COURSE TECHNOLOGY
CENGAGE Learning™

Data Structures Using C++ 2E

Chapter 11
Binary Trees and B-Trees

Objectives

- Learn about binary trees
- Explore various binary tree traversal algorithms
- Learn how to organize data in a binary search tree
- Discover how to insert and delete items in a binary search tree
- Explore nonrecursive binary tree traversal algorithms
- Learn about AVL (height-balanced) trees
- Learn about B-trees

Binary Trees

- Definition: a binary tree, T, is either empty or such that
  - T has a special node called the root node
  - T has two sets of nodes, L_T and R_T, called the left subtree and right subtree of T, respectively
  - L_T and R_T are binary trees

- Can be shown pictorially
  - Parent, left child, right child
  - Circle labeled by the node
  - Root node at the top
    - left child
    - Right child
  - Directed edge (branch): arrow
  - Leaf node: no left/right children

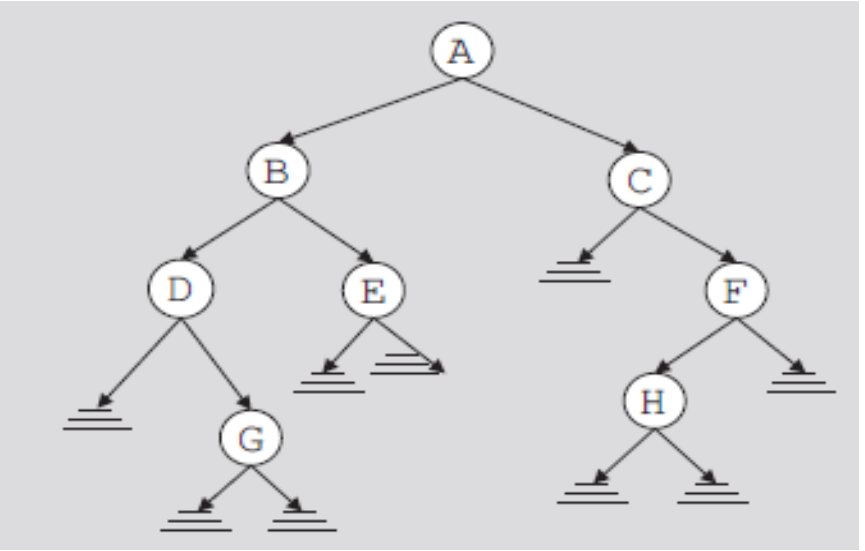


FIGURE 11-1 Binary tree

Binary Trees (cont'd.)

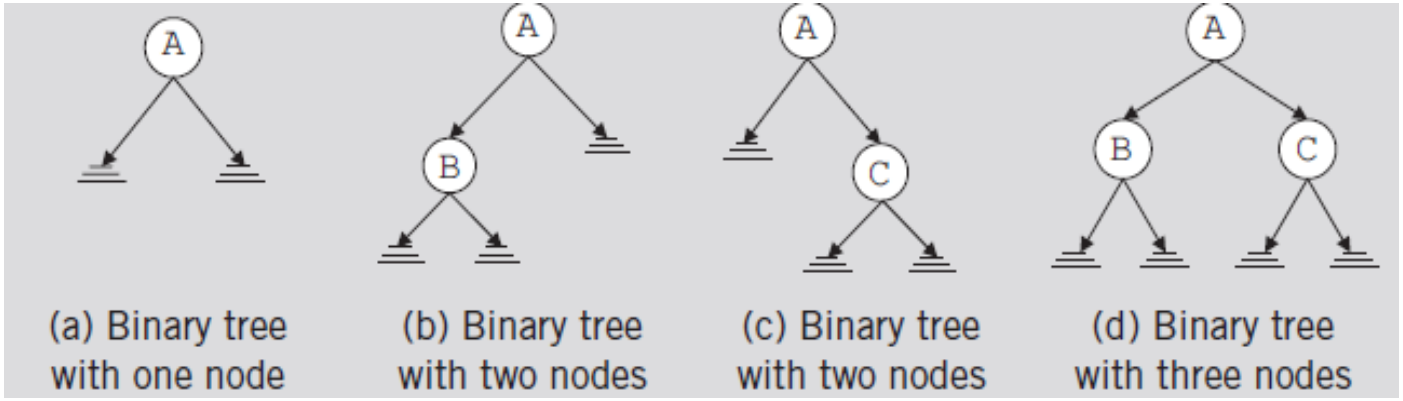


FIGURE 11-2 Binary tree with one, two, or three nodes

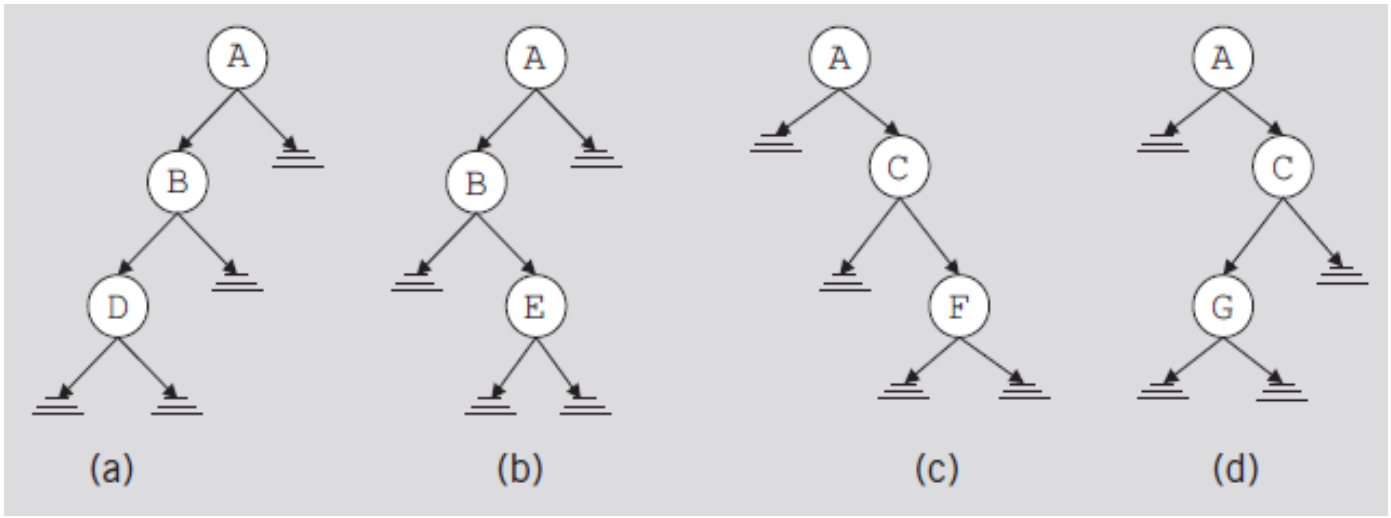


FIGURE 11-3 Various binary trees with three nodes

Binary Trees (cont'd.)

- Every node in a binary tree
  - Has at most two children
  - Vs. singly linked list: only one link for each node
- `struct` defining node of a binary tree
  - The data stored in `info`
  - A pointer to the left child stored in `llink`
  - A pointer to the right child stored in `rlink`

```
template <class elemType>
struct binaryTreeNode
{
    elemType info;
    binaryTreeNode<elemType> *llink;
    binaryTreeNode<elemType> *rlink;
};
```

Binary Trees (cont'd.)

- Pointer to root node stored outside the binary tree
  - In pointer variable `root` of type `binaryTreeNode`

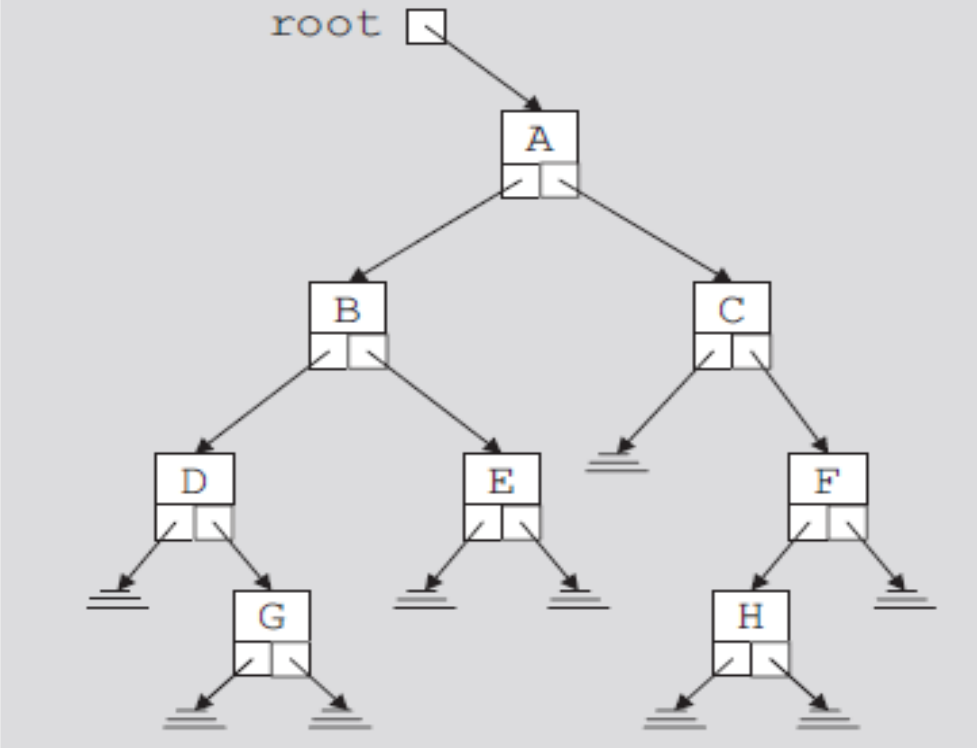


FIGURE 11-4 Binary tree

Binary Trees (cont'd.)

- Level of a node
  - # of branches on the path from the root to the node
- Height of a binary tree
  - # of nodes on the longest path from the root to a leaf
  - How to calculate? (see p. 604)

Root: level = 0
Child of root: level = 1

Empty tree: height = 0
Single node tree: height = 1

```
template <class elemType>
int height(binaryTreeNode<elemType> *p) const
{
    if (p == NULL)
        return 0;
    else
        return 1 + max(height(p->llink), height(p->rlink));
}
```

Copy Tree

- Shallow copy of the data
  - Obtained when value of the pointer of the root node used to make a copy of a binary tree
- Identical (deep) copy of a binary tree
  - Need to create as many nodes as there are in the binary tree to be copied
  - Nodes must appear in the same order as in the original binary tree
- Function `copyTree`
  - Makes a copy of a given binary tree
  - See code on pages 604-605

Copy Tree (cont'd.)

Call by reference

```
template <class elemType>
void copyTree(binaryTreeNode<elemType>* &copiedTreeRoot,
              binaryTreeNode<elemType>* otherTreeRoot)
{
    if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
    else
    {
        copiedTreeRoot = new binaryTreeNode<elemType>;
        copiedTreeRoot->info = otherTreeRoot->info;
        copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
        copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
    }
}//end copyTree
```

What if both formal parameters of copyTree are call-by-value?

Binary Tree Traversal

- How to traverse?
  - Start with the root, and then
  - Visit the node first *or* visit the subtrees first
- Three different traversals
  - *Inorder*: left subtree ➔ node ➔ right subtree
  - *Preorder*: node ➔ left subtree ➔ right subtree
  - *Postorder*: left subtree ➔ right subtree ➔ node
- Each traversal algorithm: recursive
- Listing of nodes
  - Inorder, preorder, or postorder sequence

Binary Tree Traversal (cont'd.)

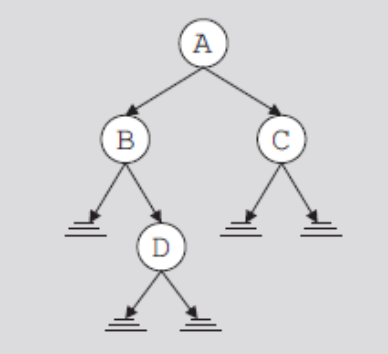
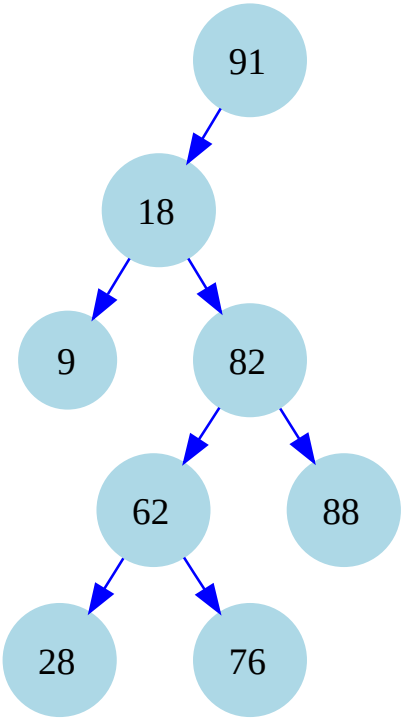


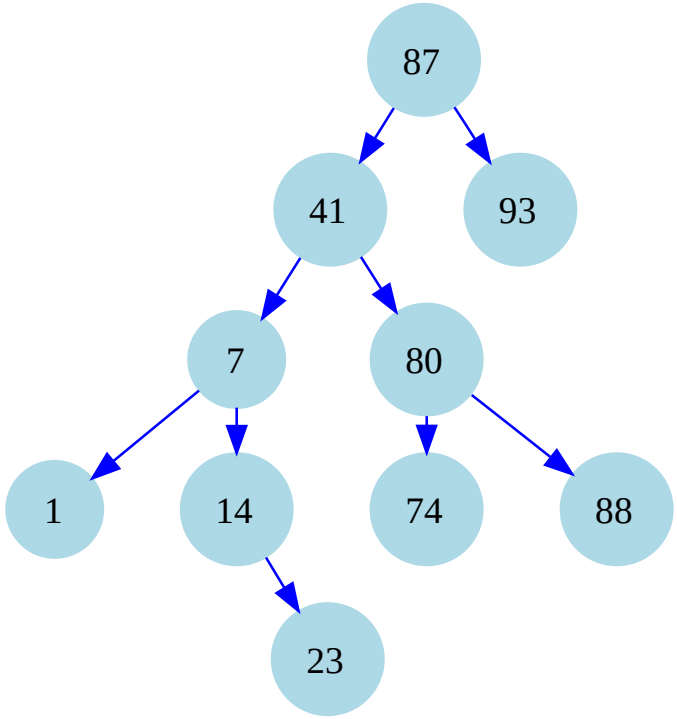
FIGURE 11-5 Binary tree for an inorder traversal

```
template <class elemType>
void inorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        inorder(p->llink);
        cout << p->info << " ";
        inorder(p->rlink);
    }
}
template <class elemType>
void preorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        cout << p->info << " ";
        preorder(p->llink);
        preorder(p->rlink);
    }
}

template <class elemType>
void postorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        postorder(p->llink);
        postorder(p->rlink);
        cout << p->info << " ";
    }
}
```

inorder: B D A C
preorder: A B D C
postorder: D B C A

Binary Tree Traversal Exercises





Implementing Binary Trees

- Operations typically performed on a binary tree
  - Determine if binary tree is empty
  - *Search binary tree for a particular item*
  - *Insert an item in the binary tree*
  - *Delete an item from the binary tree*

  binary search tree
  - Find the height of the binary tree
  - Find the number of nodes in the binary tree
  - Find the number of leaves in the binary tree
  - Traverse the binary tree
  - Copy the binary tree

Implementing Binary Trees (cont'd.)

- `class binaryTreeType`
  - Specifies basic operations to implement a binary tree
  - See code on page 609
    - Contains statements to overload the assignment operator, copy constructor, destructor
    - Contains several member functions that are private members of the class
    - Contains a member variable `root`

```
binaryTreeNode<elemType> *root;
```

- Binary tree empty if `root` is `NULL`
  - See `isEmpty` function on page 611

Implementing Binary Trees (cont'd.)

- Default constructor
  - Initializes binary tree to an empty state: `root = NULL;`
  - See code on page 612
- Other functions for binary trees
  - See code on pages 612-613
- Functions: `copyTree`, `destroy`, `destroyTree`
  - See code on page 614
- Copy constructor, destructor, and overloaded assignment operator
  - See code on page 615

```
template <class elemType>
struct binaryTreeNode
{
    elemType info;
    binaryTreeNode<elemType> *llink;
    binaryTreeNode<elemType> *rlink;
};
```

```
template <class elemType>
class binaryTreeType
{
public:
    const binaryTreeType<elemType>& operator=
                    (const binaryTreeType<elemType>&);
    bool isEmpty() const;
    void inorderTraversal() const;
    void preorderTraversal() const;
    void postorderTraversal() const;

    int treeHeight() const;
    int treeNodeCount() const;
    int treeLeavesCount() const;
    void destroyTree();

    binaryTreeType(const binaryTreeType<elemType>& otherTree);
    binaryTreeType();
    ~binaryTreeType();

protected:
    binaryTreeNode<elemType>  *root;

private:
    void copyTree(binaryTreeNode<elemType>* &copiedTreeRoot,
                  binaryTreeNode<elemType>* otherTreeRoot);
    void destroy(binaryTreeNode<elemType>* &p);
    void inorder(binaryTreeNode<elemType> *p) const;
    void preorder(binaryTreeNode<elemType> *p) const;
    void postorder(binaryTreeNode<elemType> *p) const;
    int height(binaryTreeNode<elemType> *p) const;
    int max(int x, int y) const;
    int nodeCount(binaryTreeNode<elemType> *p) const;
    int leavesCount(binaryTreeNode<elemType> *p) const;
};
```

Why?

```
template <class elemType>
binaryTreeType<elemType>::binaryTreeType()
{
    root = NULL;
}

template <class elemType>
bool binaryTreeType<elemType>::isEmpty() const
{
    return (root == NULL);
}

template <class elemType>
void binaryTreeType<elemType>::inorderTraversal() const
{
    inorder(root);
}

template <class elemType>
void binaryTreeType<elemType>::preorderTraversal() const
{
    preorder(root);
}

template <class elemType>
void binaryTreeType<elemType>::postorderTraversal() const
{
    postorder(root);
}

template <class elemType>
int binaryTreeType<elemType>::treeHeight() const
{
    return height(root);
}

template <class elemType>
int binaryTreeType<elemType>::treeNodeCount() const
{
    return nodeCount(root);
}

template <class elemType>
int binaryTreeType<elemType>::treeLeavesCount() const
{
    return leavesCount(root);
}
```

```
template <class elemType>
void  binaryTreeType<elemType>::copyTree
                      (binaryTreeNode<elemType>* &copiedTreeRoot,
                       binaryTreeNode<elemType>* otherTreeRoot)
{
    if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
    else
    {
        copiedTreeRoot = new binaryTreeNode<elemType>;
        copiedTreeRoot->info = otherTreeRoot->info;
        copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
        copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
    }
} //end copyTree

template <class elemType>
void binaryTreeType<elemType>::inorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        inorder(p->llink);
        cout << p->info << " ";
        inorder(p->rlink);
    }
}

template <class elemType>
void binaryTreeType<elemType>::preorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        cout<<p->info<<" ";
        preorder(p->llink);
        preorder(p->rlink);
    }
}

template <class elemType>
void binaryTreeType<elemType>::postorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        postorder(p->llink);
        postorder(p->rlink);
        cout << p->info << " ";
    }
}
```

```
template <class elemType>
const binaryTreeType<elemType>& binaryTreeType<elemType>::
          operator=(const binaryTreeType<elemType>& otherTree)
{
    if (this != &otherTree) //avoid self-copy
    {
        if (root != NULL)  //if the binary tree is not empty,
                           //destroy the binary tree
            destroy(root);

        if (otherTree.root == NULL) //otherTree is empty
            root = NULL;
        else
            copyTree(root, otherTree.root);
    }//end else

    return *this;
}

template <class elemType>
void  binaryTreeType<elemType>::destroy(binaryTreeNode<elemType>* &p)
{
    if (p != NULL)
    {
        destroy(p->llink);
        destroy(p->rlink);
        delete p;
        p = NULL;
    }
}

template <class elemType>
void  binaryTreeType<elemType>::destroyTree()
{
    destroy(root);
}

    //copy constructor
template <class elemType>
binaryTreeType<elemType>::binaryTreeType
             (const binaryTreeType<elemType>& otherTree)
{
    if (otherTree.root == NULL) //otherTree is empty
        root = NULL;
    else
        copyTree(root, otherTree.root);
}

template <class elemType>
binaryTreeType<elemType>::~binaryTreeType()
{
    destroy(root);
}
```

```
template <class elemType>
int binaryTreeType<elemType>::height(binaryTreeNode<elemType> *p) const
{
    if (p == NULL)
        return 0;
    else
        return 1 + max(height(p->llink), height(p->rlink));
}

template <class elemType>
int binaryTreeType<elemType>::max(int x, int y) const
{
    if (x >= y)
        return x;
    else
        return y;
}

template <class elemType>
int binaryTreeType<elemType>::nodeCount(binaryTreeNode<elemType> *p) const
{
    cout << "Write the definition of the function nodeCount"
         << endl;

    return 0;
}

template <class elemType>
int binaryTreeType<elemType>::leavesCount(binaryTreeNode<elemType> *p) const
{
    cout << "Write the definition of the function leavesCount"
         << endl;

    return 0;
}
```

Binary Search Trees

- Search in a binary tree: O(*n*)
- *Binary search tree*: Data in each node
  - Larger than the data in its right child
  - Smaller than the data in its left child

Avg case search: O(log *n*)

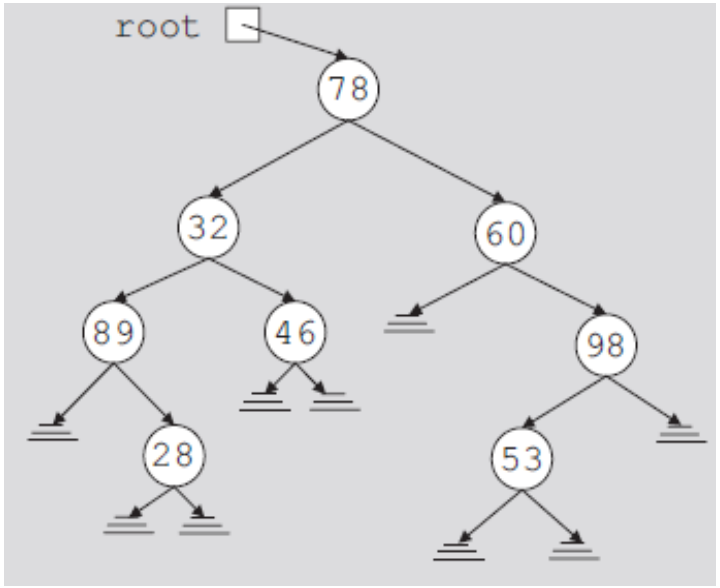


FIGURE 11-6 Arbitrary binary tree

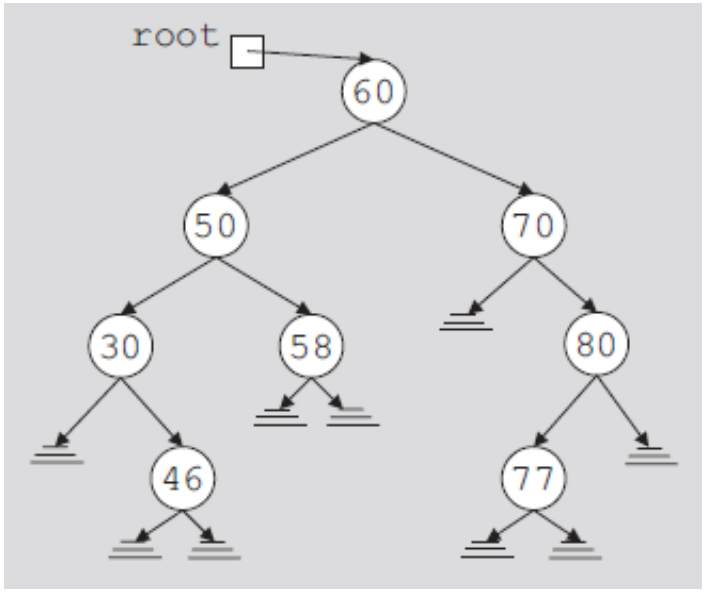


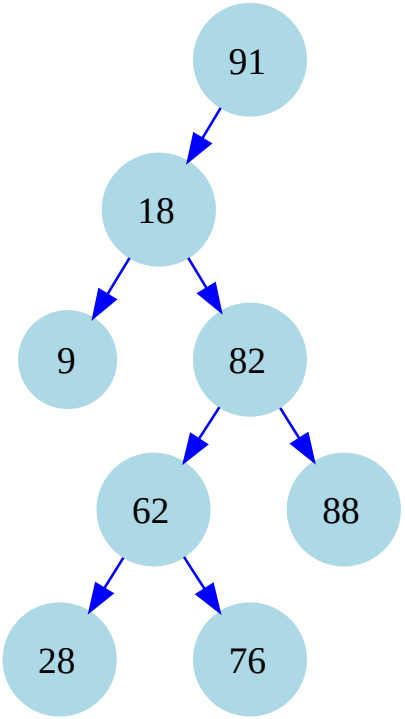
FIGURE 11-7 Binary search tree

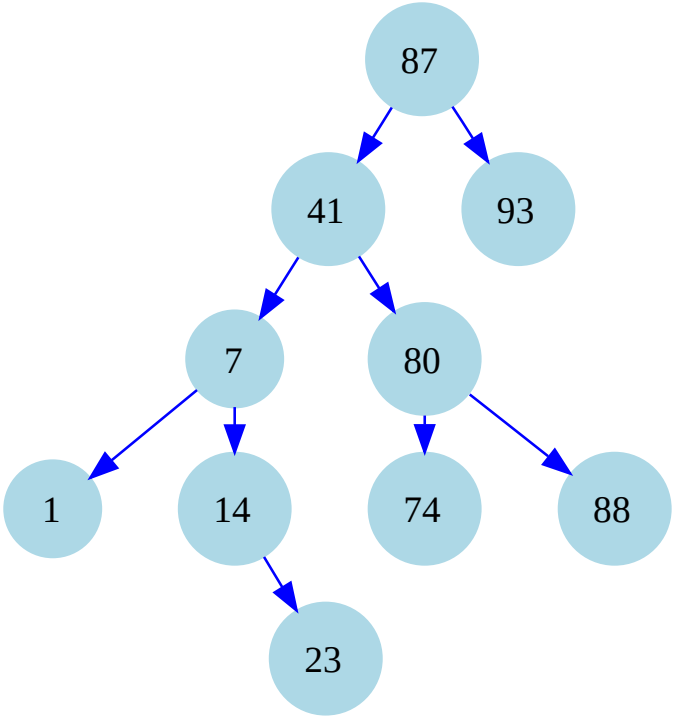
Binary Search Trees (cont'd.)

- Definition: A binary search tree, *T*, is either empty or the following is true:
  - *T* has a special node called the root node
  - *T* has two sets of nodes, L_T and R_T , called the left subtree and right subtree of T, respectively
  - The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree
  - L_T and R_T are binary search trees

Binary Search Tree: Exercises

- Binary search tree? Why or why not?





Binary Search Trees (cont'd.)

- Operations performed on a binary search tree
  - Search the binary search tree for a particular item
  - Insert an item in the binary search tree
  - Delete an item from the binary search tree
  - Find the height of the binary search tree
  - Find the number of nodes in the binary search tree
  - Find the number of leaves in the binary search tree
  - Traverse the binary search tree
  - Copy the binary search tree

Binary Search Trees (cont'd.)

- Every binary search tree is a binary tree
- Height of a binary search tree
  - Determined the same way as the height of a binary tree
- Operations to find number of nodes, number of leaves, to do inorder, preorder, postorder traversals of a binary search tree
  - Same as those for a binary tree: can inherit functions
  - Inorder traversal ➔ ascending order

Binary Search Trees (cont'd.)

- `class bSearchTreeType`
  - Illustrates basic operations to implement a binary search tree
  - See code on page 618

```
template <class elemType>
class bSearchTreeType: public binaryTreeType<elemType>
{
public:
    bool search(const elemType& searchItem) const;
    void insert(const elemType& insertItem);
    void deleteNode(const elemType& deleteItem);

private:
    void deleteFromTree(binaryTreeNode<elemType>* &p);
};
```

Binary Search Trees: search

- Function `search`

```
if root is NULL
    Cannot search an empty tree, returns false.
else
{
    current = root;
    while (current is not NULL and not found)
        if (current->info is the same as the search item)
            set found to true;
        else if(current->info is greater than the search item)
            follow the llink of current
        else
            follow the rlink of current
}
```

```
template <class elemType>
bool bSearchTreeType<elemType>::
                        search(const elemType& searchItem) const
{
    binaryTreeNode<elemType> *current;
    bool found = false;

    if (root == NULL)
       cerr << "Cannot search the empty tree." << endl;
    else
    {
        current = root;

        while (current != NULL && !found)
        {
            if (current->info == searchItem)
                found = true;
            else if (current->info > searchItem)
                current = current->llink;
            else
                current = current->rlink;
        }//end while
    }//end else

    return found;
}//end search
```

Binary Search Trees: Insert

- Function `insert`
  - `trailCurrent`: point to the parent of `current`

a. Create a new node and copy **insertItem** into the new node. Also set **llink** and **rlink** of the new node to **NULL**.

b. **if** the **root** is **NULL**, the tree is empty so make **root** point to the new node.

```
else
{
    current = root;
    while (current is not NULL)    //search the binary tree
    {
        trailCurrent = current;
        if (current->info is the same as the insertItem)
            Error: Cannot insert duplicate
            exit
        else if(current->info > insertItem)
            Follow llink of current
        else
            Follow rlink of current
    }

    //insert the new node in the binary tree

    if (trailCurrent->info > insertItem)
        make the new node the left child of trailCurrent
    else
        make the new node the right child of trailCurrent
}
```

```
template <class elemType>
void bSearchTreeType<elemType>::insert(const elemType& insertItem)
{
    binaryTreeNode<elemType> *current;  //pointer to traverse the tree
    binaryTreeNode<elemType> *trailCurrent; //pointer behind current
    binaryTreeNode<elemType> *newNode;  //pointer to create the node

    newNode = new binaryTreeNode<elemType>;
    assert(newNode != NULL);
    newNode->info = insertItem;
    newNode->llink = NULL;
    newNode->rlink = NULL;

    if (root == NULL)
        root = newNode;
    else
    {
        current = root;

        while (current != NULL)
        {
            trailCurrent = current;

            if (current->info == insertItem)
            {
                cerr << "The insert item is already in the list-";
                cerr << "duplicates are not allowed."
                     << insertItem << endl;
                return;
            }
            else if (current->info > insertItem)
                current = current->llink;
            else
                current = current->rlink;
        }//end while

        if (trailCurrent->info > insertItem)
            trailCurrent->llink = newNode;
        else
            trailCurrent->rlink = newNode;
    }
}//end insert
```

Binary Search Trees: Delete

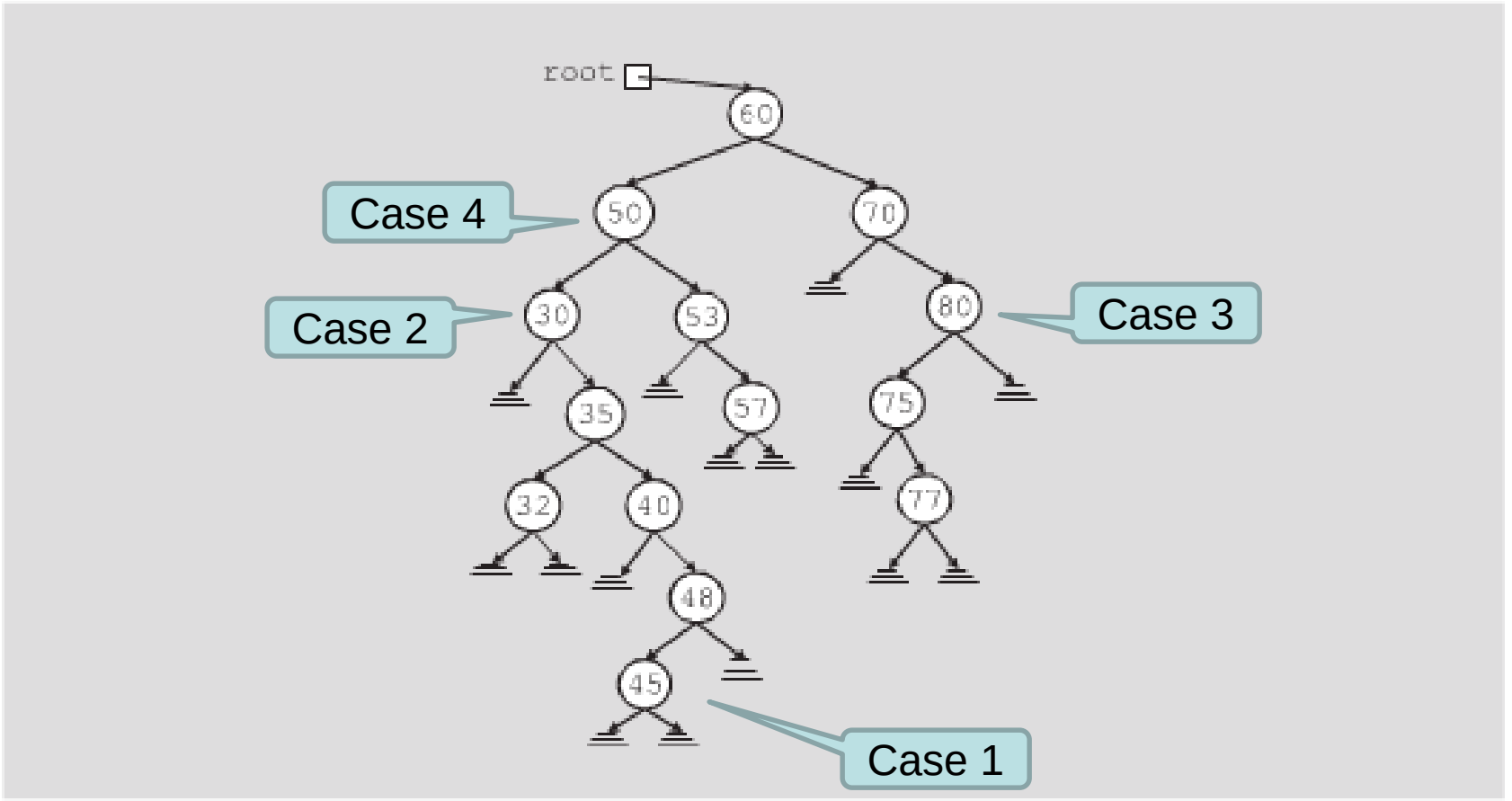


FIGURE 11-8 Binary search tree before deleting a node

Binary Search Trees (cont'd.)

- Function `delete`

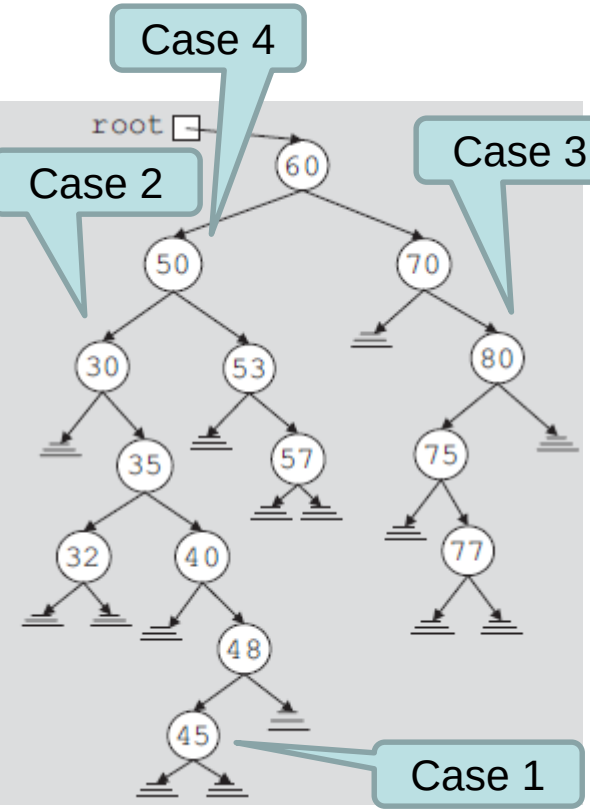
After deleting the desired item (if it exists in the binary search tree), the resulting tree must be a binary search tree. The delete operation has four cases, as follows:

Case 1: The node to be deleted has no left and right subtrees; that is, the node to be deleted is a leaf. For example, the node with **`info 45`** is a leaf.

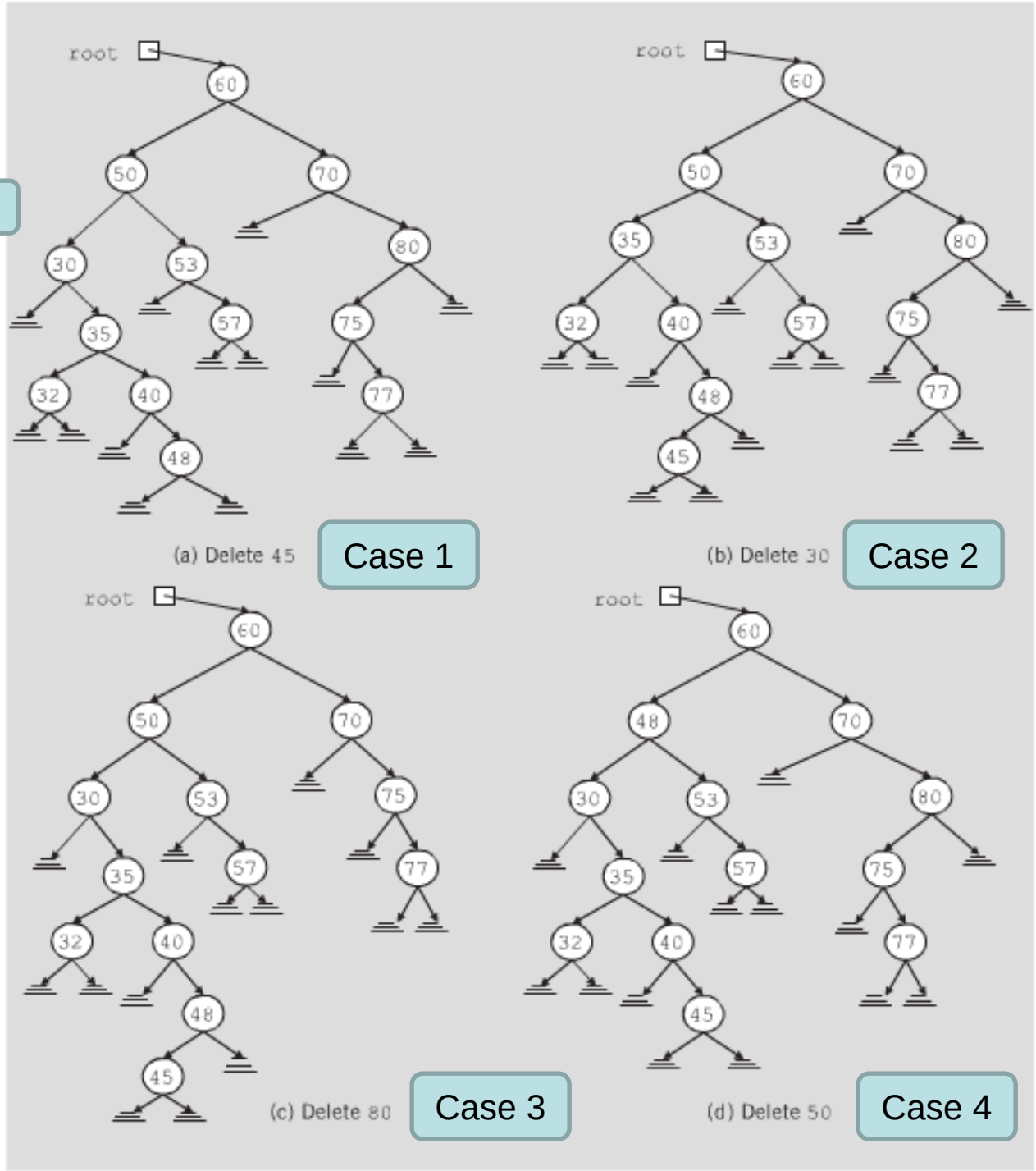
Case 2: The node to be deleted has no left subtree; that is, the left subtree is empty, but it has a nonempty right subtree. For example, the left subtree of node with **`info 40`** is empty and its right subtree is nonempty.

Case 3: The node to be deleted has no right subtree; that is, the right subtree is empty, but it has a nonempty left subtree. For example, the left subtree of node with **`info 80`** is empty and its right subtree is nonempty.

Case 4: The node to be deleted has nonempty left and right subtrees. For example, the left and the right subtrees of node with **`info 50`** are nonempty.



Case 4:
- reduce to case 3: find rightmost node from the left subtree, OR
- reduce to case 2: find leftmost node from the right subtree



```
template <class elemType>
void bSearchTreeType<elemType>::deleteFromTree
                                (binaryTreeNode<elemType>* &p)
{
    binaryTreeNode<elemType> *current; //pointer to traverse the tree
    binaryTreeNode<elemType> *trailCurrent;   //pointer behind current
    binaryTreeNode<elemType> *temp;        //pointer to delete the node

    if (p == NULL)
        cerr << "Error: The node to be deleted is NULL." << endl;
    else if(p->llink == NULL && p->rlink == NULL)
    {
        temp = p;
        p = NULL;
        delete temp;
    }
    else if(p->llink == NULL)
    {
        temp = p;
        p = temp->rlink;
        delete temp;
    }
    else if(p->rlink == NULL)
    {
        temp = p;
        p = temp->llink;
        delete temp;
    }
    else
    {
        current = p->llink;
        trailCurrent = NULL;

        while (current->rlink != NULL)
        {
            trailCurrent = current;
            current = current->rlink;
        }//end while

        p->info = current->info;

        if (trailCurrent == NULL) //current did not move;
                                  //current == p->llink; adjust p
            p->llink = current->llink;
        else
            trailCurrent->rlink = current->llink;

        delete current;
    }//end else
}//end deleteFromTree
```

Why call by reference?

Case 1

Case 2

Case 3

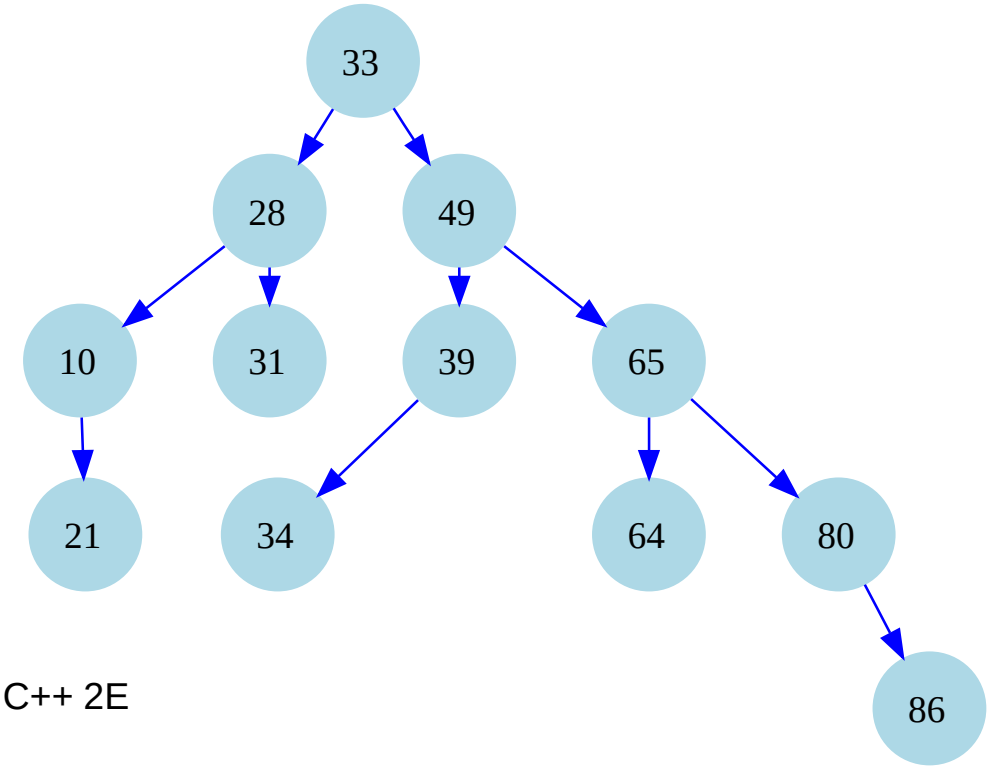
Case 4

```
template <class elemType>
void bSearchTreeType<elemType>::deleteNode(const elemType& deleteItem)
{
    binaryTreeNode<elemType> *current;  //pointer to traverse the tree
    binaryTreeNode<elemType> *trailCurrent; //pointer behind current
    bool found = false;

    if (root == NULL)
        cout << "Cannot delete from the empty tree." << endl;
    else
    {
        current = root;
        trailCurrent = root;

        while (current != NULL && !found)
        {
            if (current->info == deleteItem)
                found = true;
            else
            {
                trailCurrent = current;

                if (current->info > deleteItem)
                    current = current->llink;
                else
                    current = current->rlink;
            }
        }//end while

        if (current == NULL)
            cout << "The delete item is not in the tree." << endl;
        else if (found)
        {
            if (current == root)
                deleteFromTree(root);
            else if (trailCurrent->info > deleteItem)
                deleteFromTree(trailCurrent->llink);
            else
                deleteFromTree(trailCurrent->rlink);
        }//end if
    }
}//end deleteNode
```

Why not deleteFromTree(current)?

Binary Search Trees: Exercises

- Example: search for 34
- Example: insert 50
- Example: delete 86, delete 10, delete 39, delete 28



Binary Search Tree: Analysis

- *T*: binary search tree with *n* nodes
- Worst case: *T* is linear
  - Successful case: makes (*n* + 1) / 2 key comparisons (average)
  - Unsuccessful case: makes *n* comparisons

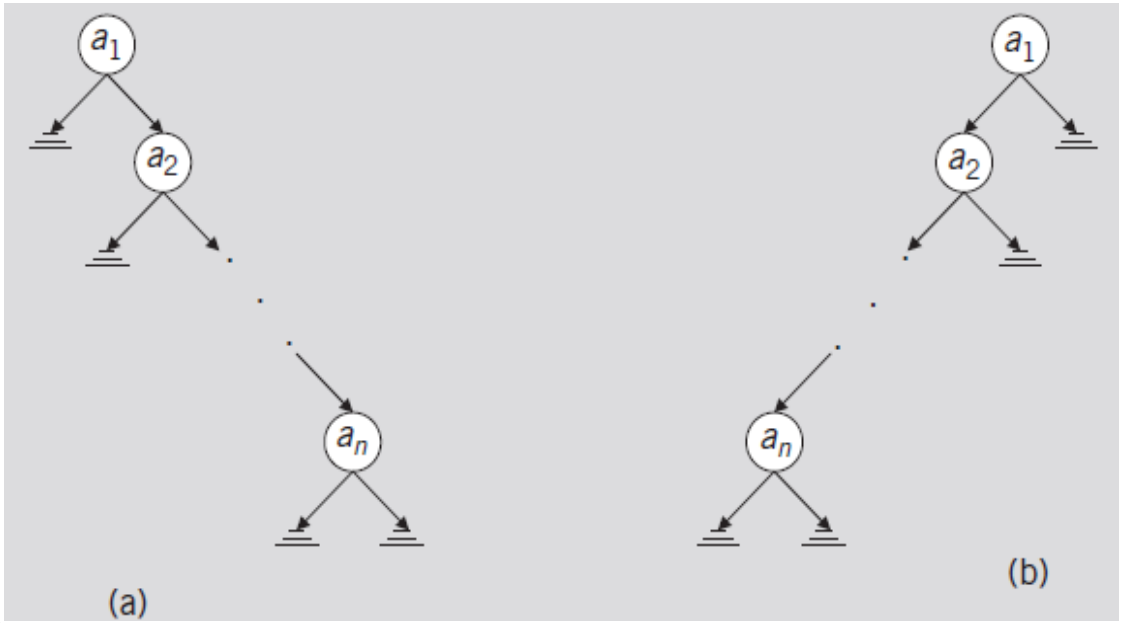


FIGURE 11-10 Linear binary trees

Binary Search Tree: Analysis (cont'd.)

- Average-case behavior
  - Successful case
    - Search would end at a node
    - *n* items ➔ *n*! possible orderings of the keys
  - # of key comparisons: S(n), U(n)
  - # of comparisons required to determine whether *x* is in *T*
    - One more than # of comparisons required to insert *x* in *T*
  - # of comparisons required to insert *x* in *T*
    - Same as # of comparisons made in the unsuccessful search, reflecting that *x* is not in *T*

Binary Search Tree: Analysis (cont'd.)

$$S(n) = 1 + \frac{U(0) + U(1) + \ldots + U(n-1)}{n} \qquad \text{(Equation 11-1)}$$

It is also known that

$$S(n) = \left(1 + \frac{1}{n}\right) U(n) - 3 \qquad \text{(Equation 11-2)}$$

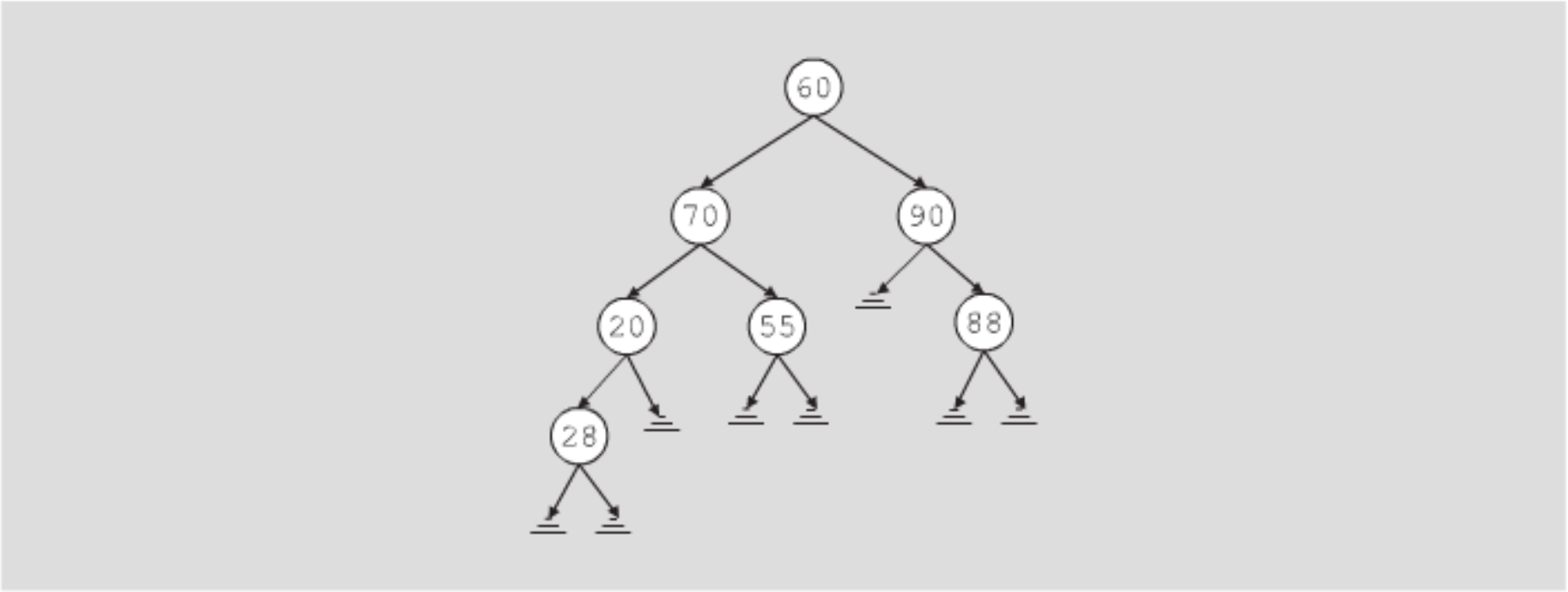
Solving Equations (11-1) and (11-2), it can be shown that $U(n) \approx 2.77\log_2 n$ and $S(n) \approx 1.39\log_2 n$.

Binary Search Tree: Analysis (cont'd.)

- Theorem: let *T* be a binary search tree with *n* nodes, where *n*> 0
  - The average number of nodes visited in a search of *T* is approximately $1.39\log_2 n = O(\log_2 n)$
  - The number of key comparisons is approximately $2.77 \log_2 n = O(\log_2 n)$

Non-recursive Binary Tree Traversal

- Recursion is less efficient than iteration
- Discuss inorder, preorder, and postorder traversal



Nonrecursive Inorder Traversal

1. `current = root; //start traversing the binary tree at the root node`
2. `while (current is not NULL or stack is nonempty)`

```
if (current is not NULL)
{
   push current into the stack;
   current = current->llink;
}
else
{
     pop stack into current;
     visit current;          //visit the node
     current = current->rlink;     //move to the right child
}
```

Nonrecursive Inorder Traversal (cont'd.)

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursiveInTraversal() const
{
    stackType<binaryTreeNode<elemType>* > stack;
    binaryTreeNode<elemType> *current;
    current = root;

    while ((current != NULL) || (!stack.isEmptyStack()))
        if (current != NULL)
        {
            stack.push(current);
            current = current->llink;
        }
        else
        {
            current = stack.top();
            stack.pop();
            cout << current->info << " ";
            current = current->rlink;
        }

    cout << endl;
}
```

Nonrecursive Preorder Traversal

```
1.  current = root;          //start the traversal at the root node
2.  while (current is not NULL or stack is nonempty)
    if (current is not NULL)
    {
         visit current node;
         push current into stack;
         current = current->llink;
    }
    else
    {
         pop stack into current;
         current = current->rlink; //prepare to visit the
                                   //right subtree
    }
```

Nonrecursive Preorder Traversal (cont'd.)

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursivePreTraversal() const
{
    stackType<binaryTreeNode<elemType>*> stack;
    binaryTreeNode<elemType> *current;

    current = root;

    while ((current != NULL) || (!stack.isEmptyStack()))
        if (current != NULL)
        {
            cout << current->info << " ";
            stack.push(current);
            current = current->llink;
        }
        else
        {
            current = stack.top();
            stack.pop();
            current = current->rlink;
        }

    cout << endl;
}
```

Nonrecursive Postorder Traversal

```
1.  current = root; //start the traversal at the root node
2.  v = 0;
3.  if (current is NULL)
      the binary tree is empty
4.  if (current is not NULL)
    a.  push current into stack;
    b.  push 1 into stack;
    c.  current = current->llink;
    d.  while (stack is not empty)
            if (current is not NULL and v is 0)
            {
                push current and 1 into stack;
                current = current->llink;
            }
            else
            {
                pop stack into current and v;
                if (v == 1)
                {
                    push current and 2 into stack;
                    current = current->rlink;
                    v = 0;
                }
                else
                    visit current;
            }
```

v	
0	visit left tree
1	left tree is done; visit right tree
2	right tree is done; visit the node

Binary Tree Traversal and Functions as Parameters

- Passing a function as a parameter to the traversal algorithms
  - Enhances program's flexibility
- C++ function name without any parentheses
  - Considered a pointer to the function
- A function as a formal parameter to another function

```
void fParamFunc1(void (*visit) (int));
```

// `visit`: any `void` function w/ one `int` formal parameter

```
void fParamFunc2(int (*visit)(elemType&));
```

// `visit`: any `int` function w/ one `elemType&` formal parameter

- See Example 11-3

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal
                               (void (*visit) (elemType& item))
{
    inorder(root, *visit);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(binaryTreeNode<elemType>* p,
                               void (*visit) (elemType& item))
{
    if (p != NULL)
    {
        inorder(p->llink, *visit);
        (*visit)(p->info);
        inorder(p->rlink, *visit);
    }
}
```

AVL (Height-Balanced) Trees

- AVL tree (height-balanced tree)
  - Special type of binary search tree
  - Resulting binary search is nearly balanced
- Perfectly balanced binary tree
  - Heights of left and right subtrees of the root: equal
  - Left and right subtrees of the root are perfectly balanced binary trees

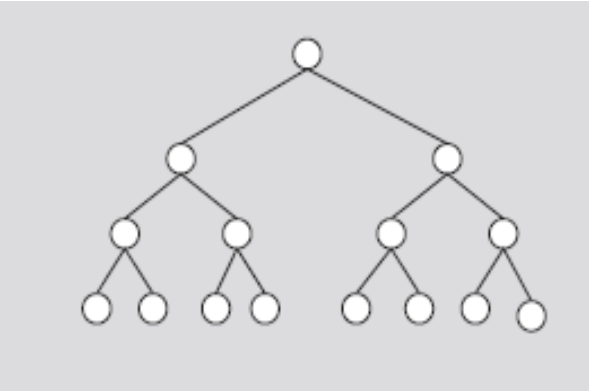


FIGURE 11-12 Perfectly balanced binary tree

Let T be a perfectly balanced binary tree with height h, # of nodes in T is $2^h - 1$

AVL (Height-Balanced) Trees (cont'd.)

- An AVL tree (or height-balanced tree) is a binary search tree such that
  - The heights of the left and right subtrees of the root differ by at most one
  - The left and right subtrees of the root are AVL trees

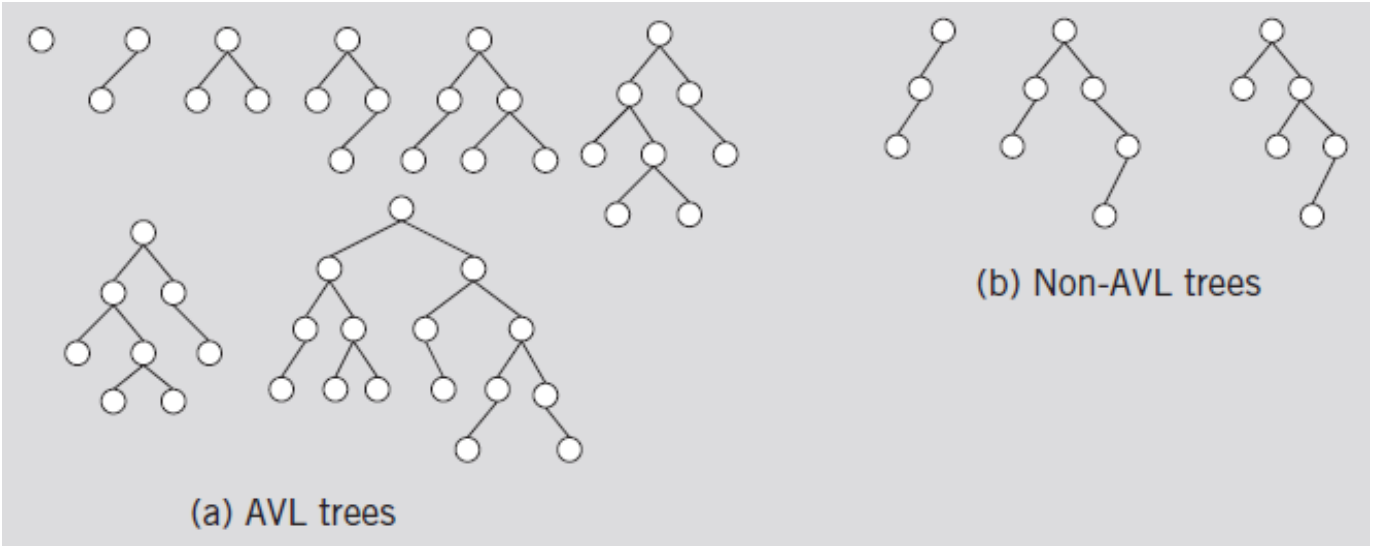


FIGURE 11-13 AVL and non-AVL trees

AVL (Height-Balanced) Trees (cont'd.)

Let x be a node in a binary tree. Let x_l denote the height of the left subtree of x, and x_h denote the height of the right subtree of x.

Proposition: Let T be an AVL tree and x be a node in T. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.

Let x be a node in the AVL tree T.

1. If $x_l > x_h$, we say that x is **left high**. In this case, $x_l = x_h + 1$.
2. If $x_l = x_h$, we say that x is **equal high**.
3. If $x_h > x_l$, we say that x is **right high**. In this case, $x_h = x_l + 1$.

Definition: The **balance factor** of x, written $bf(x)$, is defined by $bf(x) = x_h - x_l$.

Let x be a node in the AVL tree T. Then,

1. If x is left high, $bf(x) = -1$.
2. If x is equal high, $bf(x) = 0$.
3. If x is right high, $bf(x) = 1$.

Definition: Let x be a node in a binary tree. We say that the node x **violates the balance criteria** if $|x_h - x_l| > 1$, that is, the heights of the left and right subtrees of x differ by more than 1.

AVL (Height-Balanced) Trees (cont'd.)

- Definition of a node in the AVL tree

```
template<class elemType>
struct AVLNode
{
    elemType info;
    int bfactor;  //balance factor
    AVLNode<elemType> *llink;
    AVLNode<elemType> *rlink;
};
```

- AVL binary search tree search algorithm
  - Same as for a binary search tree
  - Other operations on AVL trees
    - Implemented the same way as binary search trees
  - Item insertion and deletion operations on AVL trees
    - Somewhat different from binary search trees operations

AVL Trees: Insertion

- First search the tree and find the place where the new item is to be inserted
  - Can search using algorithm similar to search algorithm designed for binary search trees
  - If the item is already in tree
    - Search ends at a nonempty subtree
    - Duplicates are not allowed
  - If item is not in AVL tree
    - Search ends at an empty subtree; insert the item there
- After inserting new item in the tree
  - Resulting tree might not be an AVL tree

AVL Trees: Insert 90

= ➔ bf = 0
< ➔ bf = -1
> ➔ bf = 1

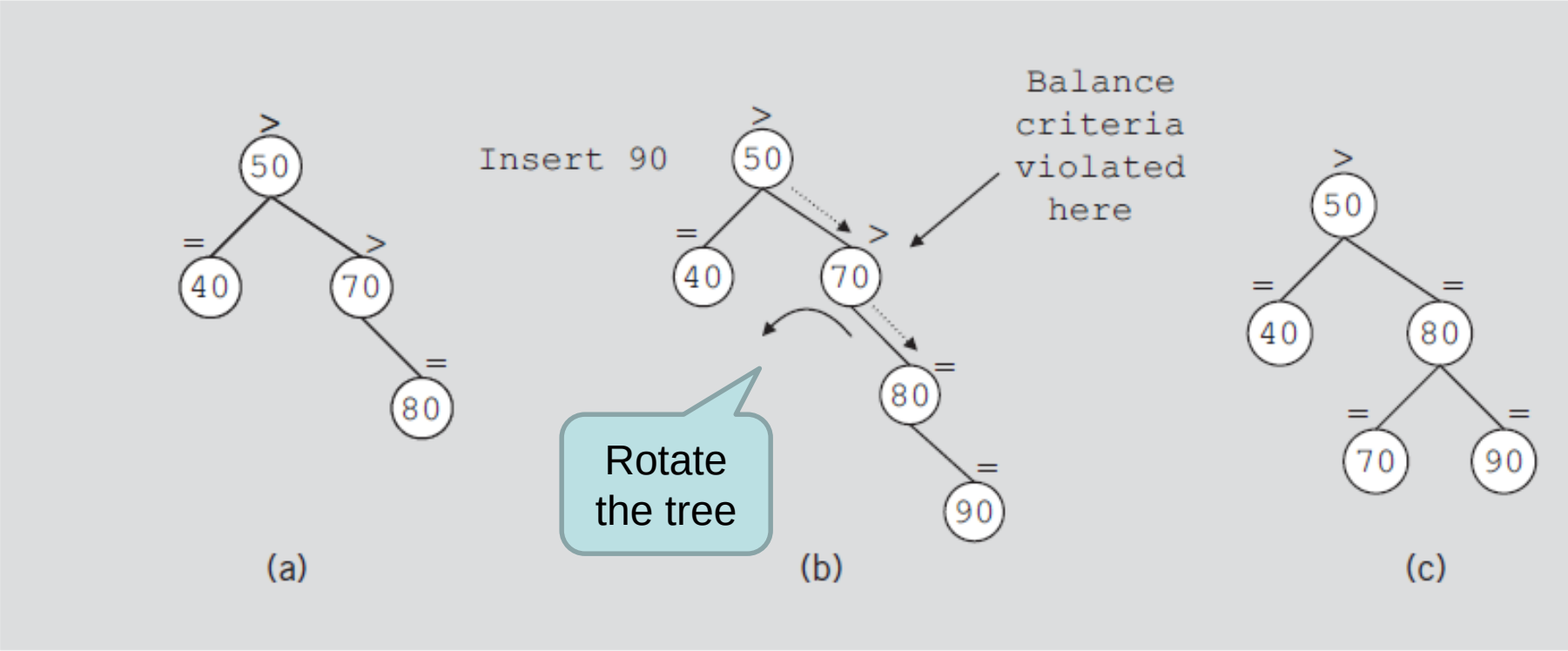


FIGURE 11-14 AVL tree before and after inserting 90

AVL Trees: Insert 75

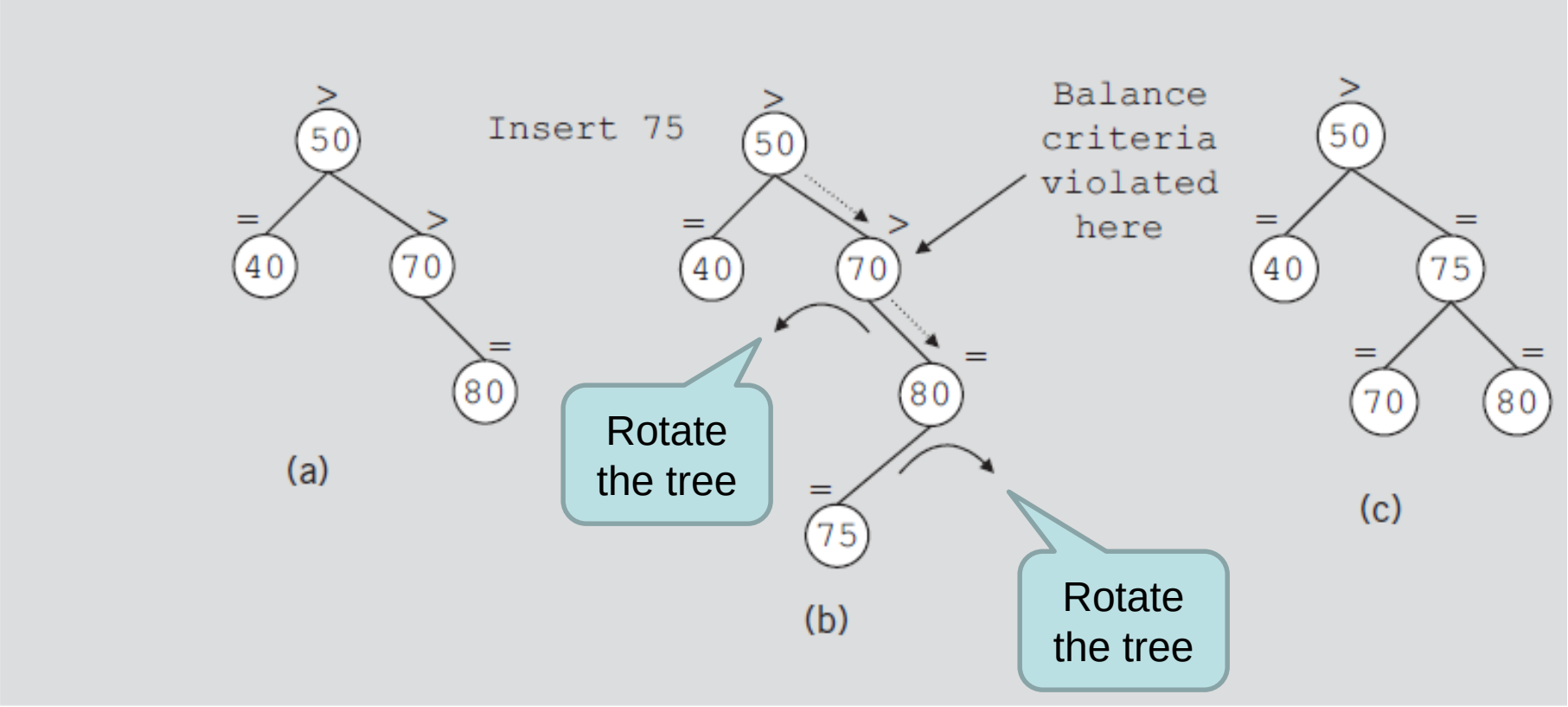


FIGURE 11-15 AVL tree before and after inserting 75

AVL Trees: Insert 95

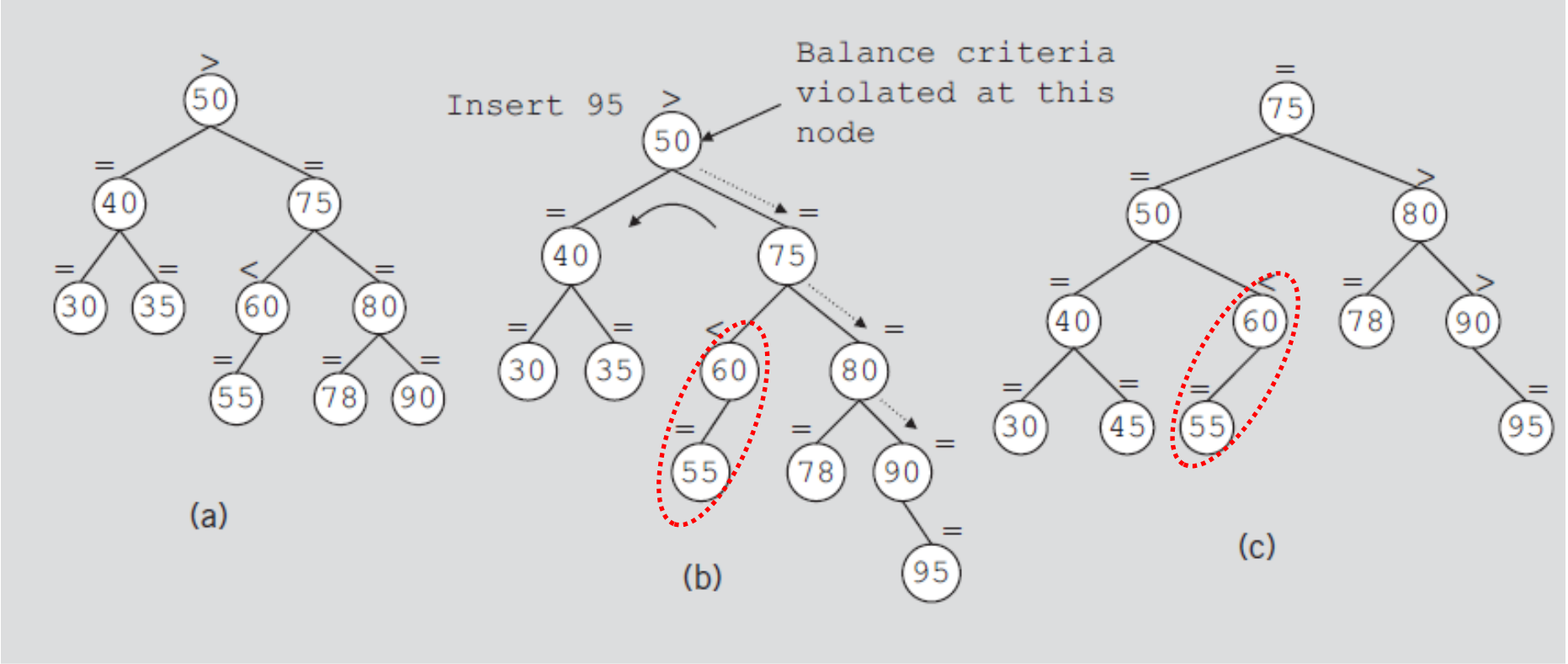


FIGURE 11-16 AVL tree before and after inserting 95

AVL Trees: Insert 88

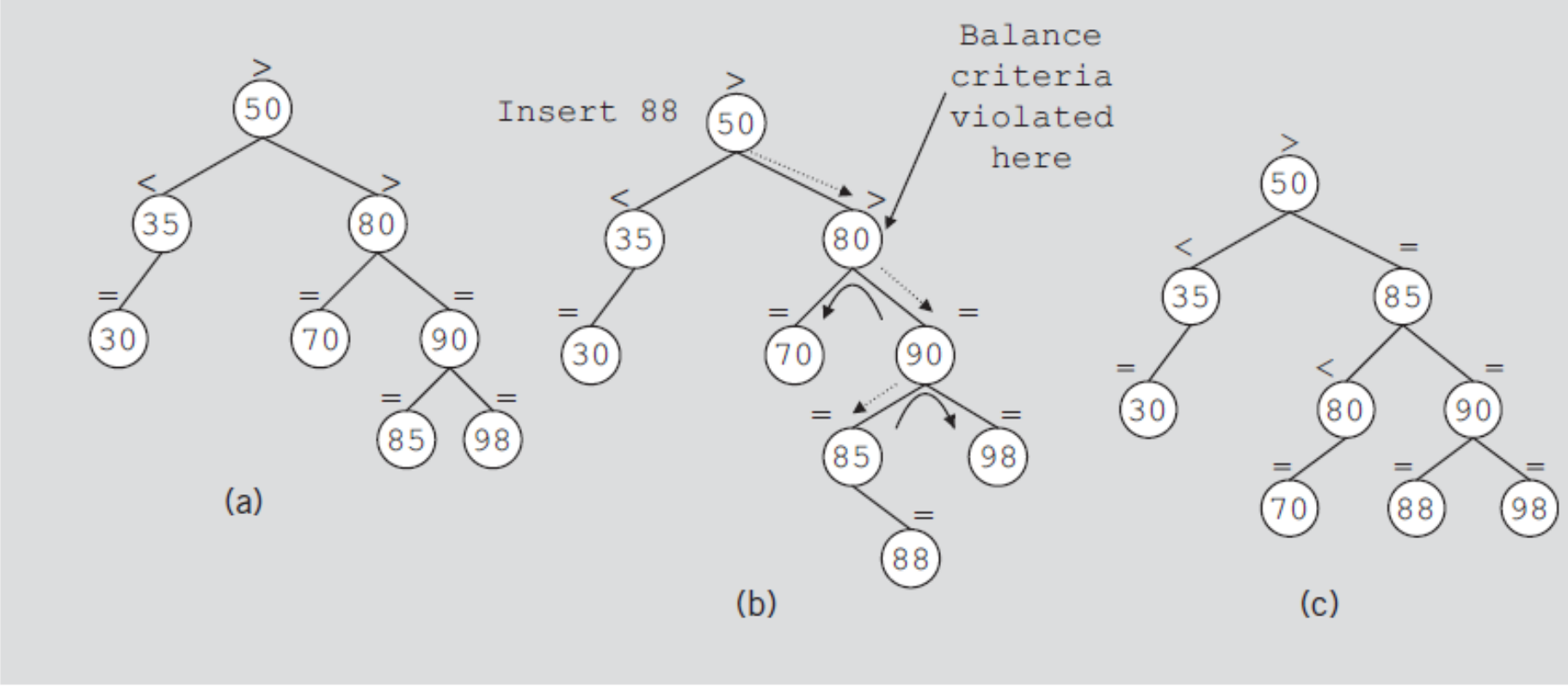


FIGURE 11-17 AVL tree before and after inserting 88

AVL Tree Rotations

- Rotating tree: reconstruction procedure
- Suppose that the rotation occurs at node x
  - Left rotation
    - Move certain nodes from the right subtree of x to its left subtree
    - The new root of the reconstructed tree: the root of the right subtree of x
  - Right rotation
    - Move certain nodes from the left subtree of x to its right subtree
    - The new root of the reconstructed tree: the root of the left subtree of x

Insertion on outside: left-left

bf(b) < 0
bf(a) < 0

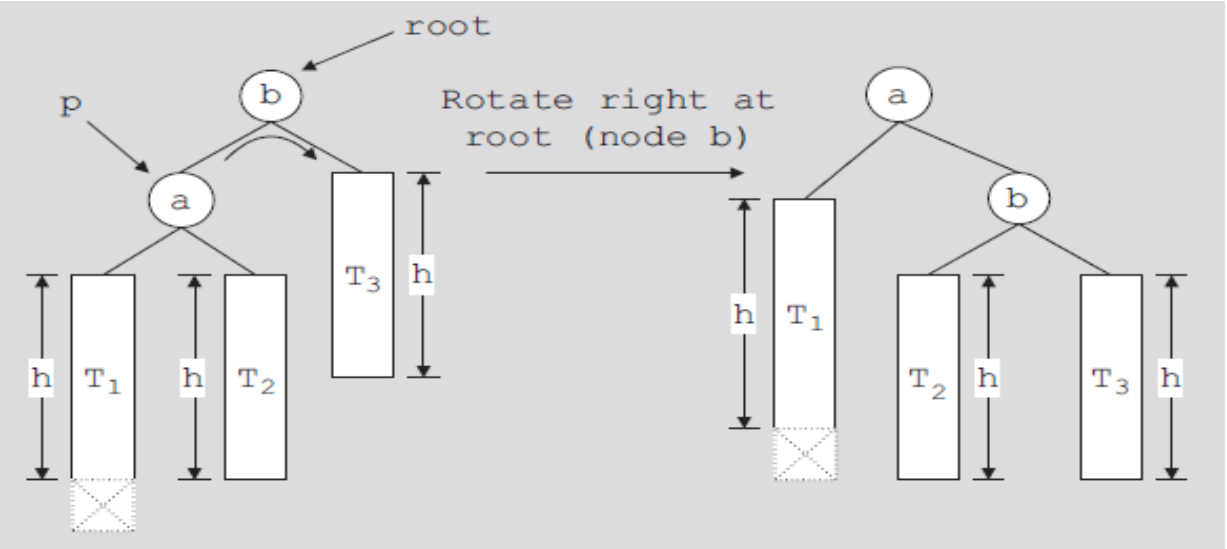


FIGURE 11-18 Right rotation at *b*

Insertion on outside: right-right

bf(b) > 0
bf(a) > 0

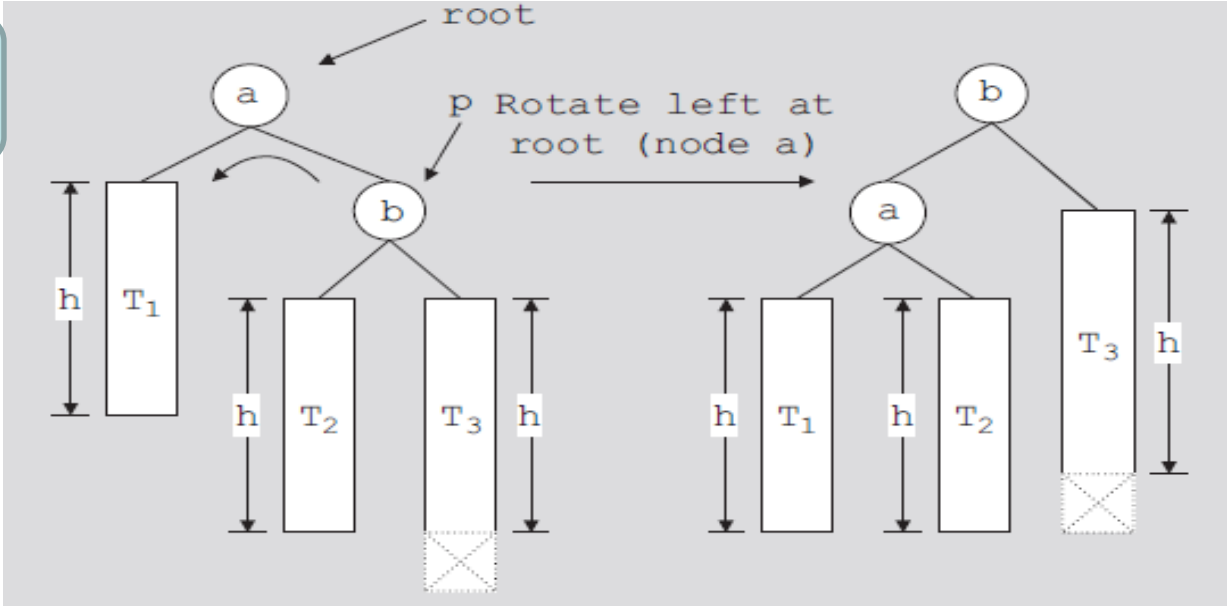


FIGURE 11-19 Left rotation at *a*

Insertion on inside: left-right

bf(c) < 0
bf(a) > 0

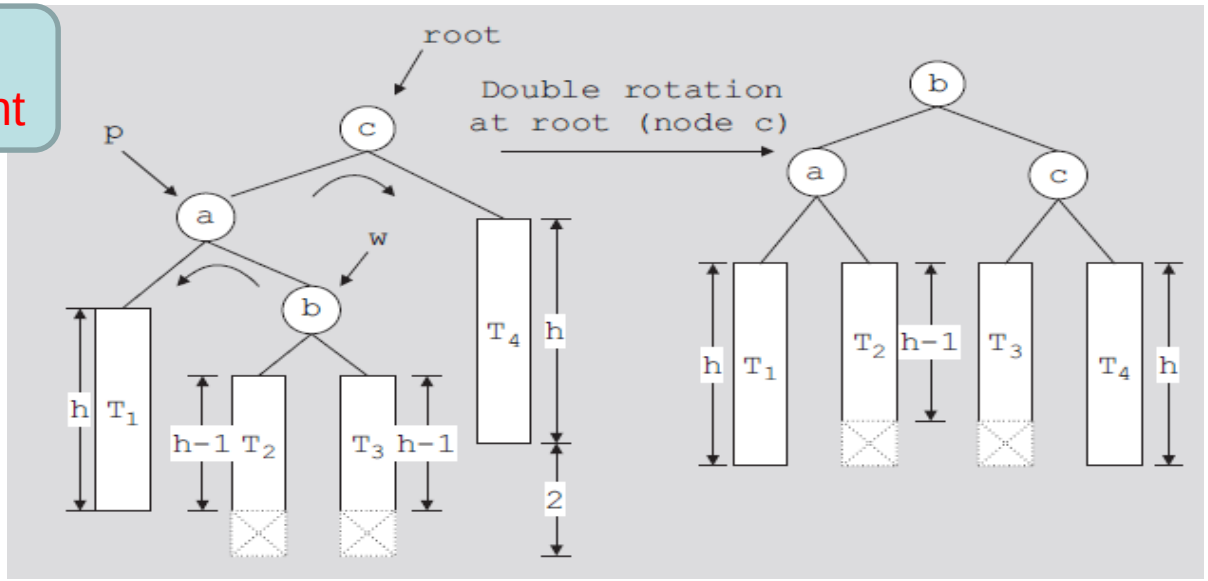


FIGURE 11-20 Double rotation: First rotate left at *a* and then right at *c*

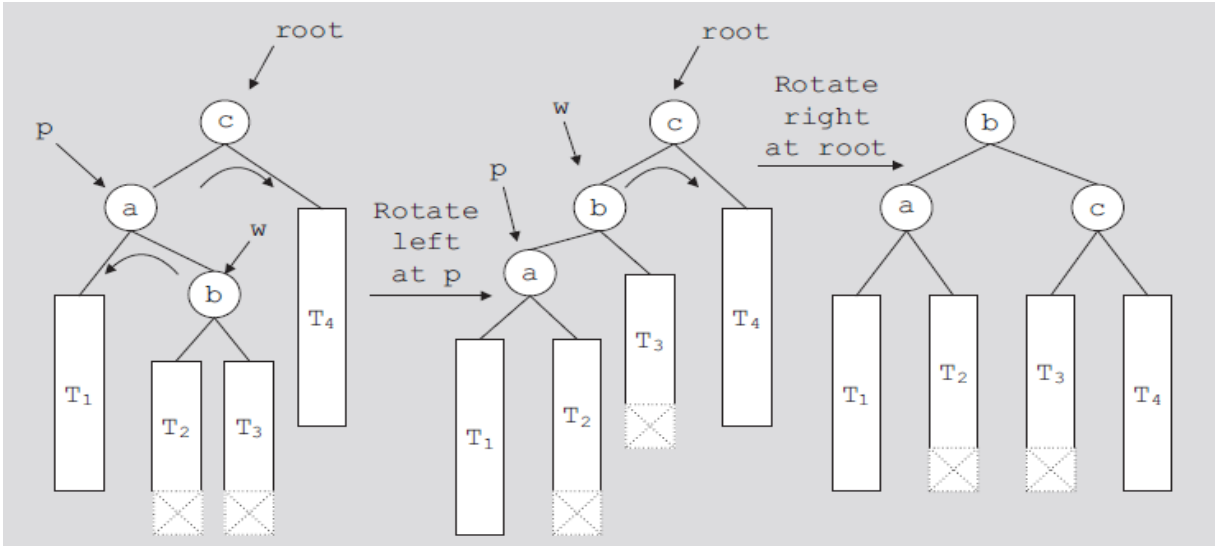


FIGURE 11-21 Left rotation at *a* followed by a right rotation at *c*

AVL Tree Rotations (cont'd.)

Insertion on inside: right-left

bf(a) > 0
bf(c) < 0

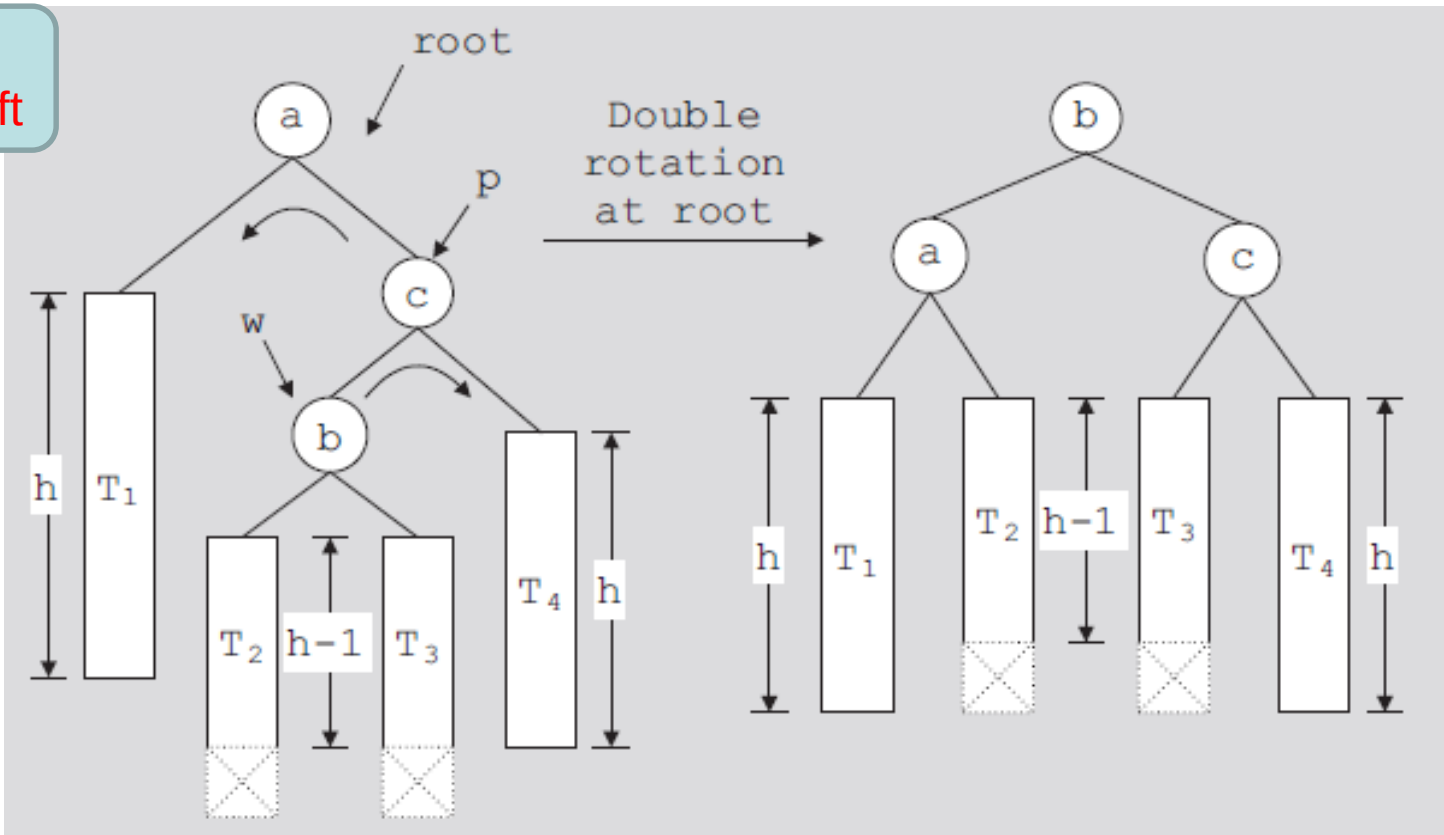


FIGURE 11-22 Double rotation: First rotate right at *c*, then rotate left at *a*

```
template <class elemT>
void rotateToLeft(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;  //pointer to the root of the
                        //right subtree of root
    if (root == NULL)
        cerr << "Error in the tree" << endl;
    else if (root->rlink == NULL)
        cerr << "Error in the tree:"
             <<" No right subtree to rotate." << endl;
    else
    {
        p = root->rlink;
        root->rlink = p->llink; //the left subtree of p becomes
                                //the right subtree of root
        p->llink = root;
        root = p;   //make p the new root node
    }
}//rotateLeft
```

Why call by ref?

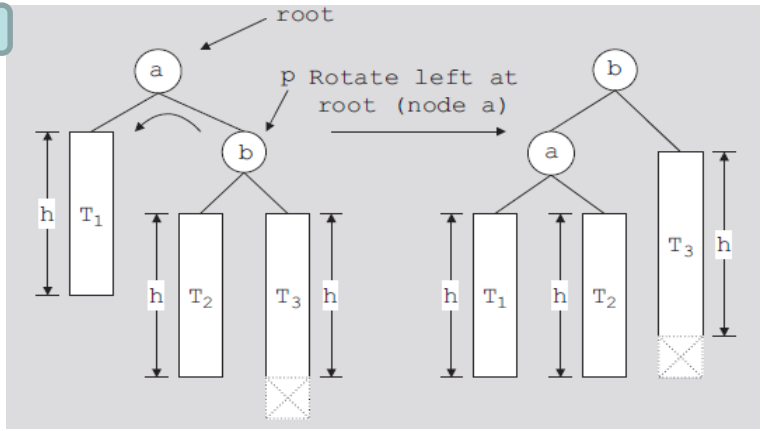


FIGURE 11-19 Left rotation at *a*

```
template <class elemT>
void rotateToRight(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;  //pointer to the root of
                        //the left subtree of root

    if (root == NULL)
        cerr << "Error in the tree" << endl;
    else if (root->llink == NULL)
        cerr << "Error in the tree:"
             << " No left subtree to rotate." << endl;
    else
    {
        p = root->llink;
        root->llink = p->rlink; //the right subtree of p becomes
                                //the left subtree of root
        p->rlink = root;
        root = p;   //make p the new root node
    }
}//end rotateRight
```

Why call by ref?

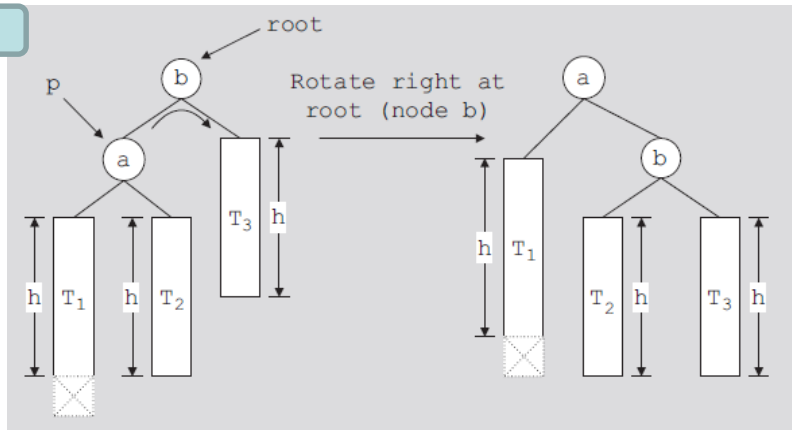


FIGURE 11-18 Right rotation at *b*

```
template <class elemT>
void balanceFromLeft(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;

    p = root->llink;   //p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToRight(root);
        break;

    case 0:
        cerr << "Error: Cannot balance from the left." << endl;
        break;

    case 1:
        w = p->rlink;
        switch (w->bfactor)  //adjust the balance factors
        {
        case -1:
            root->bfactor = 1;
            p->bfactor = 0;
            break;

        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;

        case 1:
            root->bfactor = 0;
            p->bfactor = -1;
        }//end switch

        w->bfactor = 0;
        rotateToLeft(p);
        root->llink = p;
        rotateToRight(root);
    }//end switch;
}//end balanceFromLeft
```

left left

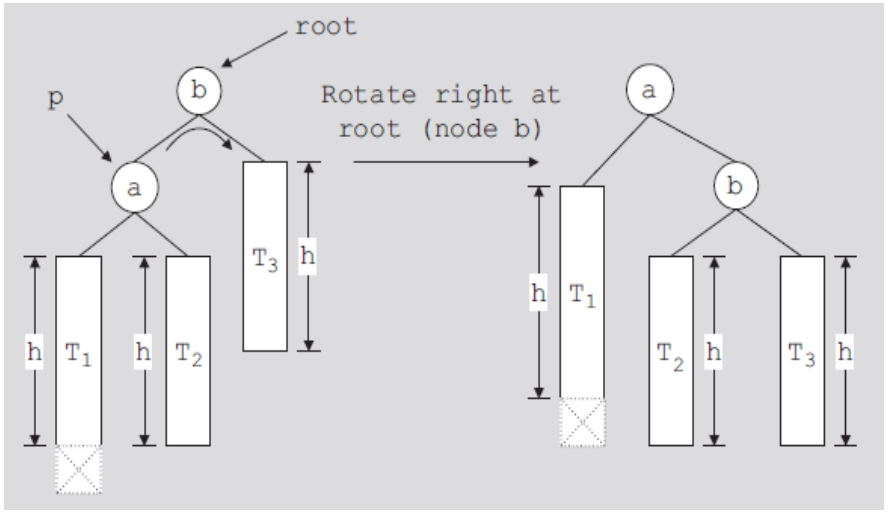


FIGURE 11-18 Right rotation at *b*

left right

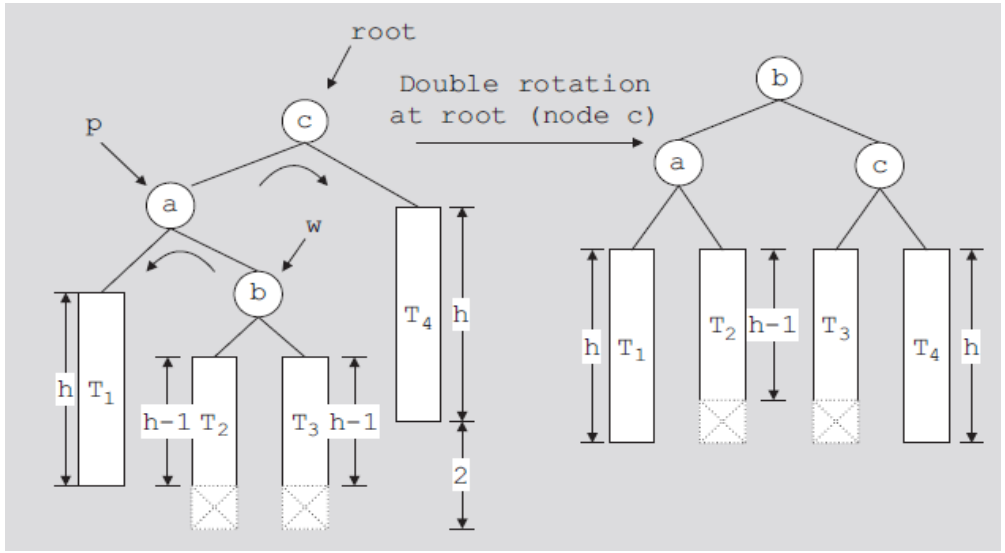


FIGURE 11-20 Double rotation: First rotate left at *a* and then right at *c*

```
template <class elemT>
void balanceFromRight(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;

    p = root->rlink;    //p points to the right subtree of root

    switch (p->bfactor)
    {
    case -1:
        w = p->llink;
        switch (w->bfactor)  //adjust the balance factors
        {
        case -1:
            root->bfactor = 0;
            p->bfactor = 1;
            break;

        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;

        case 1:
            root->bfactor = -1;
            p->bfactor = 0;
        }//end switch

        w->bfactor = 0;
        rotateToRight(p);
        root->rlink = p;
        rotateToLeft(root);
        break;

    case 0:
        cerr << "Error: Cannot balance from the left." << endl;
        break;

    case 1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToLeft(root);
    }//end switch;
}//end balanceFromRight
```

right left

right right

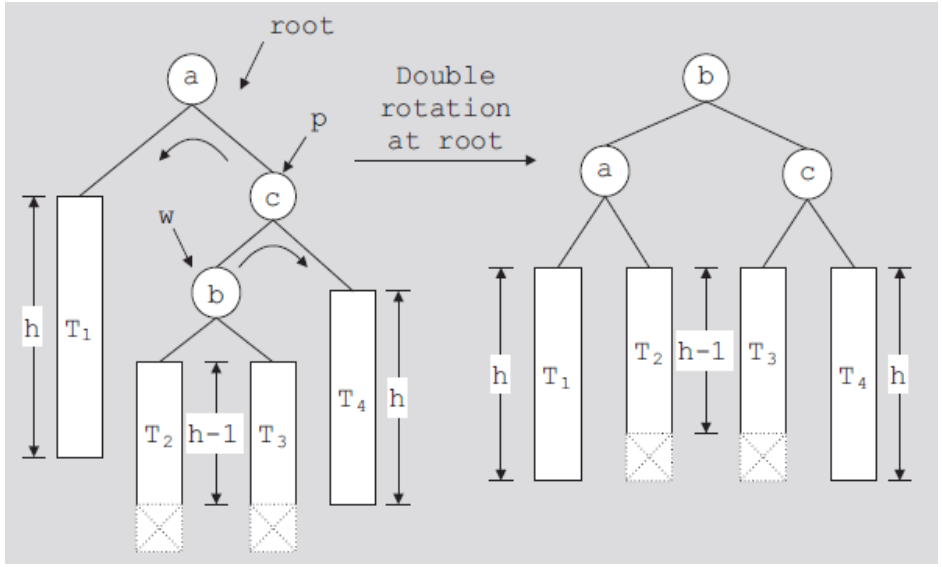


FIGURE 11-22 Double rotation: First rotate right at *c*, then rotate left at *a*

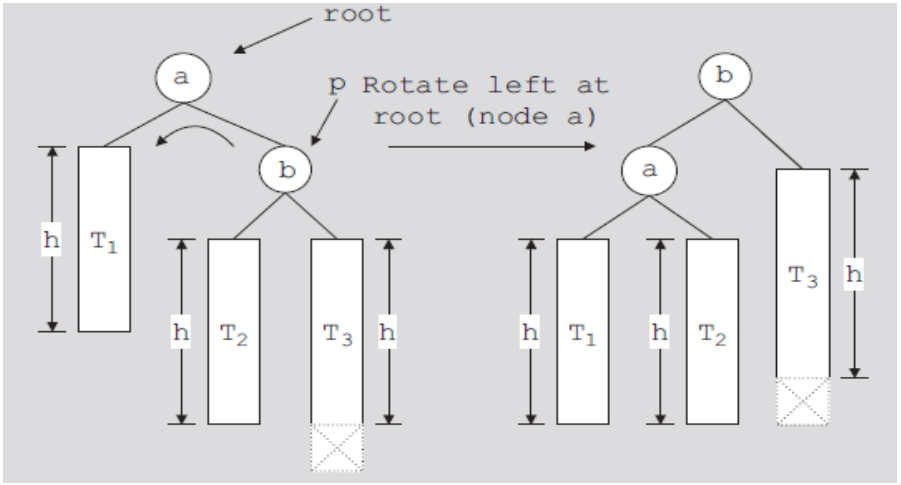


FIGURE 11-19 Left rotation at *a*

AVL Tree Rotations (cont'd.)

- Steps describing the function `insertIntoAVL`
  - Create node and copy item to be inserted into the newly created node
  - Search the tree and find the place for the new node in the tree
  - Insert new node in the tree
  - Backtrack the path, which was constructed to find the place for the new node in the tree, to the root node
    - If necessary, adjust balance factors of the nodes, or reconstruct the tree at a node on the path

```
template <class elemT>
void insertIntoAVL(AVLNode<elemT>* &root,
                   AVLNode<elemT>  *newNode, bool& isTaller)
{
    if (root == NULL)
    {
        root = newNode;
        isTaller = true;
    }
    else if (root->info == newNode->info)
        cerr << "No duplicates are allowed." << endl;
    else if (root->info > newNode->info) //newItem goes in
                                         //the left subtree
    {
        insertIntoAVL(root->llink, newNode, isTaller);

        if (isTaller) //after insertion, the subtree grew in heigh
            switch (root->bfactor)
            {
            case -1:
                balanceFromLeft(root);
                isTaller = false;
                break;

            case 0:
                root->bfactor = -1;
                isTaller = true;
                break;

            case 1:
                root->bfactor = 0;
                isTaller = false;
            }//end switch
        }//end if

    else
    {
        insertIntoAVL(root->rlink, newNode, isTaller);

        if (isTaller) //after insertion, the subtree grew in
                      //height
            switch (root->bfactor)
            {
            case -1:
                root->bfactor = 0;
                isTaller = false;
                break;

            case 0:
                root->bfactor = 1;
                isTaller = true;
                break;

            case 1:
                balanceFromRight(root);
                isTaller = false;
            }//end switch
        }//end else
}//insertIntoAVL
```

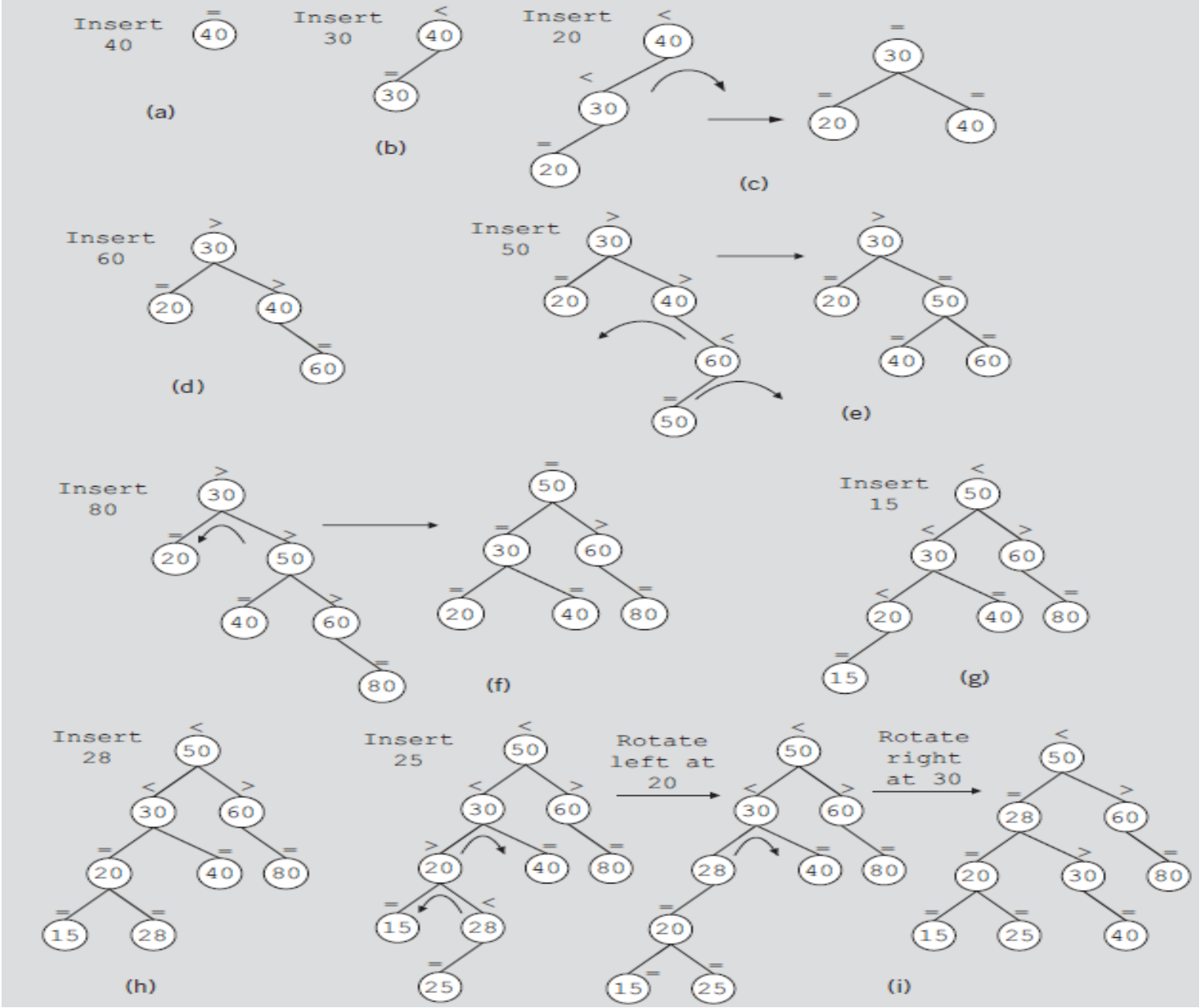


FIGURE 11-23 Item insertion into an initially empty AVL tree

AVL Tree Rotations (cont'd.)

- Function `insert`
  - Creates a node, stores the info in the node, and calls the function `insertIntoAVL` to insert the new node in the AVL tree

```
template <class elemT>
void insert(const elemT &newItem)
{
    bool isTaller = false;
    AVLNode<elemT>  *newNode;

    newNode = new AVLNode<elemT>;
    newNode->info = newItem;
    newNode->bfactor = 0;
    newNode->llink = NULL;
    newNode->rlink = NULL;

    insertIntoAVL(root, newNode, isTaller);
}
```

AVL Trees: Delete

- Four cases (similar to binary search tree)
  - Case 1: The node to be deleted is a leaf
  - Case 2: The node to be deleted has no right child, that is, its right subtree is empty
  - Case 3: The node to be deleted has no left child, that is, its left subtree is empty
  - Case 4: The node to be deleted has a left child and a right child
- Cases 1–3: easier
- Case 4:
  - reduce to case 2 (find the immediate predecessor), or case 3 (find the immediate successor)
  - Single or double rotation

Analysis: AVL Trees

- Consider all possible AVL trees of height *h*
  - Let T_h be an AVL tree of height *h* such that T_h has the *fewest* number of nodes
  - Let T_{hl} denote the left subtree of T_h and T_{hr} denote the right subtree of T_h
  - Then:

$$|T_h| = |T_{hl}| + |T_{hr}| + 1$$

where $|T_h|$ denotes the number of nodes in T_h

Analysis: AVL Trees (cont'd.)

- Suppose:
  - T_{hl} is of height $h - 1$ and T_{hr} is of height $h - 2$
    - T_{hl} is an AVL tree of height $h - 1$ such that T_{hl} has the fewest number of nodes among all AVL trees of height $h - 1$
    - T_{hr} is an AVL tree of height $h - 2$ that has the fewest number of nodes among all AVL trees of height $h - 2$
  - T_{hl} is of the form T_{h-1} and T_{hr} is of the form T_{h-2}
  - Hence:

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1$$

Clearly,

$$|T_0| = 1$$
$$|T_1| = 2$$

Let $F_{h+2} = |T_h| + 1$. Then,

$$F_{h+2} = F_{h+1} + F_h$$
$$F_2 = 2$$
$$F_3 = 3.$$

This is called a Fibonacci sequence. The solution to F_h is given by

$$F_h \approx \frac{\phi^h}{\sqrt{5}}, \text{ where } \phi = \frac{1+\sqrt{5}}{2}.$$

Hence,

$$|T_h| \approx \frac{\phi^{h+2}}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^{h+2}.$$

From this, it can be concluded that

$$h \approx (1.44)\log_2 |T_h|.$$

The height of AVL tree with n nodes: $(1.44)\log_2 n$
The height of perfectly balanced binary tree with n nodes: $\log_2 n$

m-way Search Trees

- ***m*-way search tree**
  - Tree in which each node has at most *m* children
    - If the tree is nonempty, it has the following properties:

1. Each node has the following form:

n	P_0	K_1	P_1	K_2	P_2	. . .	K_n	P_n

where $P_0, P_1, P_2, \ldots, P_n$ are pointers to the subtrees of the node, $K_1, K_2, \ldots, K_n$ are keys such that $K_1 < K_2 < \ldots < K_n$, and $n \leq m - 1$.

2. All keys, if any, in the node to which P_i points are less than K_{i+1}.
3. All keys, if any, in the node to which P_i points are greater than K_i.
4. The subtrees, if any, to which each P_i points are m-way search trees.

m-way search tree:
# of children/node: 0 .. m
# of keys/node: 1 .. m - 1

B-Trees

- Leaves on the same level
  - Not too far from the root
- ***B*-tree of order *m***
  - *m*-way search tree
  - Either empty or has the following properties:

1. All leaves are on the same level.
2. All internal nodes except the root have at most m (nonempty) children and at least $\lceil m/2 \rceil$ children. (Note that $\lceil m/2 \rceil$ denotes the ceiling of $m/2$.)
3. The root has at least 2 children if it is not a leaf, and at most m children.

B: Boeing or Bayer.
Not binary

- Basic operations
  - Search the tree
  - insert an item
  - delete an item
  - traverse the tree

B-tree of order m:
# of children/node: [m/2] .. m
# of keys/node: [m/2] - 1 .. m - 1

B-Trees or not?

Not B-tree since one internal node only has 2 children

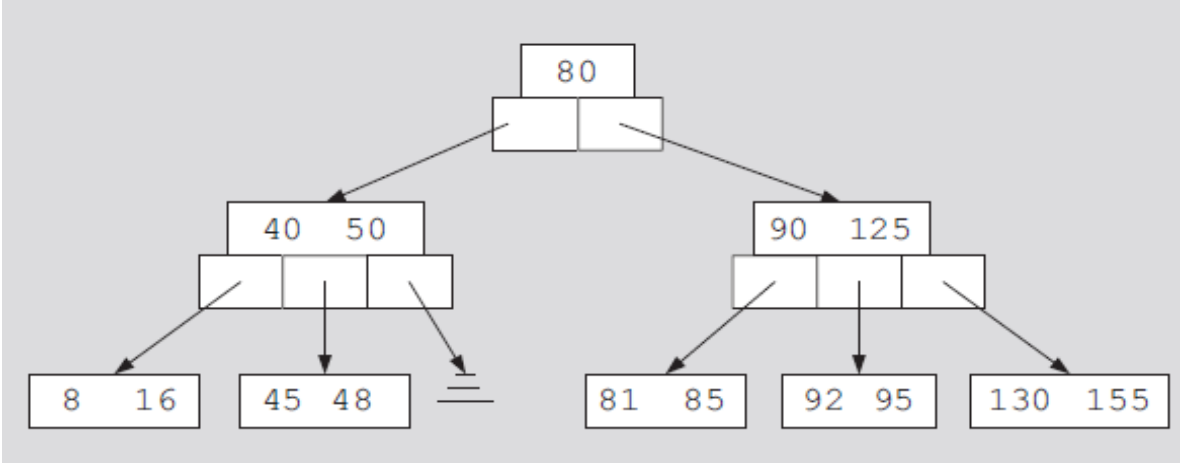


FIGURE 11-24 A 5-way **search tree**

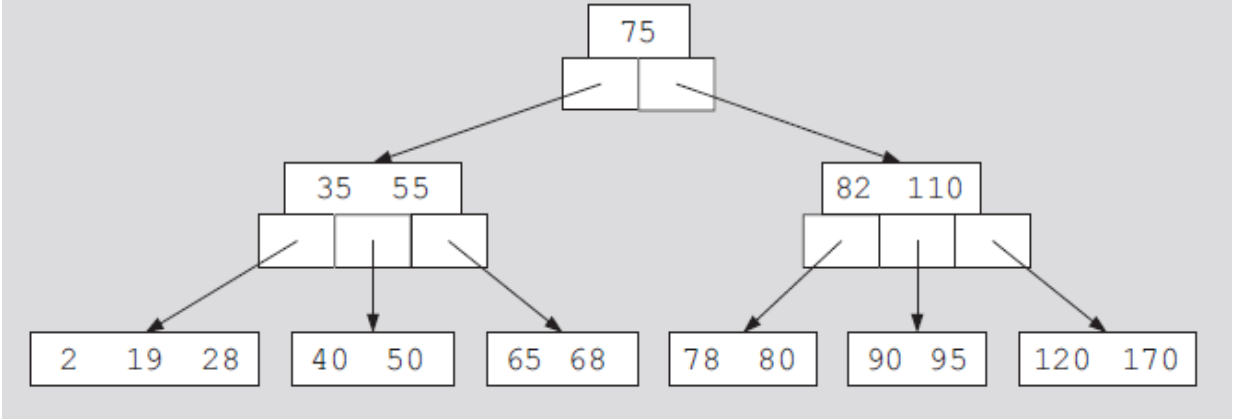


FIGURE 11-25 A **B-tree** of order 5

3 <= (# of children for internal node) <= 5

> 1 Template Parameters

- Template parameters
  - Data type
  - Constant expressions

```
template<class elemType, int size>
class listType
{
public:
     .
     .
     .
private:
     int maxSize;
     int length;
     elemType listElem[size];
};
```

```
listType<int, 100> intList;
```

B-Trees (cont'd.)

- Definition of a B-tree node

```
template <class recType, int bTreeOrder>
struct bTreeNode
{
    int recCount;
    recType list[bTreeOrder - 1];
    bTreeNode *children[bTreeOrder];
};
```

- Class implementing B-tree properties
  - Implements B-tree basic properties as an ADT
    - Search, insert, inOrder, etc

```
template <class recType, int bTreeOrder>
struct bTreeNode
{
    int recCount;
    recType list[bTreeOrder - 1];
    bTreeNode *children[bTreeOrder];
};

template <class recType, int bTreeOrder>
{
public:
    bool search(const recType& searchItem);
      //Function to determine if searchItem is in the B-tree.
      //Postcondition: Returns true if searchItem is found in the
      //    B-tree; otherwise, returns false.

    void insert(const recType& insertItem);
      //Function to insert insertItem in the B-tree.
      //Postcondition: If insertItem is not in the the B-tree, it
      //    is inserted in the B-tree.

    void inOrder();
      //Function to do an inorder traversal of the B-tree.

    bTree();
      //constructor

    //Add additional members as needed.

protected:
    bTreeNode<recType, bTreeOrder> *root;
};
```

B-tree: Search

- Binary search tree
  - Search must start at root
  - If item found in the binary search tree: returns true
  - Otherwise: returns false
- B-tree
  - Search must start at root node
  - More than one item (key) in a node (usually)
    - Must search array containing the data
  - Two functions required
    - Function `search`
    - Function `searchNode`
      - Searches a node sequentially

```
template <class recType, int bTreeOrder>
struct bTreeNode
{
    int recCount;
    recType list[bTreeOrder - 1];
    bTreeNode *children[bTreeOrder];
};
```

```
template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::searchNode
                    (bTreeNode<recType, bTreeOrder> *current,
                     const recType& item,
                     bool& found, int& location)
{
    location = 0;

    while (location < current->recCount
           && item > current->list[location])
        location++;

    if (location < current->recCount
           && item == current->list[location])
        found = true;
    else
        found = false;
} //end searchNode
```

```
template <class recType, int bTreeOrder>
bool bTree<recType, bTreeOrder>::search(const recType& searchItem)
{
    bool found = false;
    int location;

    bTreeNode<recType, bTreeOrder> *current;

    current = root;

    while (current != NULL && !found)
    {
        searchNode(current, item, found, location);

        if (!found)
            current = current->children[location];
    }

    return found;
} //end search
```

Traversing a B-Tree

- B-tree traversal methods
  - Inorder, preorder, postorder

```
template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::inOrder()
{
    recInorder(root);
} // end inOrder

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::recInorder
                        (bTreeNode<recType, bTreeOrder> *current)
{
    if (current != NULL)
    {
        recInorder(current->children[0]);

        for (int i = 0; i < current->recCount; i++)
        {
            cout << current->list[i] << " ";

            recInorder(current->children[i + 1]);
        }
    }
} //end recInorder
```

B-tree: Insert

- Algorithm: search tree to see if key already exists
  - If key already in the tree: output an error message
  - If key not in the tree: search terminates at a leaf
  - Record inserted into the leaf: if room exists
    - If leaf full: split node into two nodes
    - Median key moved to parent node (median determined by considering all keys in the node and new key)
  - Splitting can propagate upward (even to the root)
    - Causing tree to increase in height

B-tree: Insert (cont'd.)

- Function `insertBTree`
  - Recursively inserts an item into a B-tree
- Function `insert` uses function `insertBTree`
- Function `insertNode`
  - Inserts item in the node
- Function `splitNode`
  - Splits node into two nodes
  - Inserts new item in the relevant node
  - Returns median key and pointer to second half of the node

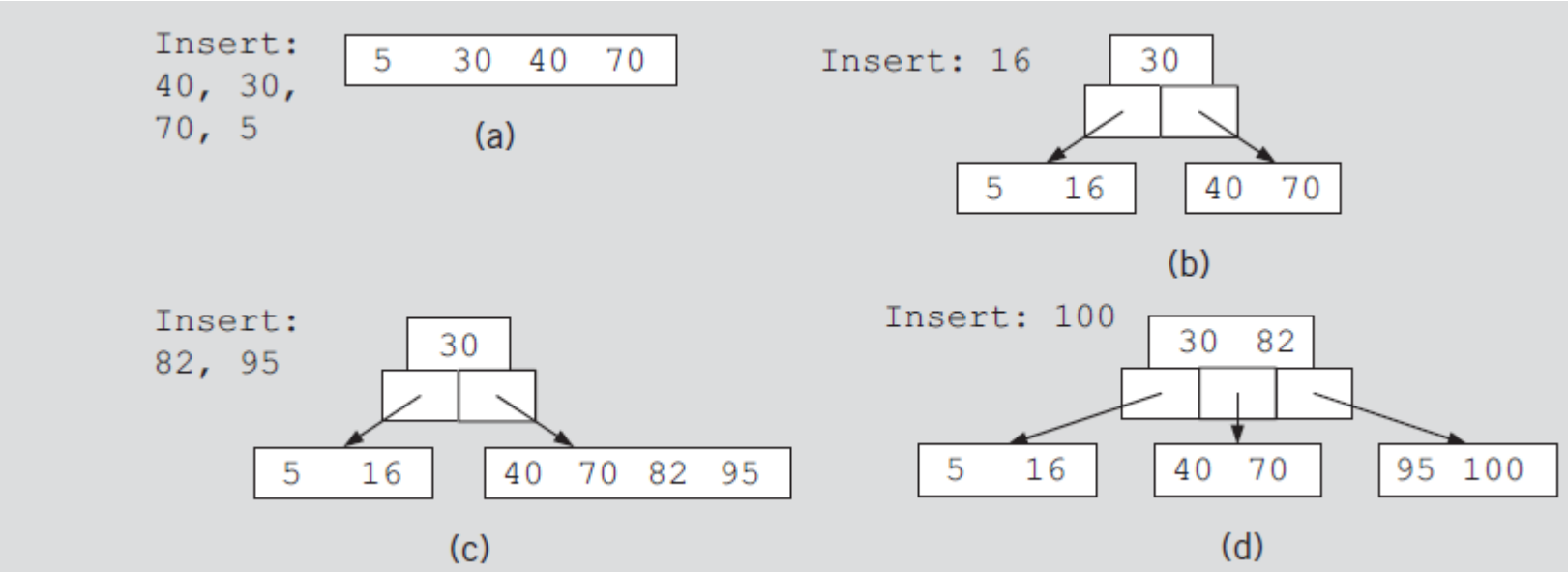


FIGURE 11-26 Item insertion into a B-tree of order 5

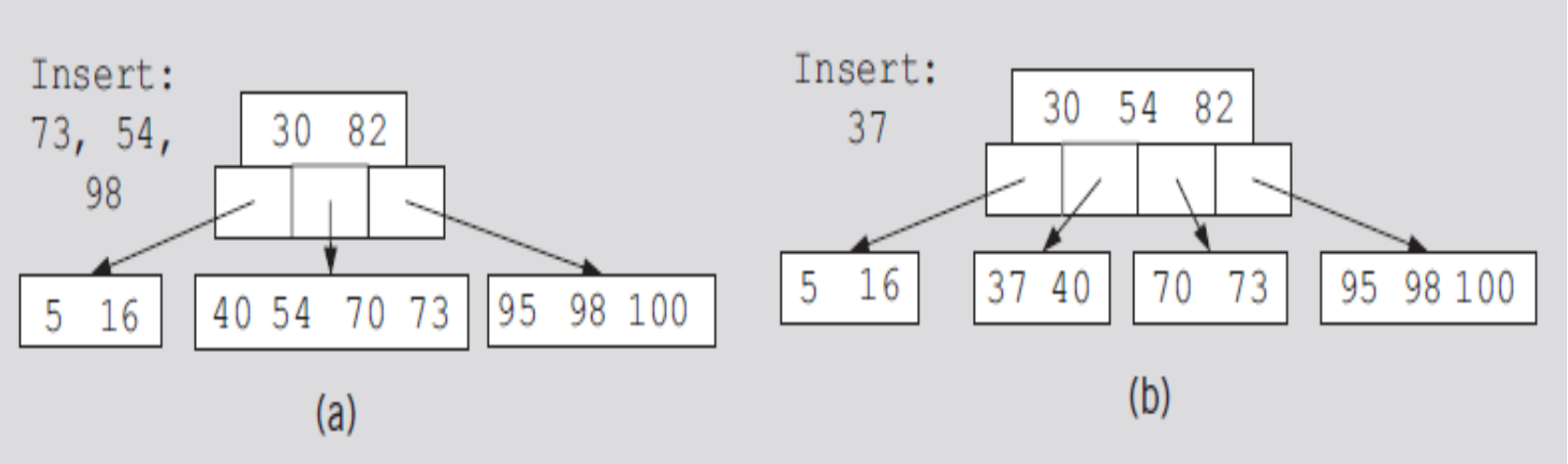


FIGURE 11-27 Insertion of *73*, *54*, *98*, and *37*

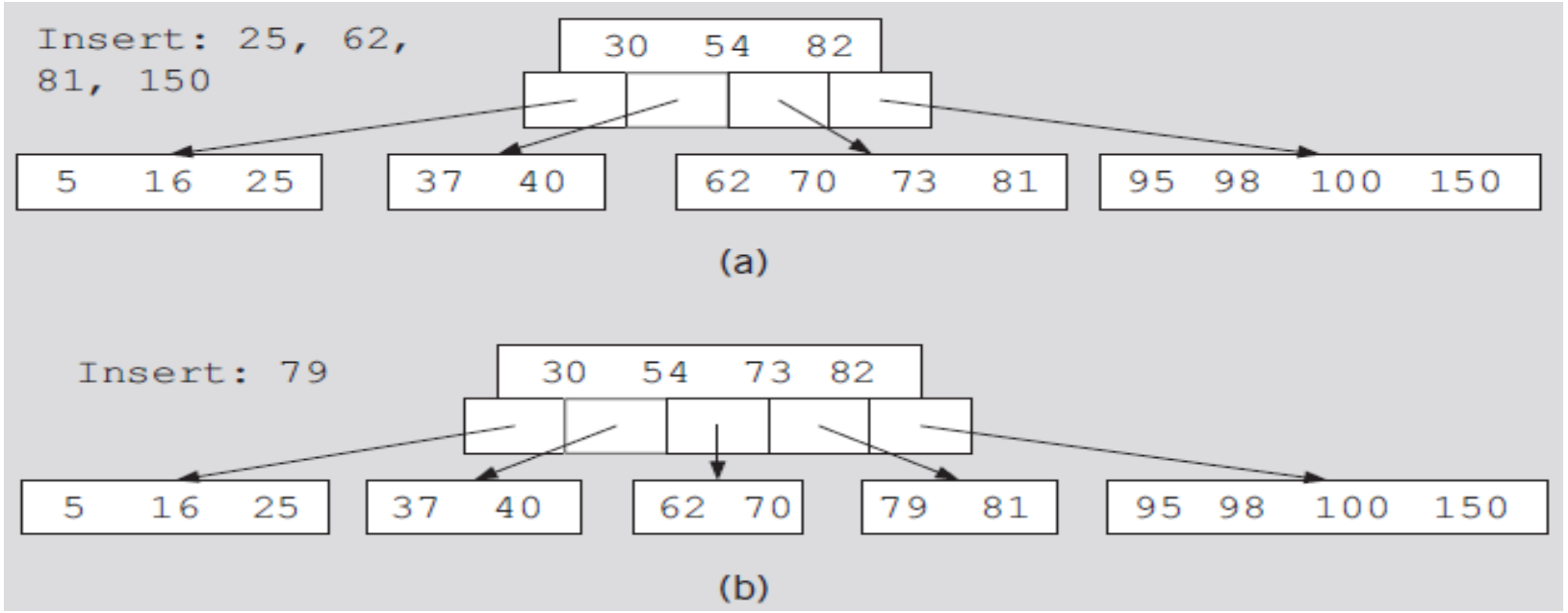


FIGURE 11-28 Insertion of *25*, *62*, *81*, *150*, and *79*

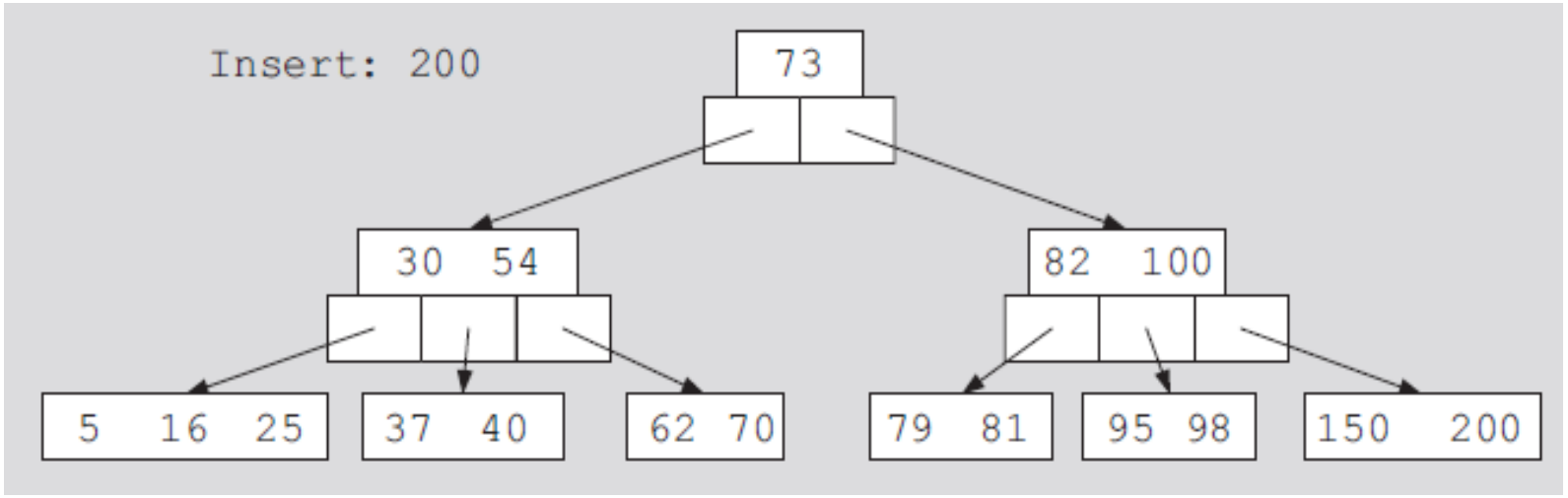


FIGURE 11-29 Insertion of *200*

B-tree: Delete

- Case to consider

1. If `deleteItem` is not in the tree, output an appropriate error message.
2. If `deleteItem` is in the tree, find the node containing the `deleteItem`. If the node containing the `deleteItem` is not a leaf, its immediate predecessor (or successor) is in a leaf. So we can swap the immediate predecessor (or successor) with the `deleteItem` to move the `deleteItem` into a leaf. We consider the cases to delete an item from a leaf.
   a. If the leaf contains more than the minimum number of keys, delete the `deleteItem` from the leaf. (In this case, no further action is required.)
   b. If the leaf contains only the minimum number of keys, look at the sibling nodes that are adjacent to the leaf. (Note that the sibling nodes and the leaf have the same parent node.)
      i. If one of the sibling nodes has more than the minimum number of keys, move one of the keys from that sibling node to the parent and one key from the parent to the leaf, and then delete `deleteItem`.
      ii. If the adjacent siblings have only the minimum number of keys, then combine one of the siblings with the leaf and the median key from the parent. If this action does not leave the minimum number of keys in the parent node, this process of combining the nodes propogates upward, possibly as far as the root node, which could result in reducing the height of the B-tree.

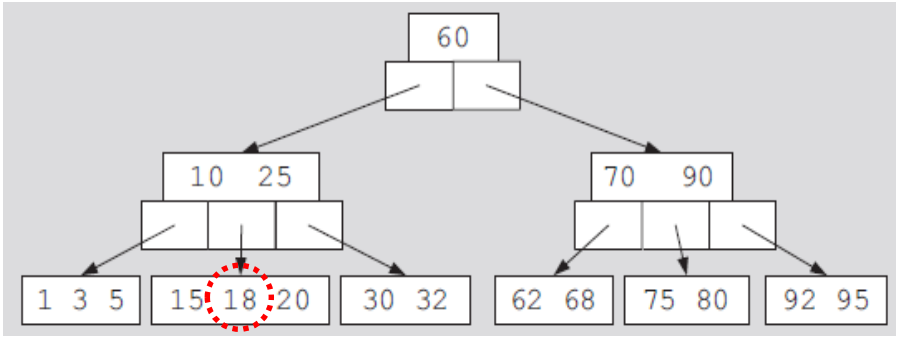


FIGURE 11-30 A B-tree of order 5

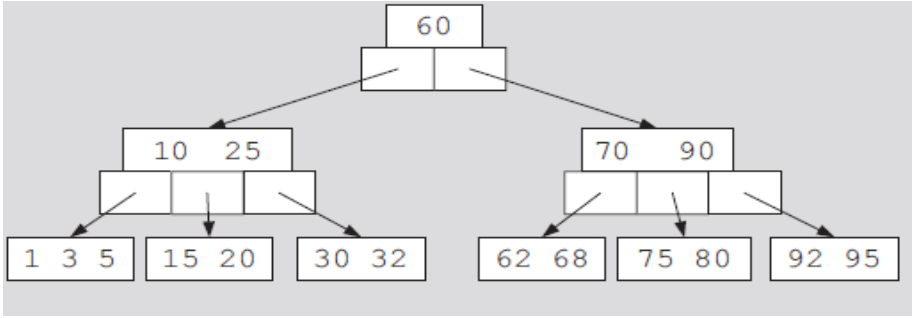


FIGURE 11-31 Deleting `18` from a B-tree of order 5

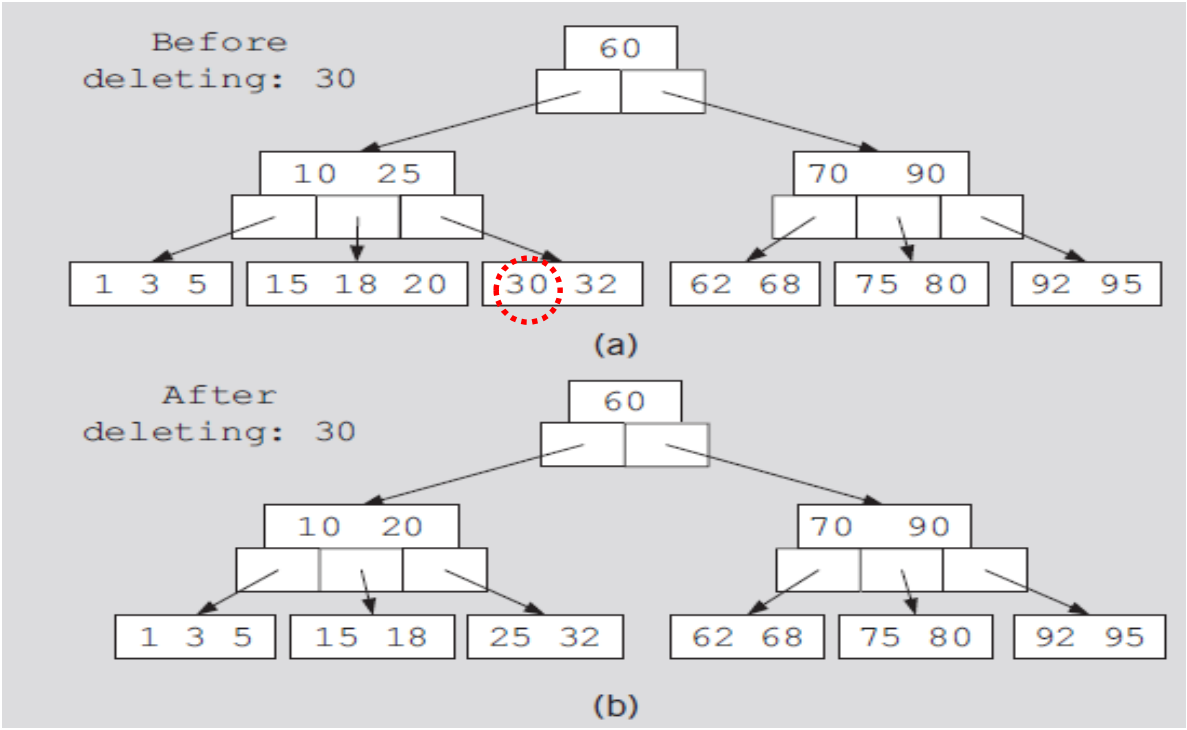


FIGURE 11-32 B-tree before and after deleting `30`

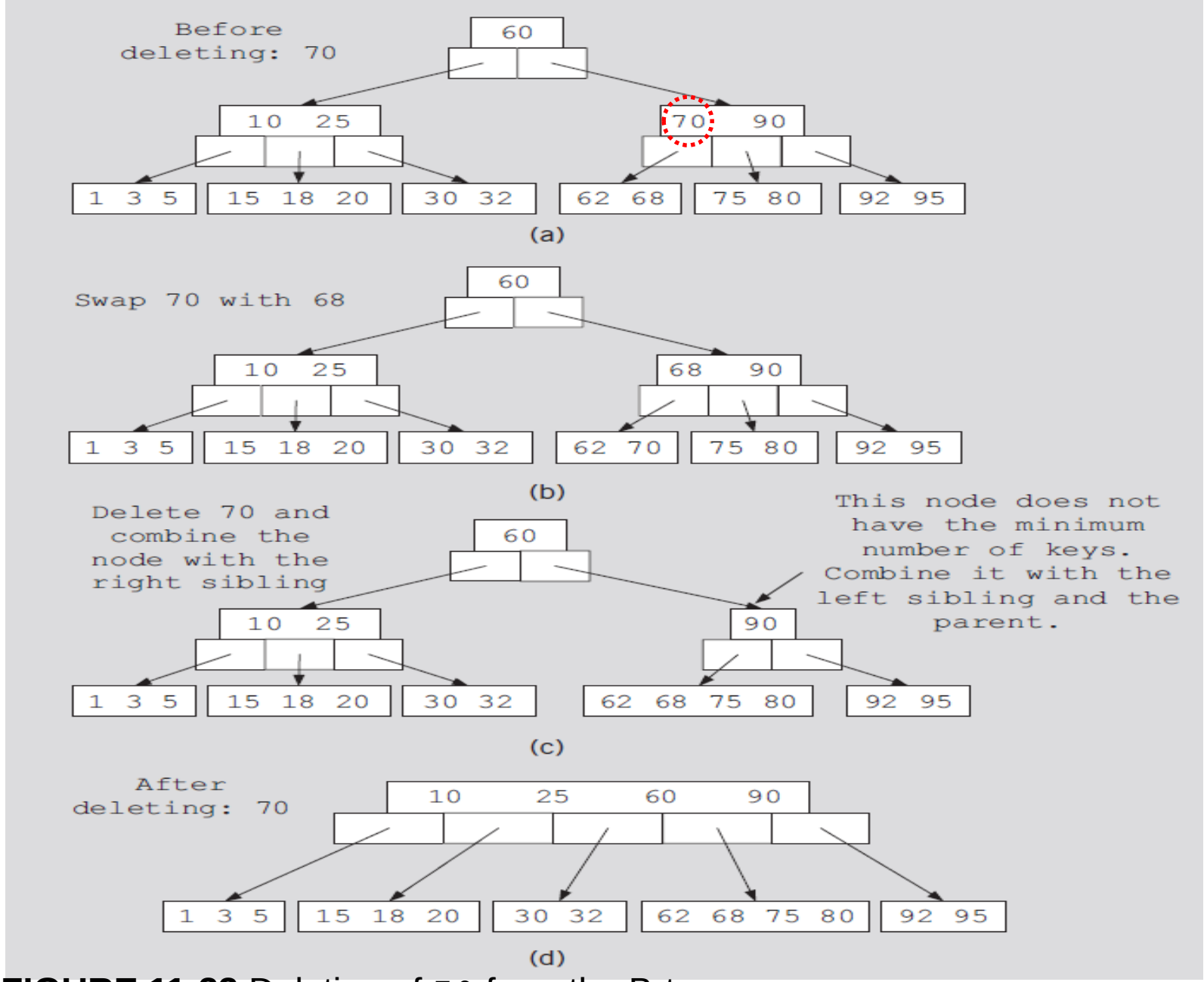
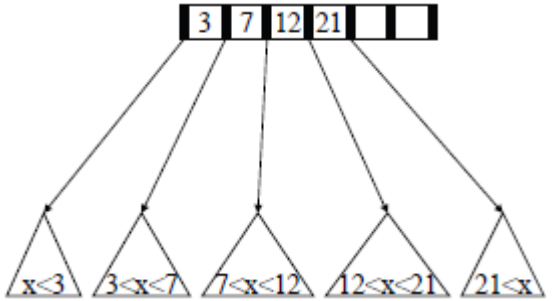


FIGURE 11-33 Deletion of `70` from the B-tree

B+-Tree

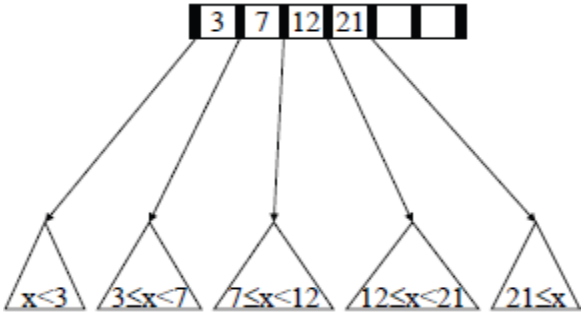
- B-Tree
  - Root + internal nodes + leaves: key + data
  - Subtree between key x and y contains leaves with value v:

    x < v < y
  - Leaves at the same depth

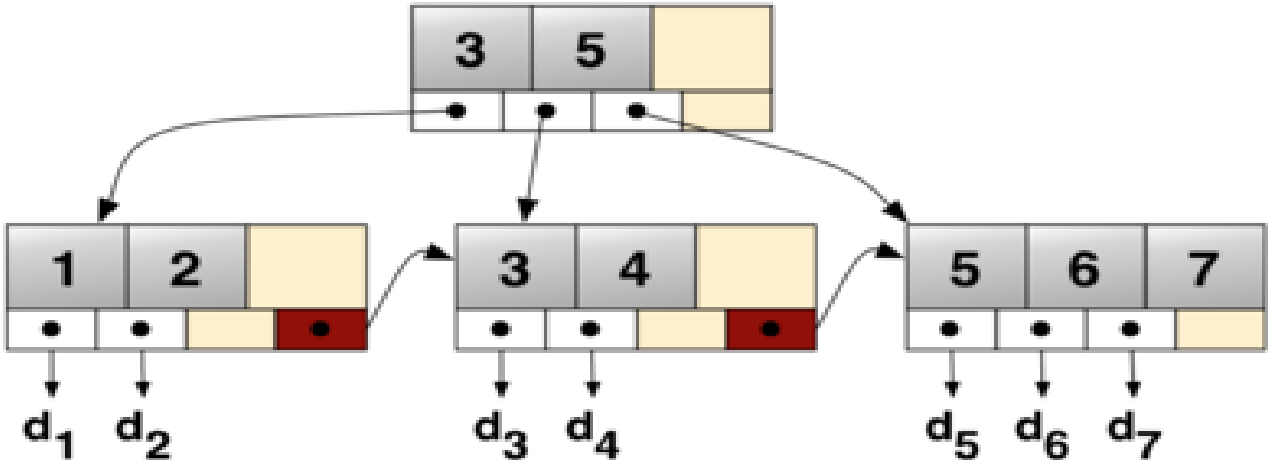


- B+-Tree: Similar to B-Tree, except
  - Root + internal nodes: no data
  - Leaf: key + data (may have more # of key + data)
  - Subtree between key x and y contains leaves with value v:

    x <= v < y
  - Links to sibling/neighboring nodes



B+-Tree (cont'd)



- Allow concurrent modification
  - Good for storing data for efficient retrieval in block-oriented storage, e.g., file systems and DBs
  - Index in DB

Summary

- Binary trees
  - Level, height, copyTree
  - traversal algorithms: inorder, preorder, postorder
- Binary search trees
  - Search, insert, delete
  - Time complexity
- Recursive traversal algorithms
- Nonrecursive traversal algorithms
- AVL trees
  - Insert, rotation, delete
- B-trees
  - Search, insert, delete
- B+-Tree

Self Exercises

- Programming Exercises: 1, 2, 3, 4, 16