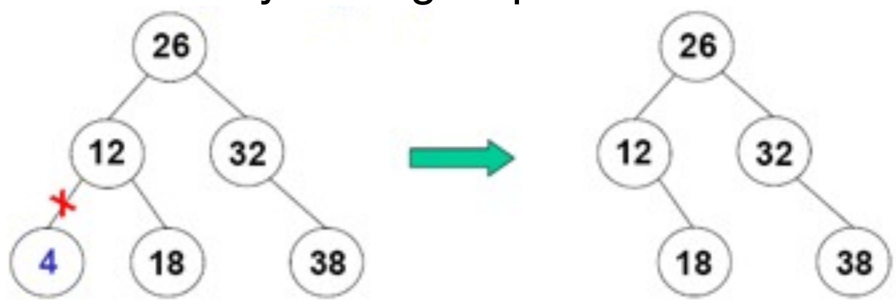


Insert 35?

Deleting An Item from A Binary Search Tree

- Three cases for deleting a node *x*
- **Case 1:** *x* has no children.
  - Remove *x* from the tree by setting its parent's reference to null.

ex: delete 4

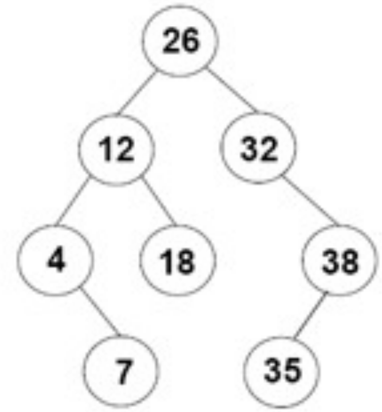


- **Case 2:** *x* has one child.
  - Take the parent's reference to *x* and make it refer to *x*'s child.

ex: delete 12

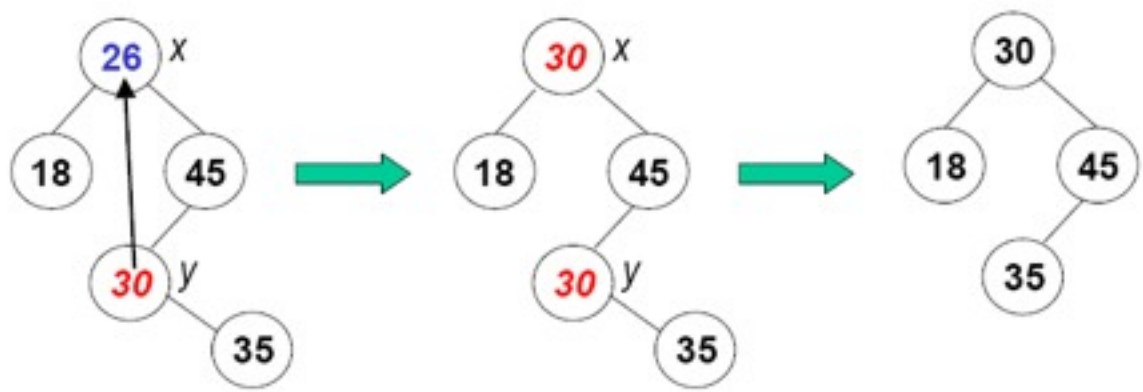
Deleting An Item from A Binary Search Tree

- **Case 3:** *x* has two children
  - we can't just delete *x*.
  - instead, we replace *x* with a node from elsewhere in the tree, and we must choose the replacement carefully



- Which node could replace?
  - replace *x* with the smallest node in *x*'s right subtree—call it *y*.
  - *y* will either be a leaf node or will have one right child.
  - after copying *y*'s item into *x*, we delete *y* using case 1 or 2.

ex:
delete 26



Implementing Delete

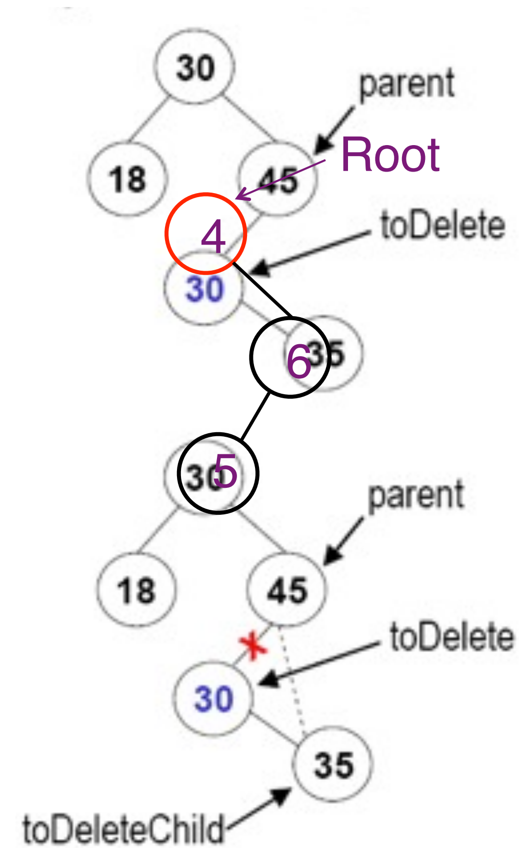
```
public String delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }
    // Delete the node (if any) and return the removed item.
    if (trav == null) // no such key
        return null;
    else {
        String removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

- This method uses a helper method to delete the node.

Implementing Delete

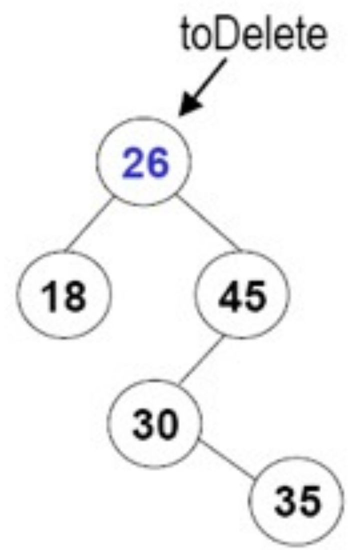
(Case 1 and 2)

```
private void deleteNode(Node toDelete, Node parent) {
        if (toDelete.left == null || toDelete.right == null) {
                // Cases 1 and 2
                Node toDeleteChild = null;
                if (toDelete.left != null)
                        toDeleteChild = toDelete.left;
                else
                        toDeleteChild = toDelete.right;
                // both Cases are included. In case 1 toDeleteChild==null
                if (toDelete == root)
                        root = toDeleteChild;
                else if (toDelete.key < parent.key)
                        parent.left = toDeleteChild;
                else
                        parent.right = toDeleteChild;
                }
        } else {  // case 3
                ...
        }
```



Implementing Delete (Case 3)

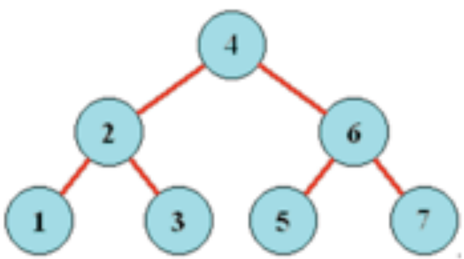
```
private void deleteNode(Node toDelete, Node parent) {
        if (toDelete.left == null || toDelete.right == null) { // case 1 and 2
                ...
        } else { // case 3
                // Get the smallest item in the right subtree.
                // or get a largest in the left (flip a coin)
                Node replacementParent = toDelete;
                Node replacement = toDelete.right;
                while (replacement != null) {
                                replacementParent = replacement;
                                replacement = replacement.left;
                }
                // Replace toDelete's key and data
                toDelete.key = replacement.key;
                toDelete.data = replacement.data;

                // Recursively delete the replacement item's old node.
                deleteNode(replacement, replacementParent);
        }
```

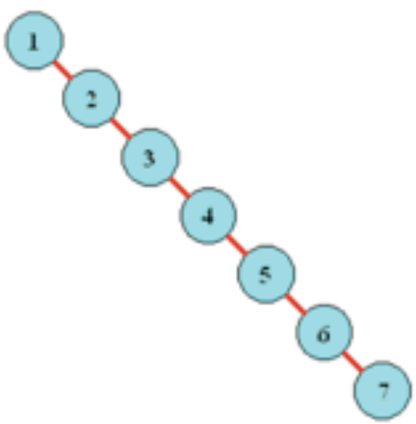


Problem-of-the-week: The highlighted lines shows this method copies the key and data fields (which could be expensive). A better method?

Efficiency of Binary Search Tree

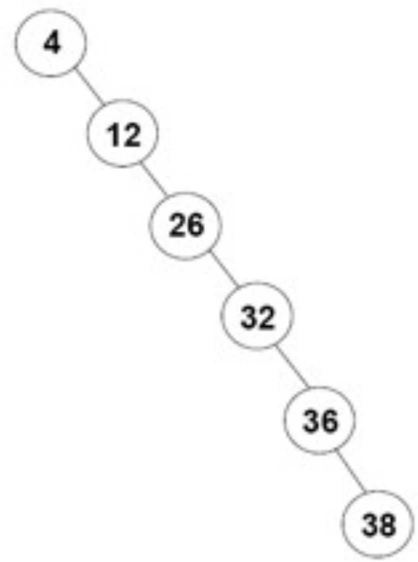
- The three key operations (search, insert, and delete) all have the same time complexity.
- Insert and delete both involve a search followed by a constant number of additional operations
- Time complexity of searching a binary search tree:
  - best case: $O(1)$
  - worst case: $O(h)$, where h is the height of the tree
  - average case: $O(h)$
- What is the height of a tree containing n items?
  - It depends!





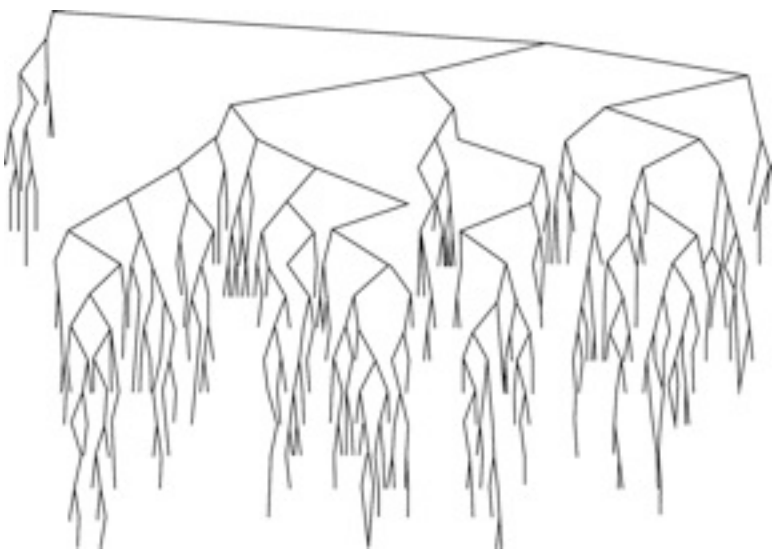
Balanced Binary Search Tree

- If a tree is not balanced, for example, an extreme case:
  - height = n – 1, and
  - worst-case time complexity = O(n)
  - Does it happen often? When?

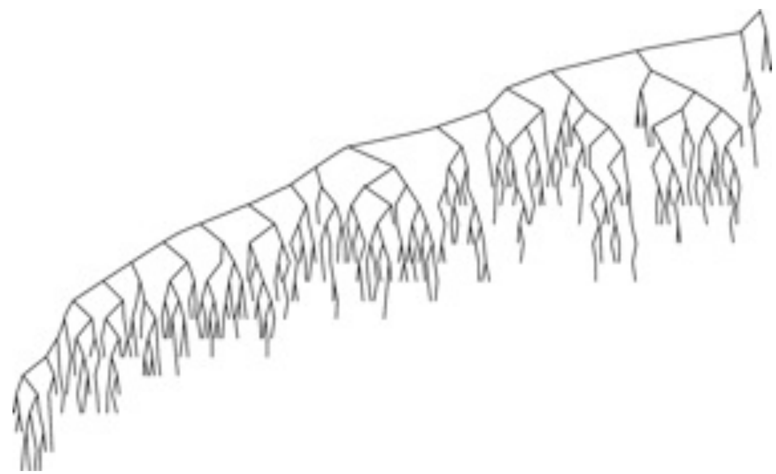


- A tree is *balanced* if, for each node, the node's subtrees
  - have heights that differ by at most 1
  - For a balanced tree with n nodes:
    - ✓ height = $O(\log_2 n)$:
      - 1 node at level 0
      - 2 nodes at level 1 …
      - 2^L nodes at level L …
    - ✓ worst-case time complexity = $O(\log_2 n)$

Balance of A Randomly Generated Tree



Random binary search tree



Same tree after a large
number of random inserts/removes

Random insertion – anywhere; random removal – from the right subtree