Day 1

Chapter 6

Binary Trees

Objectives

Discuss the following topics:

- Trees, Binary Trees, and Binary Search Trees
- Implementing Binary Trees
- Searching a Binary Search Tree
- Tree Traversal
- Insertion
- Deletion

Objectives (continued)

Discuss the following topics:

- Balancing a Tree
- Self-Adjusting Trees
- Heaps
- Polish Notation and Expression Trees
- Case Study: Computing Word Frequencies

Trees, Binary Trees, and Binary Search Trees

- A **tree** is a data type that consists of **nodes** and **arcs**
- These trees are depicted upside down with the root at the top and the **leaves** (**terminal nodes**) at the bottom
- The **root** is a node that has no parent; it can have only child nodes
- Leaves have no children (their children are null)

Trees, Binary Trees, and Binary Search Trees (continued)

- Each node has to be reachable from the root through a unique sequence of arcs, called a **path**
- The number of arcs in a path is called the **length** of the path
- The **level** of a node is the length of the path from the root to the node plus 1, which is the number of nodes in the path
- The **height** of a nonempty tree is the maximum level of a node in the tree

Trees, Binary Trees, and Binary Search Trees (continued)

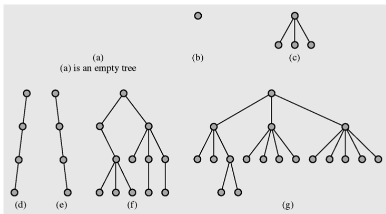


Figure 6-1 Examples of trees

Trees, Binary Trees, and Binary Search Trees (continued)

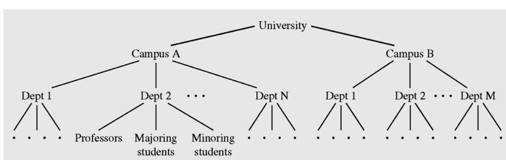
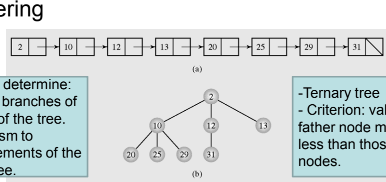


Figure 6-2 Hierarchical structure of a university shown as a tree

Trees, Binary Trees, and Binary Search Trees (continued)

- An **orderly tree** is where all elements are stored according to some predetermined criterion of ordering

We need to determine:
-Number of branches of each node of the tree.
-A mechanism to tranform elements of the list to the tree.



-Ternary tree
- Criterion: values in father node must be less than those in child nodes.

Figure 6-3 Transforming (a) a linked list into (b) a tree

Trees, Binary Trees, and Binary Search Trees (continued)

- A **binary tree** is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child

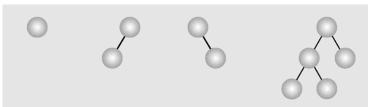


Figure 6-4 Examples of binary trees

Trees, Binary Trees, and Binary Search Trees (continued)

- In a **complete binary tree**, all nonterminal nodes have both their children, and all leaves are at the same level
- A **decision tree** is a binary tree in which all nodes have either zero or two nonempty children

Trees, Binary Trees, and Binary Search Trees (continued)

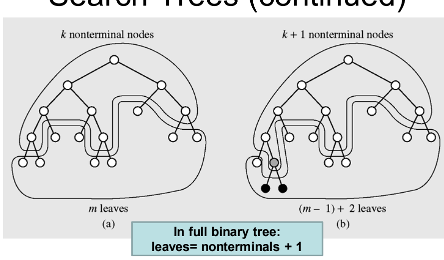


Figure 6-5 Adding a leaf to tree (a), preserving the relation of the number of leaves to the number of nonterminal nodes (b)

Trees, Binary Trees, and Binary Search Trees (continued)

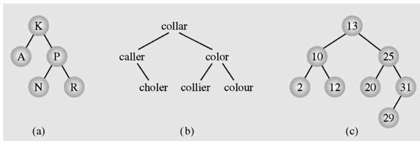


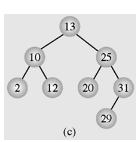
Figure 6-6 Examples of binary search trees

Implementing Binary Trees

- Binary trees can be implemented in at least two ways:
  - As arrays
  - As linked structures
- To implement a tree as an array, a node is declared as an object with an information field and two **"reference"** fields

Implementing Binary Trees (continued)

Index	Info	Left	Right
0	13	4	2
1	31	6	–1
2	25	7	1
3	12	–1	–1
4	10	5	3
5	2	–1	–1
6	29	–1	–1
7	20	–1	–1



Structure of each element in the array:
{ int info;
int left;
int right;
}

Figure 6-7 Array representation of the tree in Figure 6.6c

Implementing Binary Trees (continued)

```
/************************  IntBSTNode.java  **************************
 *                 binary search tree of integers
 */

public class IntBSTNode {
    protected int key;
    protected IntBSTNode left, right;
    public IntBSTNode() {
        left = right = null;
    }
    public IntBSTNode(int el) {
        this(el,null,null);
    }
    public IntBSTNode(int el, IntBSTNode lt, IntBSTNode rt) {
        key = el; left = lt; right = rt;
    }
}
```

Figure 6-8 Implementation of a generic binary search tree

Implementing Binary Trees (continued)

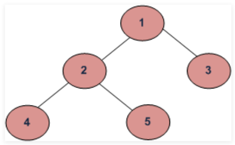
```
/************************  IntBST.java  **************************
 *                 binary search tree of integers
 */

public class IntBST {
    protected IntBSTNode root;
    public IntBST() {
        root = null;
    }
    protected void visit(IntBSTNode p) {
        System.out.print(p.key + " ");
    }
    public IntBSTNode search(IntBSTNode p, int el) {
        return search(p, root);
    }
    public IntBSTNode search(IntBSTNode p, int el) { . . . } // Figure 6.9
    public void breadthFirst() { . . . }                     // Figure 6.10
    public void preorder() {
        preorder(root);
    }
    protected void preorder(IntBSTNode p) { . . . }          // Figure 6.11
    public void inorder() {
```

Figure 6-8 Implementation of a generic binary search tree (continued)

Implementing Binary Trees (continued)

```
        inorder(root);
    }
    protected void inorder(IntBSTNode p) { . . . }            //  Figure 6.11
    public void postorder() {
        postorder(root);
    }
    protected void postorder(IntBSTNode p) { . . . }          //  Figure 6.11
    public void iterativePreorder() { . . . }                 //  Figure 6.15
    public void iterativeInorder() { . . . }                  //  Figure 6.17
    public void iterativePostorder() { . . . }                //  Figure 6.16
    public void MorrisInorder() { . . . }                     //  Figure 6.20
    public void insert(int el) { . . . }                      //  Figure 6.23
    public void deleteByMerging(int el) { . . . }             //  Figure 6.29
    public void deleteByCopying(int el) { . . . }             //  Figure 6.32
    public void balance (int date[], int first, int last)     //  Section 6.7
. . . . . . . . . . . . . . . . . . .
}
```

Figure 6-8 Implementation of a generic binary search tree (continued)



Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Searching a Binary Search Tree

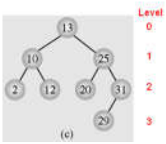
```
public IntBSTNode search(IntBSTNode p, int el) {
    while (p != null)
        if (el == p.key)
             return p;
        else if (el < p.key)
             p = p.left;
        else p = p.right;
    return null;
}
```

Figure 6-9 A function for searching a binary search tree

Searching a Binary Search Tree (continued)

- The **internal path length (IPL)** is the sum of all path lengths of all nodes
- It is calculated by summing $\Sigma(i - 1)l_i$ over all levels *i,* where l_i is the number of nodes on level *l*
- A depth of a node in the tree is determined by the path length
- An average depth, called an **average path length***,* is given by the formula IPL/*n,* which depends on the shape of the tree

How to calculate IPL



The level of a node is the length from the root to this node

Level	Number of node	Sum of path lengths
0	1	0 * 1 =0
1	2	1*2= 2
2	4	2*4= 8
3	1	3*1=3

IPL= 0 + 2 +8 +3 = 13
IPL$_{avg}$= 13/8

IPL$_{avg}$ represents the average number of comparisons in the search operation

Tree Traversal

- **Tree traversal** is the process of visiting each node in the tree exactly one time
- **Breadth-first traversal** is visiting each node starting from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left)

Breadth-First Traversal

```
public void breadthFirst() {
    IntBSTNode p = root;
    Queue queue = new Queue;
    if (p != null) {
        queue.enqueue(p);
        while (!queue.isempty()) {
            p = (IntBSTNode) queue.dequeue();
            visit(p);
            if (p.left != null)
                 queue.enqueue(p.left);
            if (p.right != null)
                 queue.enqueue(p.right);
        }
    }
}
```

Figure 6-10 Top-down, left-to-right, breadth-first traversal implementation

Depth-First Traversal

- **Depth-first traversal** proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right)
  - V — Visiting a node
  - L — Traversing the left subtree
  - R — Traversing the right subtree

Depth-First Traversal (continued)

```
protected void preorder(IntBSTNode p) {
    if (p != null) {
        visit(p);
        preorder(p.left);
        preorder(p.right);
    }
}
protected void inorder(IntBSTNode p) {
    if (p != null) {
        inorder(p.left);
        visit(p);
        inorder(p.right);
    }
}
protected void postorder(IntBSTNode p) {
    if (p != null) {
        postorder(p.left);
        postorder(p.right);
        visit(p);
    }
}
```

Figure 6-11 Depth-first traversal implementation

Depth-First Traversal (continued)

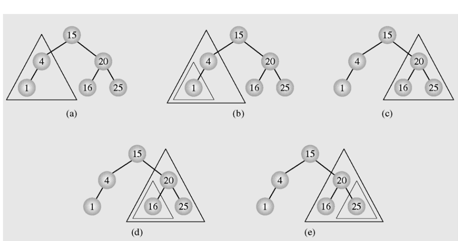


Figure 6-12 Inorder tree traversal

Depth-First Traversal (continued)

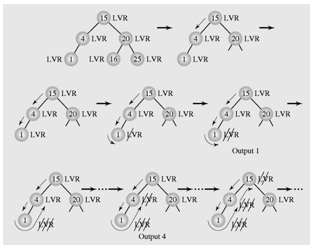


Figure 6-13 Details of several of the first steps of inorder traversal

Depth-First Traversal (continued)

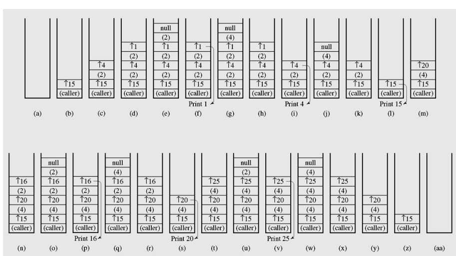


Figure 6-14 Changes in the run-time stack during inorder traversal

Depth-First Traversal (continued)

```
public void iterativePreorder() {
    IntBSTNode p = root;
    Stack travStack = new Stack();
    if (p != null) {
        travStack.push(p);
        while (!travStack.isEmpty()) {
            p = (IntBSTNode) travStack.pop();
            visit(p);
            if (p.right != null)
                travStack.push(p.right);
            if (p.left  != null)        // left child pushed after right
                travStack.push(p.left);// to be on the top of the
                                       // stack;
        }
    }
}
```

Figure 6-15 A nonrecursive implementation of preorder tree traversal

Stackless Depth-First Traversal

- **Threads** are references to the predecessor and successor of the node according to an inorder traversal
- Trees whose nodes use threads are called **threaded trees**
- The **left reference** is either a reference to the left child or to the **predecesso**r. Analogously, the **right reference** refers either to the right subtree or to the **successor**.
- Threaded trees can used for preorder, inorder and post order traversal.

Stackless Depth-First Traversal (continued)

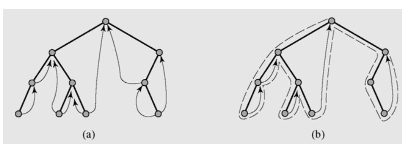


Figure 6-18 (a) A threaded tree and (b) an inorder traversal's path in a threaded tree with right successors only

Stackless Depth-First Traversal (continued)

```
/********************  IntThreadedTreeNode.java  **********************
 *                binary search threaded tree of integers
 */

class IntThreadedTreeNode {
    protected int key;
    protected boolean sucessor;
    protected IntThreadedTreeNode left, right;
    public IntThreadedTreeNode() {
        left = right = null; sucessor = false;
    }
    public IntThreadedTreeNode(int el) {
        this(el,null,null);
    }
    public IntThreadedTreeNode(int el, IntThreadedTreeNode lt,
                                       IntThreadedTreeNode rt) {
        key = el; left = lt; right = rt; sucessor = false;
    }
}
```

Figure 6-19 Implementation of the threaded tree and the inorder traversal of a threaded tree

Stackless Depth-First Traversal (continued)

```
/************************  IntThreadedTree.java  *********************
 *                binary search threaded tree of integers
 */

public class IntThreadedTree {
    private IntThreadedNode root;
    public IntThreadedTree() {
        root = null;
    }
    protected void visit(intThreadedTreeNode p) {
        System.out.print(p.key + " ");
    }
    protected void threadedInorder() {
        IntThreadedNode prev, p = root;
        if (p != null) {                  // process only nonempty trees;
            while (p.left != null)        // go to the leftmost node;
                p = p.left;
            while (p != null) {
                visit(p);
                prev = p;
```

Figure 6-19 Implementation of the threaded tree and the inorder traversal of a threaded tree (continued)

Stackless Depth-First Traversal (continued)

```
                p = p.right;            // go to the right node and only
                if (p != null && !prev.sucessor)// if it is a descendant
                    while (p.left != null)// go to the leftmost node,
                        p = p.left;       // otherwise visit the
                                          // successor;
            }
        }
    }
    public void threadedInsert(int el) {
                                          //  Figure 6.24
    }
}
```

Figure 6-19 Implementation of the threaded tree and the inorder traversal of a threaded tree (continued)

Traversal Through Tree Transformation

This algorithm will tranform a BST to a singly linked list the restore the SLL to the original BST ➔ No recursion, no temporary memory is needed

```
public void MorrisInorder() {
    IntBSTNode p = root, tmp;
    while (p != null)
        if (p.left == null) {
            visit(p);
            p = p.right;
        }
        else {
            tmp = p.left;
            while (tmp.right != null && // go to the rightmost node of
                   tmp.right != p)  // the left subtree or
                tmp = tmp.right;    // to the temporary parent of p;
            if (tmp.right == null) {// if 'true' rightmost node was
                tmp.right = p;      // reached, make it a temporary
                p = p.left;         // parent of the current root,
            }
            else {                  // else a temporary parent has been
                visit(p);           // found; visit node p and then cut
                tmp.right = null;   // the right pointer of the current
                p = p.right;        // parent, whereby it ceases to be
            }                       // a parent;
        }
}
```

Figure 6-20 Implementation of the Morris algorithm for inorder traversal

Traversal Through Tree Transformation

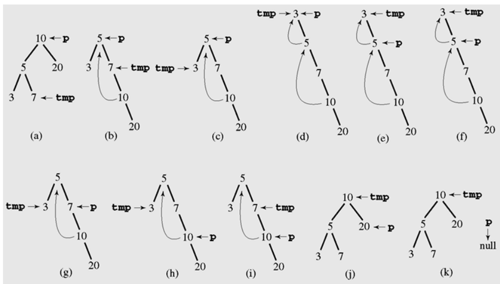


Figure 6-21 Tree traversal with the Morris method

Data Structures and Algorithms in Java 37

Day 2

Data Structures and Algorithms in Java 38

Insertion in BST

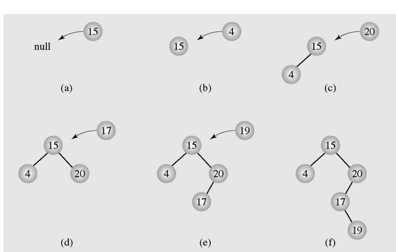


Figure 6-22 Inserting nodes into binary search trees
This operation was presented in Discrete Mathematic 2

Data Structures and Algorithms in Java 39

Insertion in BST (continued)

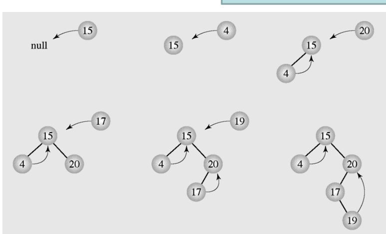
```
public void insert(int el) {
    IntBSTNode p = root, prev = null;
    while (p != null) {  // find a place for inserting new node;
        prev = p;
        if (p.key < el)
             p = p.right;
        else p = p.left;
    }
    if (root == null)    // tree is empty;
         root = new IntBSTNode(el);
    else if (prev.key < el)
         prev.right = new IntBSTNode(el);
    else prev.left  = new IntBSTNode(el);
}
```

Figure 6-23 Implementation of the insertion algorithm in BST

Insertion in Threaded BST

```
public class IntThreadedNode{
    protected int key;
    protected boolean hasSuccessor;
    protected ThreadedNode left, right;
    ....
}
```

Each node contains a field to mark whether it has a successor or not and the right reference of each node will link to the node that it will succeed.



Insertion in Threaded BST

```
public void threadedInsert(int el) {
    IntThreadedNode newNode = new IntThreadedNode(el);
    if (root == null) {              // tree is empty
          root = newNode;
          return;
    }
    IntThreadedNode p = root, prev = null;
    while (p != null) {              // find a place to insert newNode;
         prev = p;
         if (el < p.key)
              p = p.left;
         else if (!p.hasSuccessor)   // go to the right only if it is
              p = p.right;           // a descendant, not a successor;
         else break;                 // don't follow successor link;
```

Figure 6-24 Implementation of the algorithm to insert nodes into a threaded tree

Insertion in Threaded BST

```
    }
    if (el < prev.key) {               // if newNode is left child of
         prev.left  = newNode;         // its parent, the parent
         newNode.hasSuccessor = true;// also becomes its successor;
         newNode.right = prev;
    }
    else if (prev.hasSuccessor) {    // if parent of the newNode
         newNode.hasSuccessor = true;// is not the rightmost node,
         prev.hasSuccessor = false;  // make parent's successor
         newNode.right = prev.right; // newNode's successor,
         prev.right = newNode;
    }
    else prev.right = newNode;       // otherwise it has no successor;
}
```

Figure 6-24 Implementation of the algorithm to insert nodes into a threaded tree (continued)

Data Structures and Algorithms in Java 43

Insertion in Threaded BST

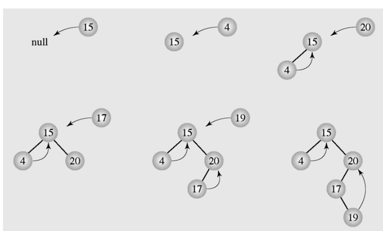


Figure 6-25 Inserting nodes into a threaded tree

Data Structures and Algorithms in Java 44

Deletion

- There are three cases of deleting a node from the binary search tree:
  - The node is a leaf; it has no children
  - The node has one child
  - The node has two children

Data Structures and Algorithms in Java 45

Deletion (continued)

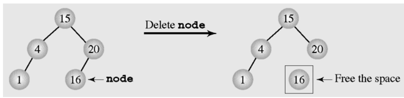


Figure 6-26 Deleting a leaf

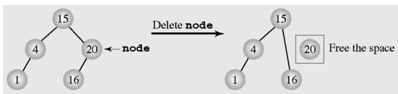


Figure 6-27 Deleting a node with one child

Deletion by Merging

- Making one tree out of the two subtrees of the node and then attaching it to the node's parent is called **deleting by merging**

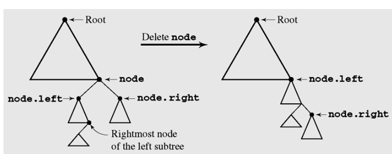


Figure 6-28 Summary of deleting by merging

Deletion by Merging (continued)

```
public void deleteByMerging(int el) {
    IntBSTNode tmp, node, p = root, prev = null;
    while (p != null && p.key != el) { // find the node p
         prev = p;                     // with element el;
         if (p.key < el)
              p = p.right;
         else p = p.left;
    }
    node = p;
    if (p != null && p.key == el) {
         if (node.right == null) // node has no right child: its left
              node = node.left;  // child (if any) is attached to its
                                 // parent;
         else if (node.left == null) // node has no left child: its right
              node = node.right; // child is attached to its parent;
         else {                  // be ready for merging subtrees;
              tmp = node.left;   // 1. move left
              while (tmp.right != null) // 2. and then right as far as
                  tmp = tmp.right;      //    possible;
              tmp.right =        // 3. establish the link between
                  node.right;    //    the rightmost node of the left
                                 //    subtree and the right subtree;
              node = node.left;  // 4.
```

Figure 6-29 Implementation of algorithm for deleting by merging

Deletion by Merging (continued)

```
        }
        if (p == root)
             root = node;
        else if (prev.left == p)
             prev.left = node;
        else prev.right = node; // 5.
    }
    else if (root != null)
        System.out.println("key " + el + " is not in the tree");
    else System.out.println("the tree is empty");
}
```

Figure 6-29 Implementation of algorithm for deleting by merging (continued)

Deletion by Merging (continued)

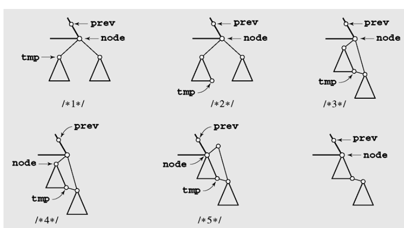


Figure 6-30 Details of deleting by merging

Deletion by Merging (continued)

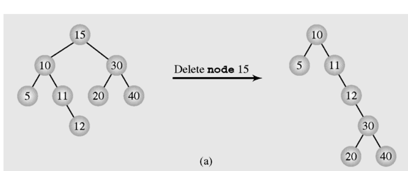


Figure 6-31 The height of a tree can be (a) extended or (b) reduced after deleting by merging

Deletion by Merging (continued)

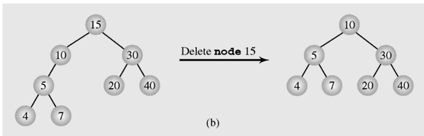


Figure 6-31 The height of a tree can be (a) extended or (b) reduced after deleting by merging (continued)

Data Structures and Algorithms in Java 52

Deletion by Copying

- If the node has two children, the problem can be reduced to:
  - The node is a leaf
  - The node has only one nonempty child
- Solution: replace the key being deleted with its immediate predecessor (or successor)
- A key's predecessor is the key in the rightmost node in the left subtree

Data Structures and Algorithms in Java 53

Deletion by Copying (continued)

```
public void deleteByCopying(int el) {
   IntBSTNode node, p = root, prev = null;
   while (p != null && p.key != el)       {// find the node p
        prev = p;                          // with element el;
        if (p.key < el)
            p = p.right;
       else p = p.left;
  }
  node = p;
  if (p != null && p == el) {
       if (node.right == null)             // node has no right child;
            node = node.left;
       else if (node.left == null)         // no left child for node;
            node = node.right;
```

Figure 6-32 Implementation of an algorithm for deleting by copying

Data Structures and Algorithms in Java 54

Deletion by Copying

```
else {
     IntBSTNode tmp = node.left;     // node has both children;
     IntBSTNode previous = node;     // 1.
     while (tmp.right != null) {     // 2. find the rightmost
         previous = tmp;             //    position in the
         tmp = tmp.right;            //    left subtree of node;
     }
     node.key = tmp.key;             // 3. overwrite the reference
                                     //    of the key being deleted;
     if (previous == node)           // if node's left child's
          previous.left  = tmp.left; // right subtree is null;
     else previous.right = tmp.left; // 4.
```

Figure 6-32 Implementation of an algorithm for deleting by copying (continued)

Deletion by Copying

```
          }
          if (p == root)
               root = node;
          else if (prev.left == p)
               prev.left = node;
          else prev.right = node;
     }
     else if (root != null)
          System.out.println("key " + el + " is not in the tree");
     else System.out.println("the tree is empty");
}
```

Figure 6-32 Implementation of an algorithm for deleting by copying (continued)

Deletion by Copying (continued)

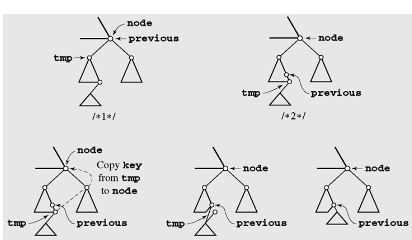


Figure 6-33 Deleting by copying

Day 3

Balancing a Tree

- A binary tree is **height-balanced** or **balanced** if the difference in height of both subtrees of **any node** in the tree is **either zero or one.**
- A tree is considered **perfectly balanced** if it is balanced and all leaves are to be found on one level or two levels.
- Advantage of balanced tree: Its height is small → Search operation is faster and predictable.
- In the worst case, the complexity of the search operation in a tree is O(height).

Balancing a Tree (continued)

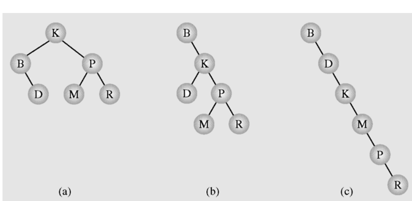
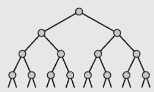


Figure 6-34 Different binary search trees with the same information

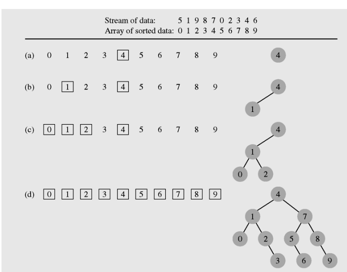
Balancing a Tree (continued)



Height	Nodes at One Level	Nodes at All Levels
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
:		
11	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
:		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
:		
h	2^{h-1}	$n = 2^h - 1$
:		

Figure 6-35 Maximum number of nodes in binary trees of different heights

Balancing a Tree (continued)



How to create a balanced tree from a normal BST?
(1) Use the inorder traversal to get its information to an array.
(2) Create a a balanced BST from this array.

➔Additional storage is needed
➔Another algorithm: DSW

Figure 6-36 Creating a binary search tree from an ordered array