Trees

- Linear Vs non-linear data structures
- Types of binary trees
- Binary tree traversals
- Representations of a binary tree
- Binary tree ADT
- Binary search tree

Overview

- We have discussed *linear* data structures
  - arrays, linked lists, stacks, queues
- Some other data structures we will consider
  - trees, tables, graphs, hash-tables
- Trees are extremely useful and suitable for a wide range of applications
  - sorting, searching, expression evaluation, data set representation
  - especially well suited to recursive algorithm implementation

Terminology

- A Tree T is a set of n >= 0 elements:
  - if n == 0, T is an empty tree
  - if n > 0 then there exists some element called $r \in T$ called the root of T such that T - {r} can be partitioned into zero or more disjoint sets T_1 ,T_2 , ... where each subset forms a tree
- Trees are composed of nodes and edges
- Trees are hierarchical
  - parent-child relationship between two nodes
  - ancestor-descendant relationships among nodes
- Subtree of a tree: Any node and its descendants

Terminology

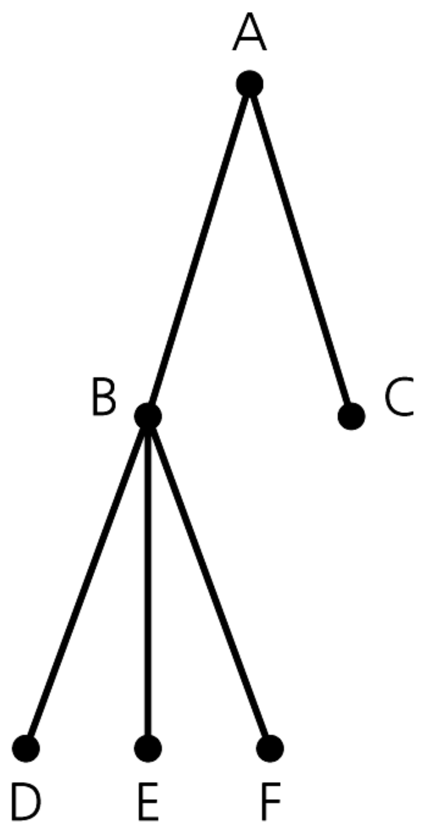


Figure 10-1 A general tree

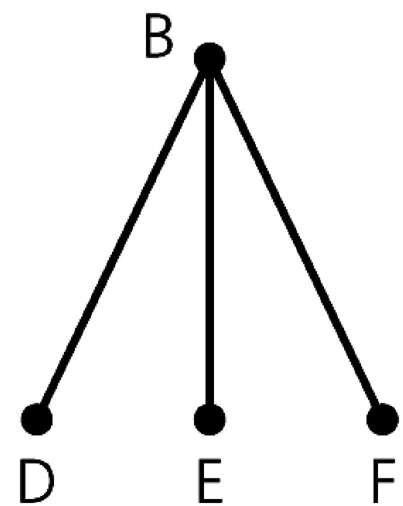


Figure 10-2

A subtree of the tree in Figure 10-1

Terminology

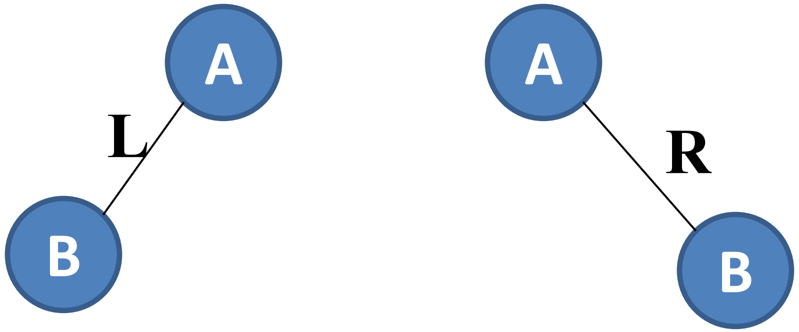
- Parent of node n
  - The node directly above node n in the tree
- Child of node n
  - A node directly below node n in the tree
- Root
  - The only node in the tree with no parent
- Subtree of node n
  - A tree that consists of a child (if any) of node n and the child's descendants

Terminology

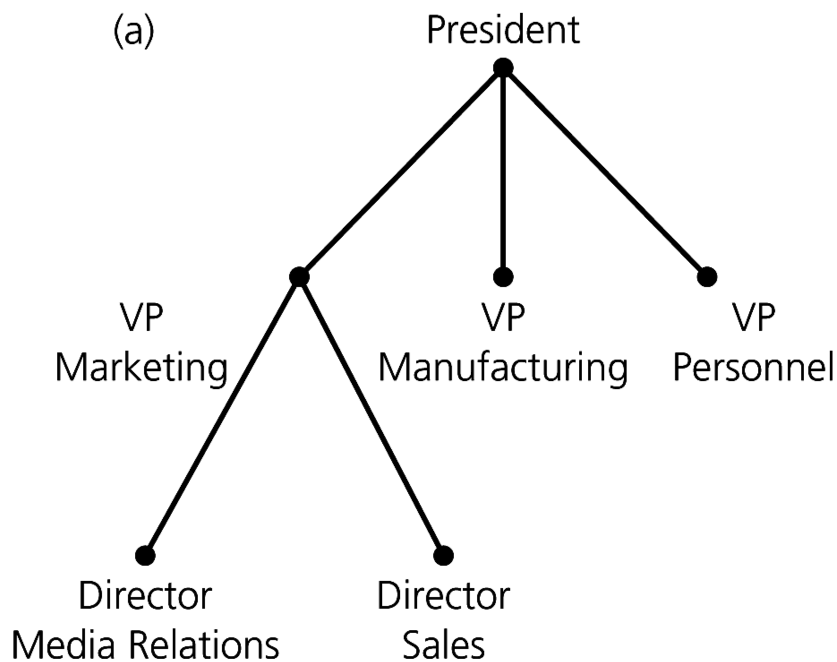
- Leaf
  - A node with no children
- Siblings
  - Nodes with a common parent
- Ancestor of node n
  - A node on the path from the root to n
- Descendant of node n
  - A node on a path from n to a leaf

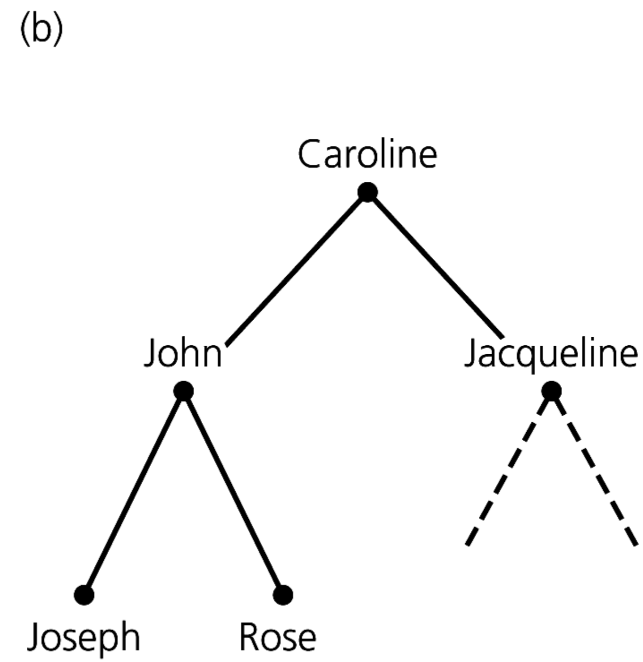
A Binary Tree

- A binary tree is a set T of nodes such that
  - T is empty, or
  - T is partitioned into three disjoint subsets:
    - a single node r, the root
    - two possibly empty sets that are binary trees, called the left subtree of r and the right subtree of r
- Binary trees are ordered
- These trees are not equal



A General Tree & A Binary Tree





More Binary Trees

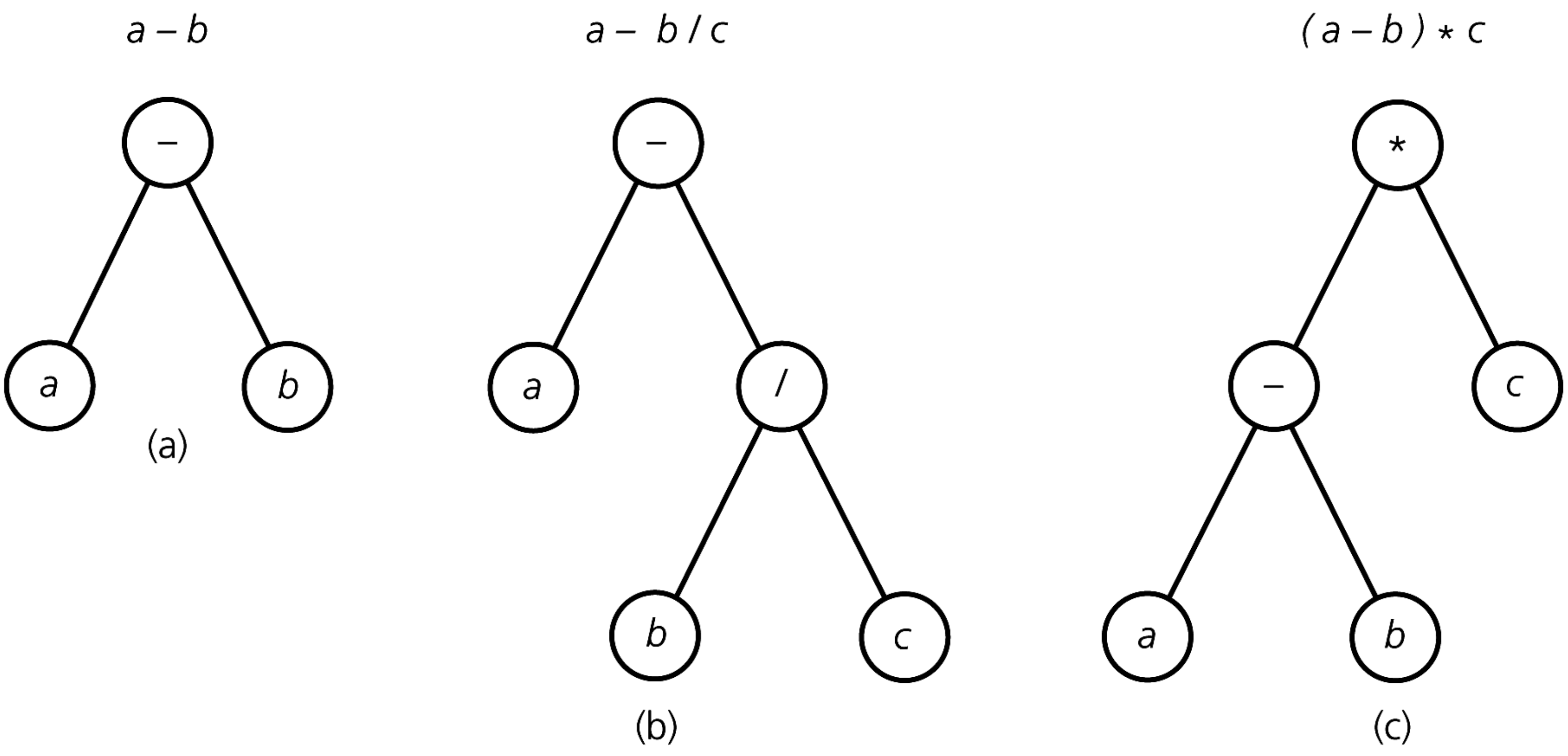
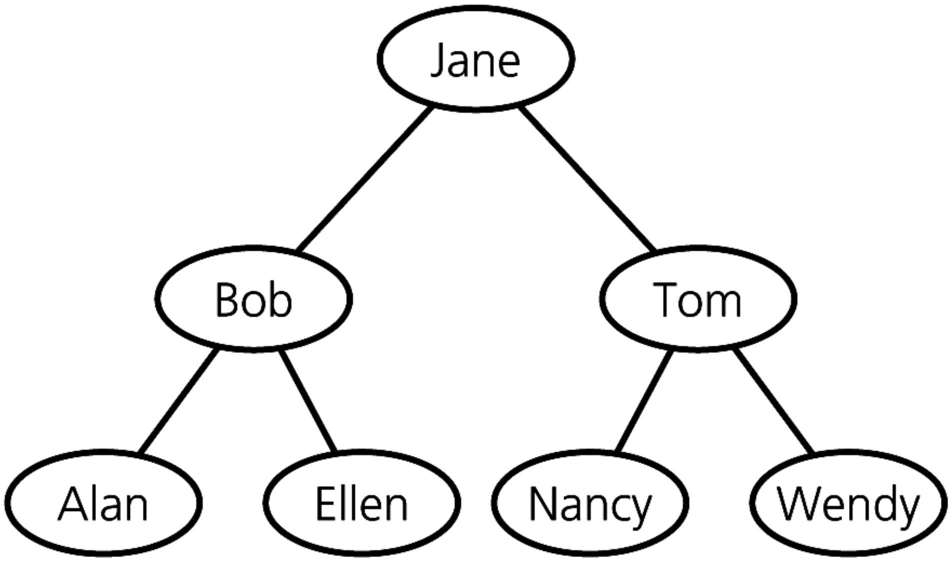


Figure 10-4 Binary trees that represent algebraic expressions

A Binary Search Tree

- A binary search tree is a binary tree that has the following properties for each node n
  - n’s value is > all values in n’s left subtree TL
  - n’s value is < all values in n’s right subtree TR
  - both TL and TR are binary search trees



The Height of Trees

- Height of a tree
  - Number of nodes along the longest path from the root to a leaf

Height 3 Height 5 Height 7

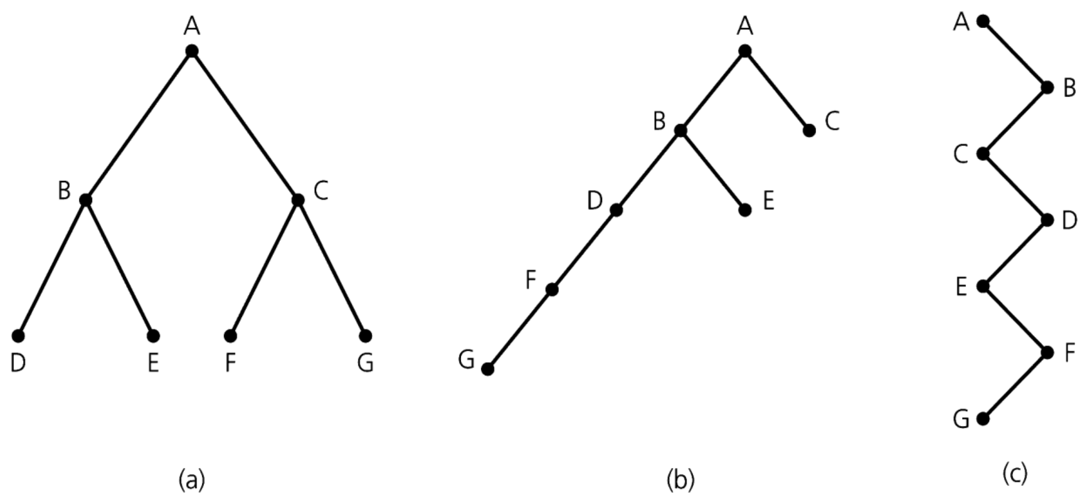


Figure 10-6

Binary trees with the same nodes but different heights

The Height of Trees

- Level of a node n in a tree T
  - If n is the root of T, it is at level 1
  - If n is not the root of T, its level is 1 greater than the level of its parent
- Height of a tree T defined in terms of the levels of its nodes
  - If T is empty, its height is 0
  - If T is not empty, its height is equal to the maximum level of its nodes

The Height of Trees

- A recursive definition of height
  - If T is empty, its height is 0
  - If T is not empty,
  - height(T) = 1 + max{height(TL), height(TR)}

```
         r
       /   \
     TL      TR
```

Full Binary Trees

- A binary tree of height h is full if
  - Nodes at levels < h have two children each
- Recursive definition
  - If T is empty, T is a full binary tree of height 0
  - If T is not empty and has height h > 0, T is a full binary tree if its root's subtrees are both full binary trees of height h – 1

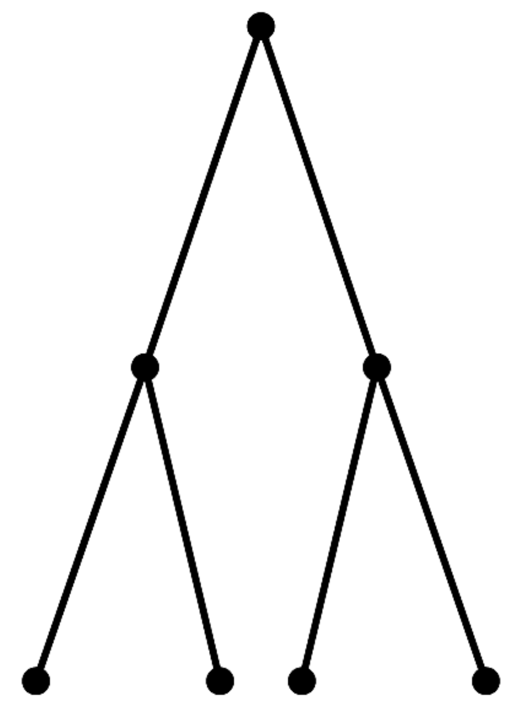
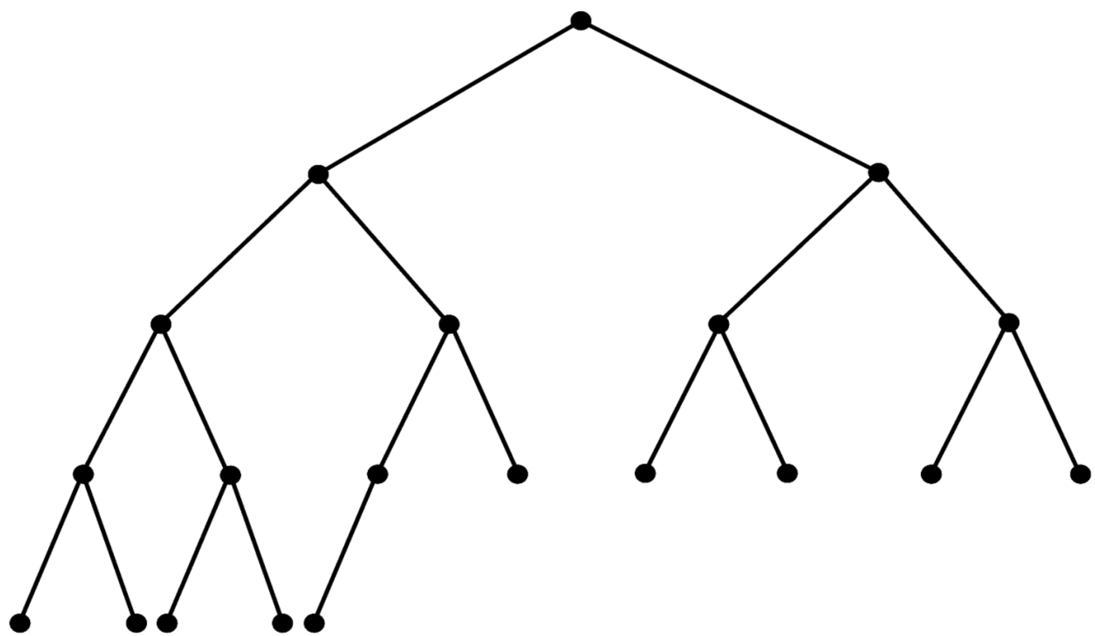


Figure 10-7

A full binary tree of height 3

Complete Binary Trees

- A binary tree of height h is complete if
  - It is full to level h – 1, and
  - Level h is filled from left to right



Complete Binary Trees

- Another definition:
- A binary tree of height h is complete if
  - All nodes at levels <= h – 2 have two children each, and
  - When a node at level h – 1 has children, all nodes to its left at the same level have two children each, and
  - When a node at level h – 1 has one child, it is a left child

Balanced Binary Trees

- A binary tree is balanced if the heights of any node’s two subtrees differ by no more than 1
- Complete binary trees are balanced
- Full binary trees are complete and balanced

Traversals of a Binary Tree

- A traversal visits each node in a tree
  - to do something with or to the node during a visit
  - for example, display the data in the node

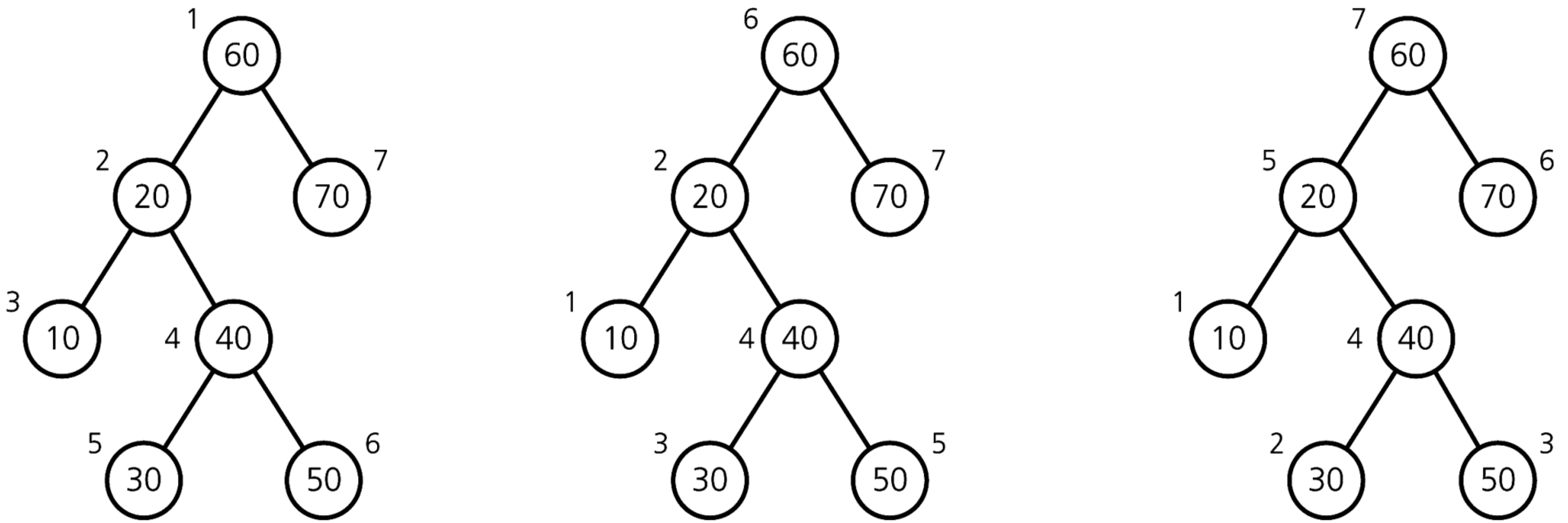
- General form of a recursive traversal algorithm

```
traverse (in binTree:BinaryTree)
   if (binTree is not empty)
   {  traverse(Left subtree of binTree's root)
      traverse(Right subtree of binTree's root)
   }
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before visiting its subtrees
    - i. e. Before the recursive calls
- Inorder traversal
  - Visit root between visiting its subtrees
    - i. e. Between the recursive calls
- Postorder traversal
  - Visit root after visiting its subtrees
    - i. e. After the recursive calls

Traversals of a Binary Tree



(a) Preorder: 60, 20, 10, 40, 30, 50, 70

(b) Inorder: 10, 20, 30, 40, 50, 60, 70

(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

Figure 10-10

Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

Traversals of a Binary Tree

- A traversal operation can call a function to perform a task on each item in the tree
  - this function defines the meaning of "visit"
  - the client defines and passes this function as an argument to the traversal operation
- Tree traversal orders correspond to algebraic expressions
  - infix, prefix, and postfix

The ADT Binary Tree

Binary tree
root left subtree right subtree
createBinaryTree() destroyBinaryTree() isEmpty() getRootData() setRootData() attachLeft() attachRight() attachLeftSubtree() attachRightSubtree() detachLeftSubtree() detachRightSubtree() getLeftSubtree() getRightSubtree() preorderTraverse() inorderTraverse() postorderTraverse()

```
+createBinaryTree()
+createBinaryTree(in rootItem: TreeItemType)
+createBinaryTree(in rootItem: TreeItemType,
                  inout leftTree: BinaryTree,
                  inout rightTree: BinaryTree)
+destroyBinaryTree()
+isEmpty(): boolean {query}
+getRootData(): TreeItemType                          throw TreeException
+setRootData(in newItem: TreeItemType)                throw TreeException
+attachLeft(in newItem: TreeItemType)                 throw TreeException
+attachRight(in newItem: TreeItemType)                throw TreeException
+attachLeftSubtree(inout leftTree: BinaryTree)        throw TreeException
+attachRightSubtree(inout rightTree: BinaryTree)      throw TreeException
+detachLeftSubtree(out leftTree: BinaryTree)          throw TreeException
+detachRightSubtree(out rightTree: BinaryTree)        throw TreeException
+getLeftSubtree(): BinaryTree
+getRightSubtree(): BinaryTree
+preorderTraverse(in visit:FunctionType)
+inorderTraverse(in visit:FunctionType)
+postorderTraverse(in visit:FunctionType)
```

The ADT Binary Tree

- Building the ADT binary tree in Fig. 10-6b

```
tree1.setRootData('F')
tree1.attachLeft('G')
tree2.setRootData('D')
tree2.attachLeftSubtree(tree1)
tree3.setRootData('B')
tree3.attachLeftSubtree(tree2)
tree3.attachRight('E)
tree4.setRootData('C')
tree10_6.createBinaryTree('A',tree3,tree4)
```

Possible Representations of a Binary Tree

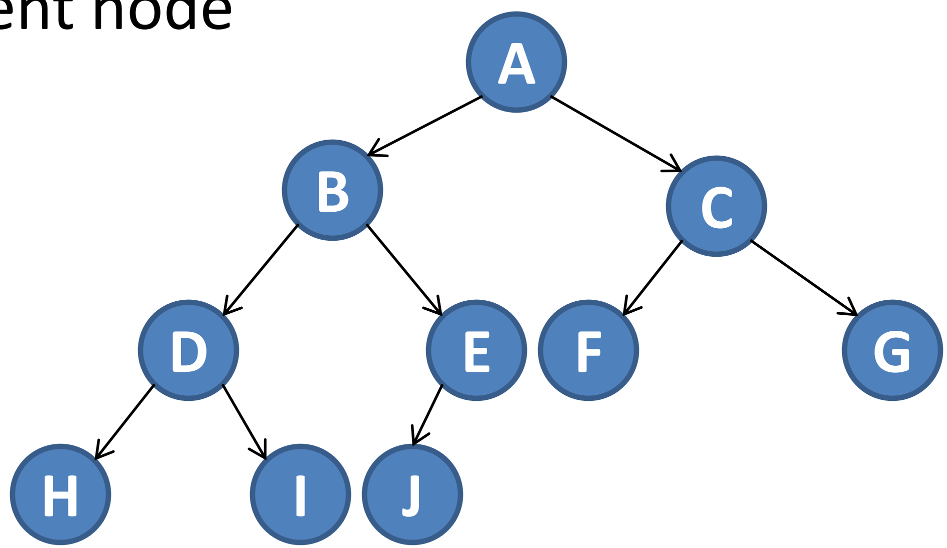
- An array-based representation
  - Uses an array of tree nodes
  - Requires the creation of a free list that keeps track of available nodes
  - only suitable for *complete* binary trees
- A pointer-based representation
  - Nodes have two pointers that link the nodes in the tree

Array Based Binary Tree

- Given a complete binary tree T with n nodes, T can be represented using an array A[0:n-1] such that
  - root of T is in A[0]
  - for node A[i], its left child is at A[2i+1] and its right child at A[2i+2] if it exists
- Completeness of the tree is important because it minimizes the size of the array required
- Note that
  - parent of node A[i] is at A[(i-1)/2]
  - for n > 1, A[i] is a leaf node iff n <= 2i
- Balanced requirement makes an array representation unsuitable for binary search tree implementation

Array Based Binary Tree

- Complete tree fits in minimum size array
  - space efficient
- Nodes do not need child or parent pointers
  - index of these can be calculated from the index of the current node



A	B	C	D	E	F	G	H	I	J		

Array Based Binary Tree

- Advantages
  - space saving through direct computation of child and parent indices rather than pointers
  - O(1) access time through direct computation
    - pointers are also O(1) access but with larger K
- Disadvantages
  - only useful when tree is complete
    - or, complete enough that unused cells do not waste much memory
  - sparse tree representation is too memory intensive
- If a complete tree is of height h, it requires an array of size 2^h-1
  - a skewed BST of 10 nodes is of height 10, requiring an array of size $2^{10}-1 = 1023$

Pointer-based ADT Binary Tree

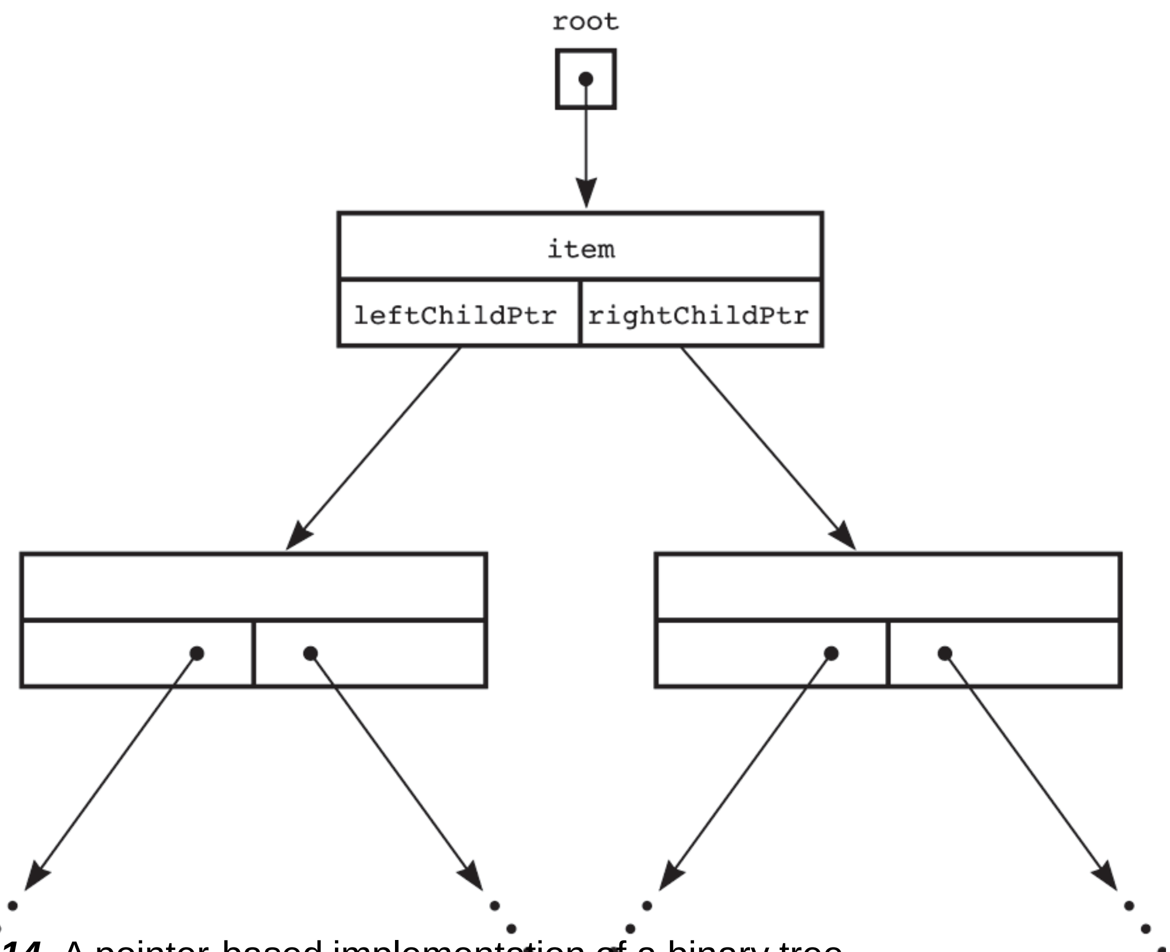


Figure 10-14 A pointer-based implementation of a binary tree

Pointer-based ADT Binary Tree

- `TreeException` and `TreeNode` classes
- `BinaryTree` class
  - Several constructors, including a
    - Protected constructor whose argument is a pointer to a root node; prohibits client access
    - Copy constructor that calls a private function to copy each node during a traversal of the tree
  - Destructor

Binary Tree ADT – TreeNode.h

```
// TreeNode.h

typedef string TreeItemType;
// node in the tree
class TreeNode {
private:
   TreeNode() {};
   TreeNode(const TreeItemType& nodeItem, TreeNode *left = NULL,
                  TreeNode *right = NULL): item(nodeItem),
                                           leftChildPtr(left),
                                           rightChildPtr(right) {}
   TreeItemType item;          // data portion
   TreeNode *leftChildPtr;     // pointer to left child
   TreeNode *rightChildPtr; // pointer to right child
   friend class BinaryTree;  // friend class
};
```

Binary Tree ADT – TreeException.h

```
// TreeException.h

#include <stdexcept>
#include <string>
using namespace std;

Class Tree Exception : public logic_error {
public:
   TreeException(const string& message = “”) :
                          logic_error(message.c_str())
    {}
};
```

Binary Tree ADT – BinaryTree.h

```
//Begin BinaryTree.h

#include "TreeException.h"
#include "TreeNode.h"

// This function pointer is used by the client
// to customize what happens when a node is visited
typedef void (*FunctionType)(TreeItemType& anItem);

class BinaryTree {
public:
    // constructors and destructor:
    BinaryTree();
    BinaryTree(const TreeItemType& rootItem);
    BinaryTree(const TreeItemType& rootItem, BinaryTree& leftTree,
                                              BinaryTree& rightTree);
    BinaryTree(const BinaryTree& tree);
    virtual ~BinaryTree();
```

Binary Tree ADT – BinaryTree.h

```
// binary tree operations:
virtual bool isEmpty() const;

virtual TreeItemType getRootData() const                      throw(TreeException);
virtual void setRootData(const TreeItemType& newItem) throw (TreeException);

virtual void attachLeft(const TreeItemType& newItem)     throw(TreeException);
virtual void attachRight(const TreeItemType& newItem)   throw(TreeException);
virtual void attachLeftSubtree(BinaryTree& leftTree)         throw(TreeException);
virtual void attachRightSubtree(BinaryTree& rightTree)    throw(TreeException);

virtual void detachLeftSubtree(BinaryTree& leftTree)       throw(TreeException);
virtual void detachRightSubtree(BinaryTree& rightTree)   throw(TreeException);

virtual BinaryTree getLeftSubtree() const;
virtual BinaryTree getRightSubtree() const;

virtual void preorderTraverse(FunctionType visit);
virtual void inorderTraverse(FunctionType visit);
virtual void postorderTraverse(FunctionType visit);
```

Binary Tree ADT – BinaryTree.h

```
// overloaded assignment operator:
    virtual BinaryTree& operator=(const BinaryTree& rhs);

protected:
    BinaryTree(TreeNode *nodePtr);                    // constructor

    // Copies the tree rooted at treePtr into a tree rooted
    // at newTreePtr. Throws TreeException if a copy of the
    // tree cannot be allocated.
    void copyTree(TreeNode *treePtr, TreeNode* & newTreePtr) const
                          throw(TreeException);;

    // Deallocate memory for a tree.
   void destroyTree(TreeNode * &treePtr);

    // The next two functions retrieve and set the value
    // of the private data member root.
    TreeNode *rootPtr( ) const;
    void setRootPtr(TreeNode *newRoot);
```

Binary Tree ADT – BinaryTree.h

```
    // The next two functions retrieve and set the values
    // of the left and right child pointers of a tree node.
    void getChildPtrs(TreeNode *nodePtr, TreeNode * &leftChildPtr,
                      TreeNode * &rightChildPtr) const;
    void setChildPtrs(TreeNode *nodePtr, TreeNode *leftChildPtr,
                      TreeNode *rightChildPtr);

    void preorder(TreeNode *treePtr, FunctionType visit);
    void inorder(TreeNode *treePtr, FunctionType visit);
    void postorder(TreeNode *treePtr, FunctionType visit);

private:
    TreeNode *root; // pointer to root of tree

}; // end class
// End of header file. BinaryTree.h
```

Binary Tree ADT – BinaryTree.cpp

```
// Implementation file BinaryTree.cpp for the ADT binary  tree.
#include "BinaryTree.h"             // header file
#include <cstddef>                  // definition of NULL
#include <cassert>                  // for assert()

BinaryTree::BinaryTree() : root(NULL) { }

BinaryTree::BinaryTree(const TreeItemType& rootItem) {
   root = new TreeNode(rootItem, NULL, NULL);
   assert(root != NULL);
}

BinaryTree::BinaryTree(const TreeItemType& rootItem,
                              BinaryTree& leftTree, BinaryTree& rightTree) {
   root = new TreeNode(rootItem, NULL, NULL);
   assert(root != NULL);
   attachLeftSubtree(leftTree);
   attachRightSubtree(rightTree);
}
```

Binary Tree ADT – BinaryTree.cpp

```
BinaryTree::BinaryTree(const BinaryTree& tree) {
    copyTree(tree.root, root);
}

BinaryTree::BinaryTree(TreeNode *nodePtr): root(nodePtr) { }

BinaryTree::~BinaryTree() {
    destroyTree(root);
}

bool BinaryTree::isEmpty() const {
    return (root == NULL);
}

TreeItemType BinaryTree::getRootData() const {
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    return root>item;
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::setRootData(const TreeItemType& newItem) {
    if (!isEmpty()) {
        root>item = newItem;
    } else {
        root = new TreeNode(newItem, NULL, NULL);
        if (root == NULL)
            throw TreeException("TreeException: Cannot allocate memory");
    }
}

void BinaryTree::attachLeft(const TreeItemType& newItem) {
    if (isEmpty()) {
        throw TreeException("TreeException: Empty tree");
    } else if (root>leftChildPtr != NULL) {
        throw TreeException("TreeException: Cannot overwrite left subtree");
    } else {            // Assertion: nonempty tree; no left child
        root>leftChildPtr = new TreeNode(newItem, NULL, NULL);
        if (root>leftChildPtr == NULL)
            throw TreeException("TreeException: Cannot allocate memory");
    }
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::attachRight(const TreeItemType& newItem) {
   if (isEmpty())
      throw TreeException("TreeException: Empty tree");
   else if (root>rightChildPtr != NULL)
      throw TreeException("TreeException: Cannot overwrite right subtree");
   else { // Assertion: nonempty tree; no right child
      root>rightChildPtr = new TreeNode(newItem, NULL, NULL);
      if (root>rightChildPtr == NULL)
         throw TreeException("TreeException: Cannot allocate memory");
   }
}
void BinaryTree::attachLeftSubtree(BinaryTree& leftTree) {
   if (isEmpty())
      throw TreeException("TreeException: Empty tree");
   else if (root>leftChildPtr != NULL)
      throw TreeException("TreeException: Cannot overwrite left subtree");
   else {                    // Assertion: nonempty tree; no left child
      root>leftChildPtr = leftTree.root;
      leftTree.root = NULL;
   }
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::attachRightSubtree(BinaryTree& rightTree) {
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    else if (root>rightChildPtr != NULL)
        throw TreeException("TreeException: Cannot overwrite right subtree");
    else {                  // Assertion: nonempty tree; no right child
        root>rightChildPtr = rightTree.root;
        rightTree.root = NULL;
    }
}

void BinaryTree::detachLeftSubtree(BinaryTree& leftTree) {
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    else {
        leftTree = BinaryTree(root>leftChildPtr); // constructor taking node * not
tree *
        root>leftChildPtr = NULL;
    }
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::detachRightSubtree(BinaryTree& rightTree) {
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    else {
        rightTree = BinaryTree(root>rightChildPtr); // node * to tree
conversion
        root>rightChildPtr = NULL; // this tree no longer holds that subtree
    }
}

BinaryTree BinaryTree::getLeftSubtree() const {
    TreeNode *subTreePtr;
    if (isEmpty())
        return BinaryTree();
    else {
        copyTree(root>leftChildPtr, subTreePtr);
        return BinaryTree(subTreePtr);
    }
}
```

Binary Tree ADT – BinaryTree.cpp

```
BinaryTree BinaryTree::rightSubtree() const {
    TreeNode *subTreePtr;
    if (isEmpty())
        return BinaryTree();
    else {
        copyTree(root>rightChildPtr, subTreePtr);
        return BinaryTree(subTreePtr);
    }
}

void BinaryTree::preorderTraverse(FunctionType visit) {
    preorder(root, visit); // preorder written with respect to a tree ptr
}

void BinaryTree::inorderTraverse(FunctionType visit) {
    inorder(root, visit);
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::postorderTraverse(FunctionType visit) {
    postorder(root, visit);
}
BinaryTree& BinaryTree::operator=(const BinaryTree& rhs) {
    if (this != &rhs) {
        destroyTree(root);             // deallocate lefthand side
        copyTree(rhs.root, root);      // copy righthand side
    }
    return *this;
}
void BinaryTree::destroyTree(TreeNode *& treePtr) {
    if (treePtr != NULL) {
        destroyTree(treePtr>leftChildPtr);
        destroyTree(treePtr>rightChildPtr);
        delete treePtr;                          // postorder traversal
        treePtr = NULL;
    }
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::copyTree(TreeNode *treePtr, TreeNode *& newTreePtr) const
{
    // preorder traversal
    if (treePtr != NULL) {
        // copy node
        newTreePtr = new TreeNode(treePtr>item, NULL, NULL);
        if (newTreePtr == NULL)
            throw TreeException("TreeException: Cannot allocate memory");
        copyTree(treePtr>leftChildPtr, newTreePtr>leftChildPtr);
        copyTree(treePtr>rightChildPtr, newTreePtr>rightChildPtr);
    } else
        newTreePtr = NULL;     // copy empty tree
}

TreeNode *BinaryTree::rootPtr() const {
     return root;
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::setRootPtr(TreeNode *newRoot) {
    root = newRoot;
}

void BinaryTree::getChildPtrs(TreeNode *nodePtr, TreeNode *& leftPtr,
                                            TreeNode *& rightPtr) const {
    leftPtr = nodePtr>leftChildPtr;
    rightPtr = nodePtr>rightChildPtr;
}

void BinaryTree::setChildPtrs(TreeNode *nodePtr, TreeNode *leftPtr,
                                              TreeNode *rightPtr) {
    nodePtr>leftChildPtr = leftPtr;
    nodePtr>rightChildPtr = rightPtr;
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::preorder(TreeNode *treePtr, FunctionType visit) {
    if (treePtr != NULL) {
        visit(treePtr>item);
        preorder(treePtr>leftChildPtr, visit);
        preorder(treePtr>rightChildPtr, visit);
    }
}

void BinaryTree::inorder(TreeNode *treePtr, FunctionType visit) {
    if (treePtr != NULL) {
        inorder(treePtr>leftChildPtr, visit);
        visit(treePtr>item);
        inorder(treePtr>rightChildPtr, visit);
    }
}
```

Binary Tree ADT – BinaryTree.cpp

```
void BinaryTree::postorder(TreeNode *treePtr, FunctionType visit) {
    if (treePtr != NULL) {
        postorder(treePtr>leftChildPtr, visit);
        postorder(treePtr>rightChildPtr, visit);
        visit(treePtr>item);
    }
}
// End of implementation file.
```

Binary Tree ADT – Client Code

```
// Example client code
#include <iostream>
#include "BinaryTree.h"
using namespace std;
void display(TreeItemType& anItem);

int main()
{
   BinaryTree tree1, tree2, left;
   // tree with only a root 70
   BinaryTree tree3(70);

   // build the tree in Figure 10-10
   tree1.setRootData(40);
   tree1.attachLeft(30);
   tree1.attachRight(50);
   tree2.setRootData(20);
   tree2.attachLeft(10);
   tree2.attachRightSubtree(tree1);

   // tree in Fig 10-10
   BinaryTree binTree(60, tree2, tree3);

   binTree.inorderTraverse(display);
   binTree.getLeftSubtree().inorderTraverse
                                     (display);
   binTree.detachLeftSubtree(left);
   left.inorderTraverse(display);
   binTree.inorderTraverse(display);

   return 0;
} // end main
```

Pointer-based ADT Binary Tree: Tree Traversals

- `BinaryTree` class (continued)
  - Public methods for traversals so that visiting a node remains on the client's side of the wall

```
void inorderTraverse(FunctionType visit);
typedef void (*FunctionType)(TreeItemType&
                                    item);
```

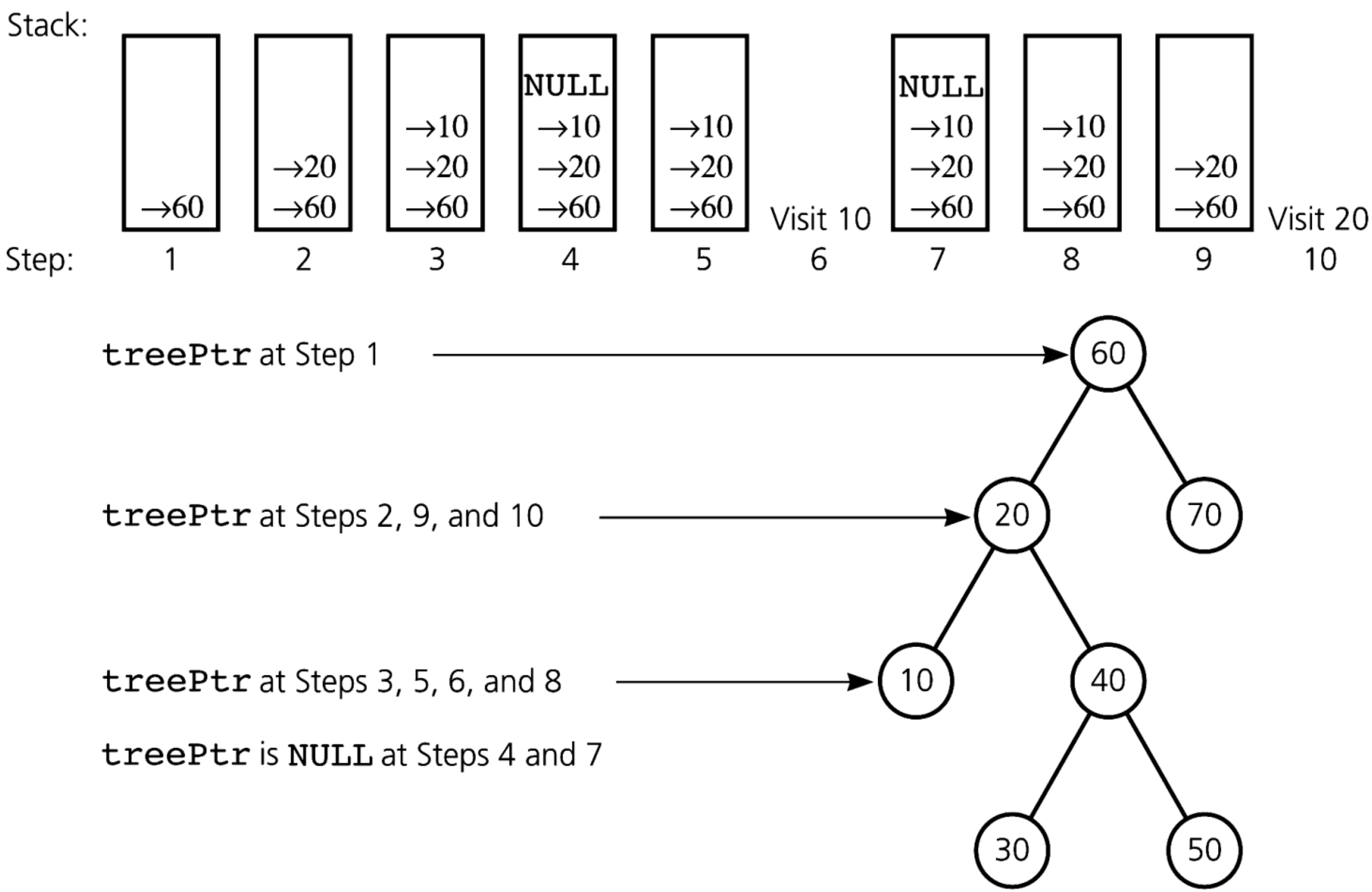
  - Protected methods, such as `inorder`, that enable the recursion

```
void inorder(TreeNode *treeptr,
             FunctionType visit);
```

  - `inorderTraverse` calls `inorder`, passing it a node pointer and the client-defined function `visit`

Recursive Inorder Traversal

(The notation →60 means "a pointer to the node containing 60.")



Nonrecursive Inorder Traversal

- An iterative method and an explicit stack can mimic the actions of a return from a recursive call to `inorder`

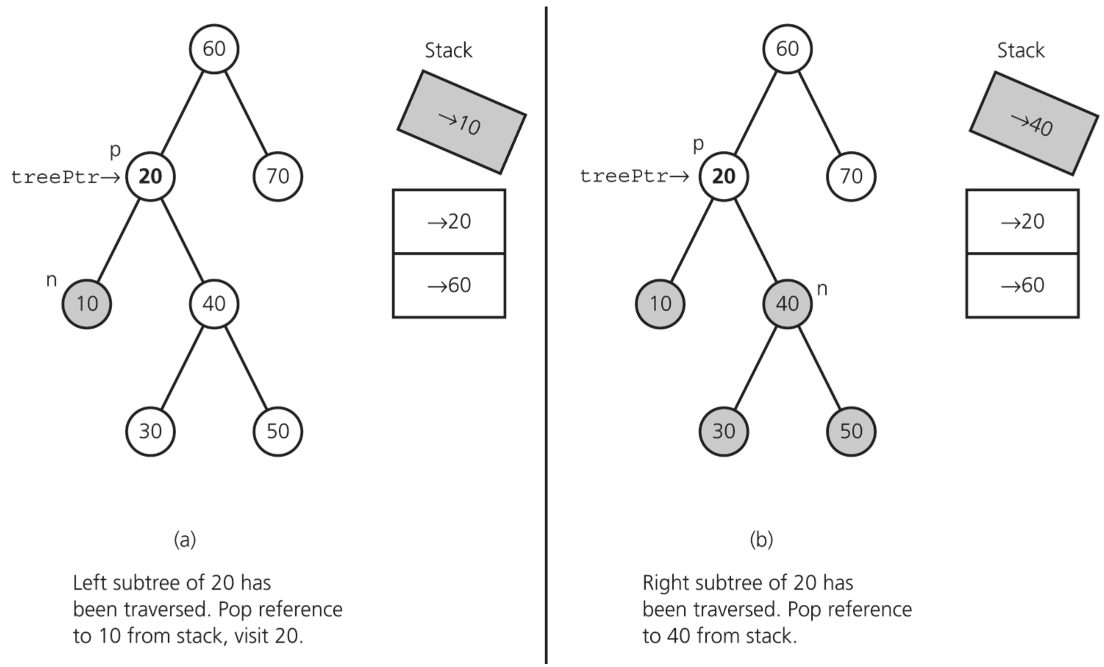


Figure 10-16

Traversing (a) the left and (b) the right subtrees of 20

Copying a Binary Tree

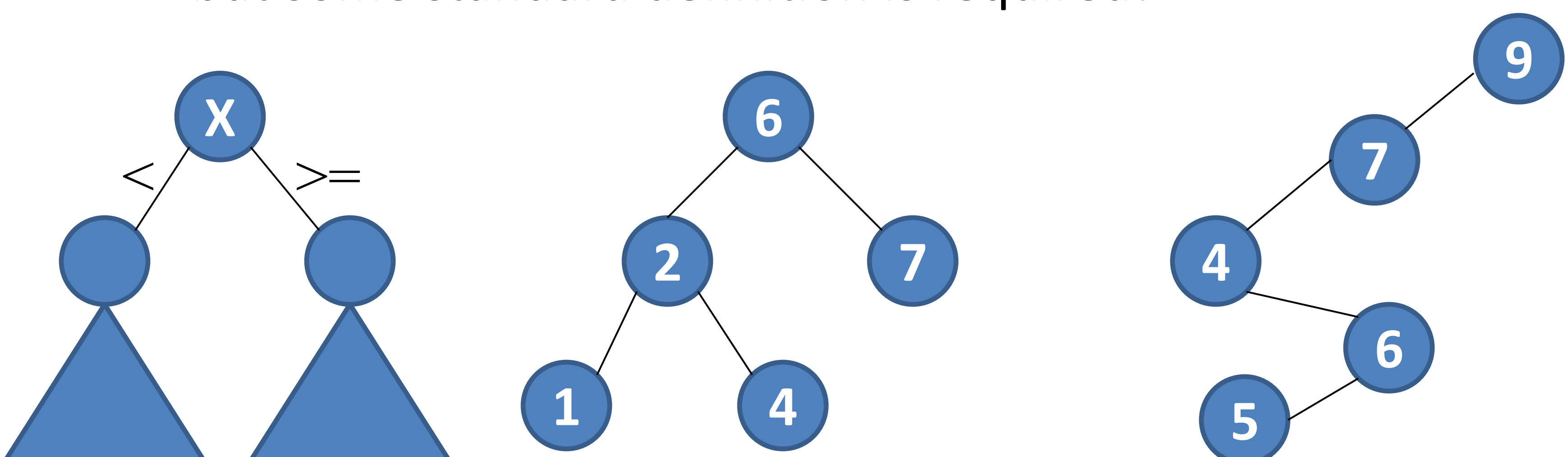
- To copy a tree
  - traverse it in preorder
  - insert each item visited into a new tree
  - use in copy constructor
- To deallocate a tree
  - traverse in postorder
  - delete each node visited
  - “visit” follows deallocation of a node’s subtrees
  - use in destructor

The ADT Binary Search Tree

- The ADT binary tree is not suitable when you need to search for a particular item
  - binary *search* tree (BST) is more suitable
- A data item in a BST has specially designated search key
  - search key is the part of a record that identifies it within a collection of records
- Assume that the set of all keys can be linearly ordered
  - a comparison function for two keys cmp(k2, k2) distinguishes 3 cases: (1) k1 < k2, (2) k1 == k2, or (3) k1 > k2
- If we use a binary search tree to organize the set of records, then each record must be a node in the tree
  - Record is a class instance held by tree node
  - Record field is a member variable
  - Key is the record field used as search tag

Binary Search Trees

- Binary tree H such that key of any node x, *key(x)*, is greater than the keys of all nodes in its *left* subtree and is less than or equal to keys of all nodes in its *right* sub-tree
  - often called the BST property
- Equal elements could as easily be in the left subtree
  - but some standard definition is required!



Binary Search Trees – Observations

- BST may not be a *balanced* binary tree
  - choice of root node is important with respect to the set of all key values present in the tree
- Leftmost descendant of root = minimum item
- Rightmost descendant of root = maximum item
- Inorder traversal of BST = sorted key order
- BST strongly analogous to binary search of an array in sorted order
- Pointer based implementation dynamically allocating tree nodes is the most obvious approach
  - nodes are wrappers for records, might point to records
  - BST **template** would use record **type** as parameter

The ADT Binary Search Tree

- Simple BST API
  - similar to 10-18 in book
- Assumes method RecordT.get_key( ) exists for all possible record types
- Logic of BST find( ) closely resembles binary search in an array
- Logic of insertion is essentially search for the right place for the inserted record in the tree

```
class BST {
public:
    BST();
    ~BST();
    boolean is_empty();
    boolean insert( RecordT& r);
    RecordT* find( KeyT key);
    boolean delete(KeyT key);
    void preorder();
    void inorder();
     void postorder();
private:
    BST_Node *lchild;
    BST_Node *rchild;
    RecordT *record;
};
```

ADT Binary Search Tree – find

- find the record with search key *skey*
- first checks the current node and then recursively searches the relevant subtree if it exists
- If relevant subtree does not exist, the search has fails

```
RecordT * BST::find(const KeyT& skey) {
    if ( record == NULL ) {
        return(NULL);
    } else if ( record>get_key() == skey ) {
        return(record);  // key found
    } else if ( record>get_key() > skey ) {
        // search left tree
        if ( lchild == NULL ) {
            return(NULL);
        }
        return(lchild>find(skey));
    } else {
        // search right tree
        if ( rchild == NULL ) {
            return(NULL);
        }
        return(rchild>find(skey));
    }
}
```

ADT Binary Search Tree: Insertion

- BST::insert() method looks for proper place and adds the record in the right spot
  - insert 7, 3, 1, 8, 13 15, 6, 9, 10 using this algorithm

```
boolean BST::insert(constRecordT& inr) {
    if ( record == NULL ) {
        // This will be the first record in empty tree
        record = &inr;
        return True;
    } else if ( inr>get_key() < record>get_key() ) {
        if ( lchild == NULL )  lchild = new BST;
            return(lchild>insert(inr));
    } else {
        if ( rchild == NULL )   rchild = new BST;
             return(rchild>insert(inr));
     }
}
```

ADT Binary Search Tree: Insertion

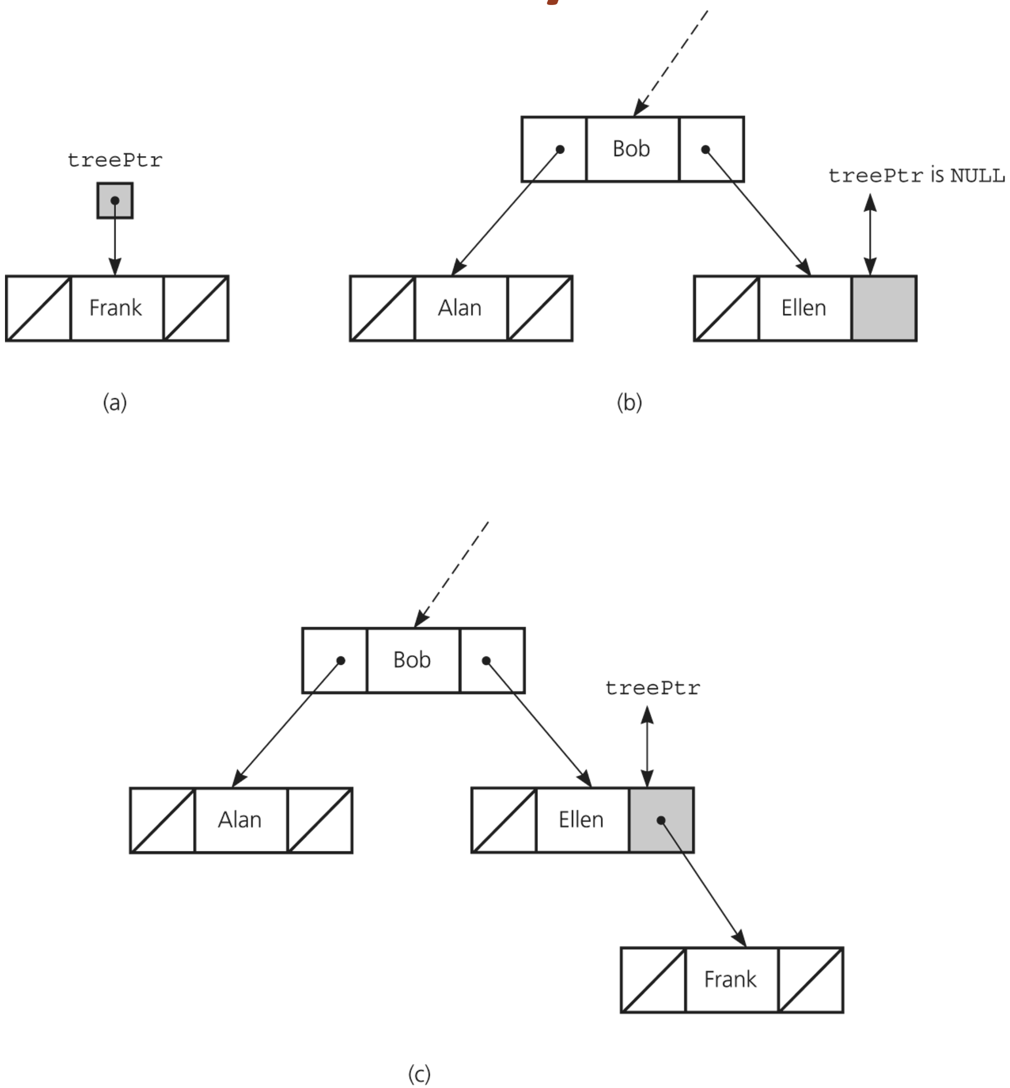


Figure 10-23

(a) Insertion into an empty tree;

(b) search terminates at a leaf;

(c) insertion at a leaf

ADT Binary Search Tree – Delete

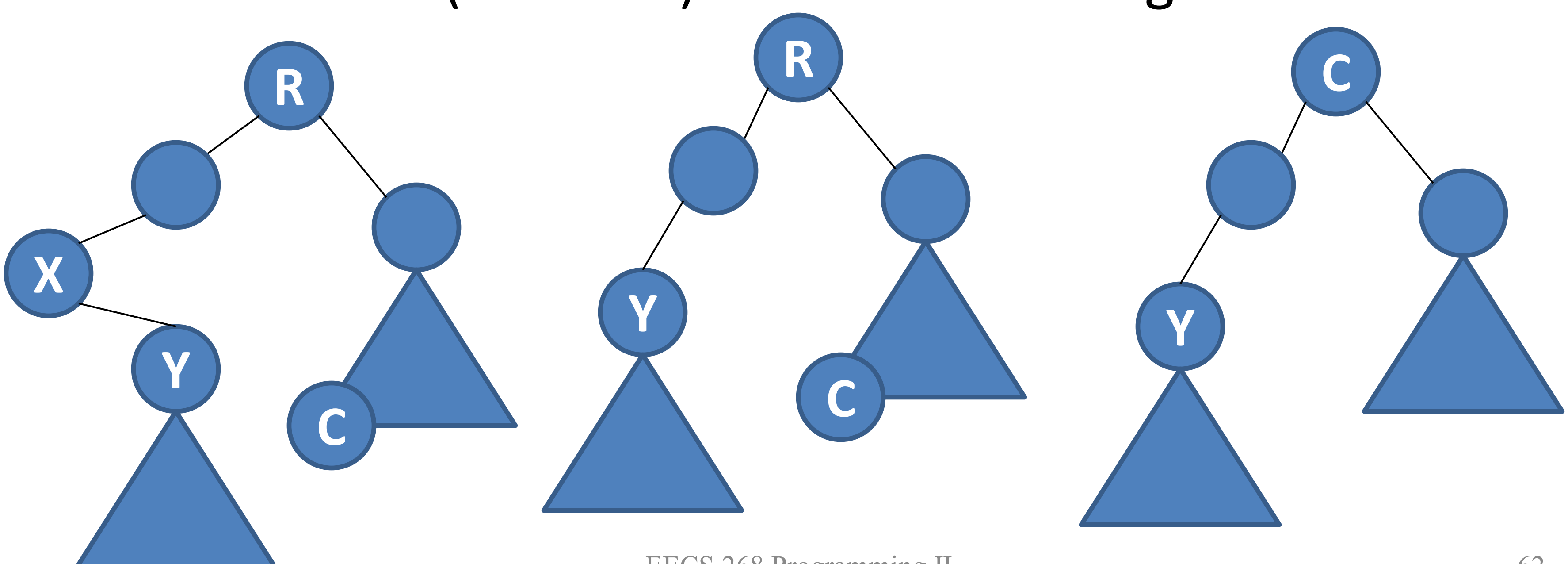
- Delete operation on node N is a bit more complicated
- If N is a leaf
  - both lchild and rchild are NULL
  - parent node pointer referring to N should be set to NULL
    - need a pointer to parent node to do this
- If N has only 1 child
  - replace N with its only child
- If N has two children
  - replace N with minimum item of its right subtree

ADT Binary Search Tree: Delete

- Deleting the item in node N when N has two children (continued)
  - locate another node M that is easier to delete
    - M is the leftmost node in N's right subtree
    - M will have no more than one child
    - M's search key is called the inorder successor of N's search key
  - copy the item that is in M to N
  - remove the node M from the tree

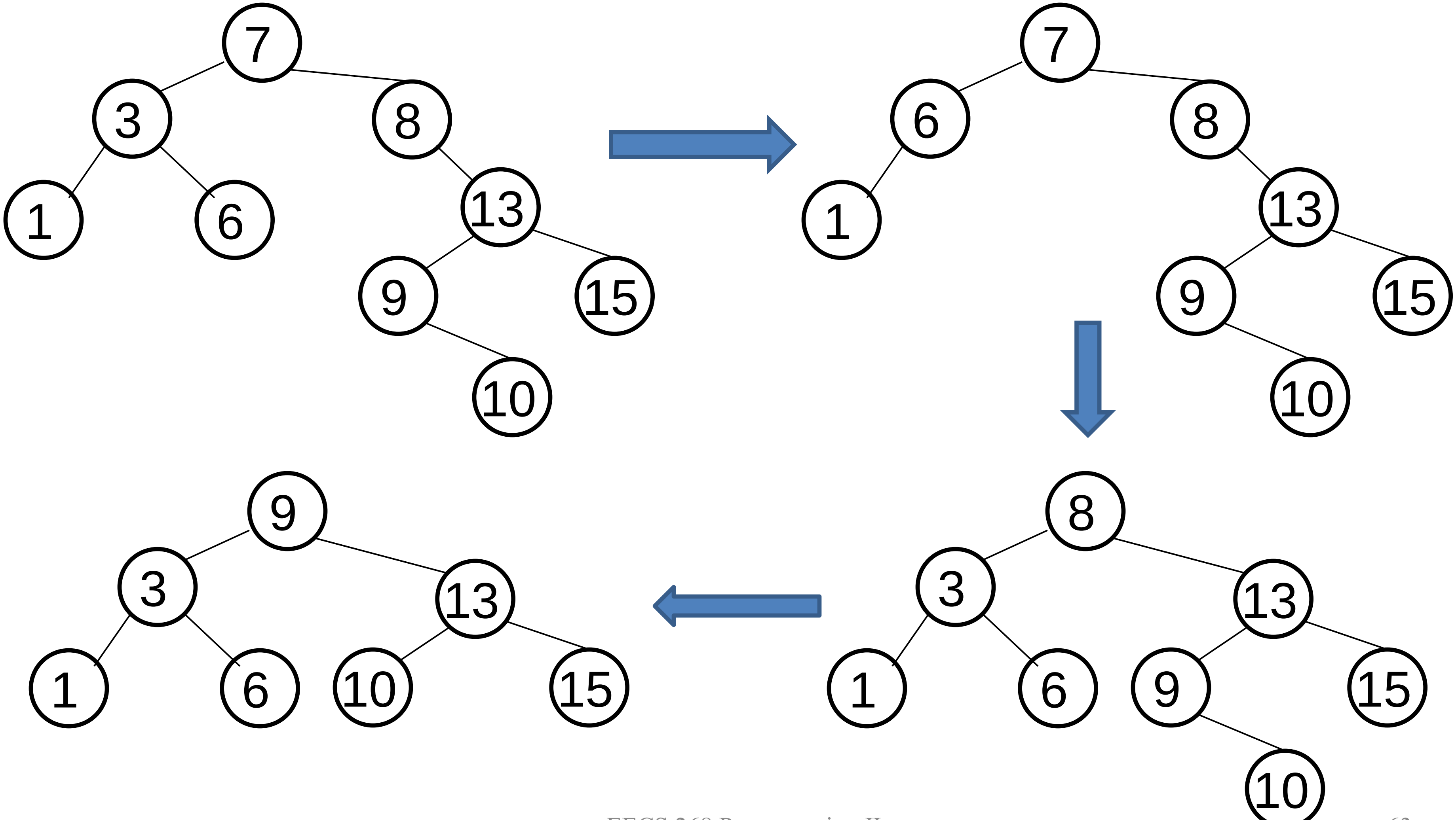
ADT Binary Search Tree: Delete

- Deleting node x is simple because it has only one child and can be replaced by the root of its child without violating any of the BST constraints
- Deleting R is harder, but c can replace it because it is the smallest (leftmost) element of the right sub-tree



ADT Binary Search Tree: Delete

- Delete 3, 7, 8 in order



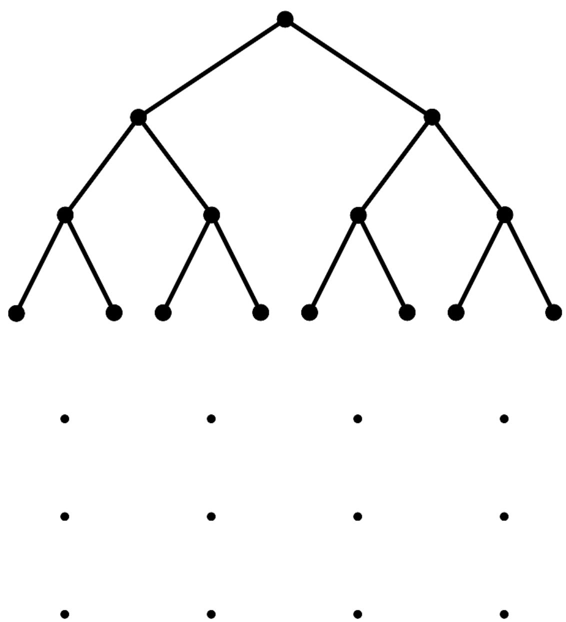
ADT Binary Search Tree: Retrieval and Traversal

- The retrieval operation can be implemented by refining the search algorithm
  - return the item with the desired search key if it exists
  - otherwise, throw TreeException
- Traversals for a binary search tree are the same as the traversals for a binary tree
- Theorem 10-1
  - the inorder traversal of a binary search tree T will visit its nodes in sorted search-key order

Height of a Binary Tree

- Theorem 10-2
  - A full binary tree of height $h \geq 0$ has $2h - 1$ nodes
- Theorem 10-3
  - The maximum number of nodes that a binary tree of height h can have is $2h - 1$
- Theorem 10-4
  - The minimum height of a binary tree with n nodes is $\lceil \log2(n+1) \rceil$
  - Complete trees and full trees have minimum height
- The maximum height of a binary tree with n nodes is n

Height of a Binary Tree



Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
⋮	⋮	⋮
h	2^{h-1}	$2^h - 1$

Figure 10-32 Counting the nodes in a full binary tree of height *h*

The Efficiency of Binary Search Tree Operations

- The maximum number of comparisons required by any b. s. t. operation is the number of nodes along the longest path from root to a leaf—that is, the tree's height
- The order in which insertion and deletion operations are performed on a binary search tree affects its height
- Insertion in random order produces a binary search tree that has near-minimum height

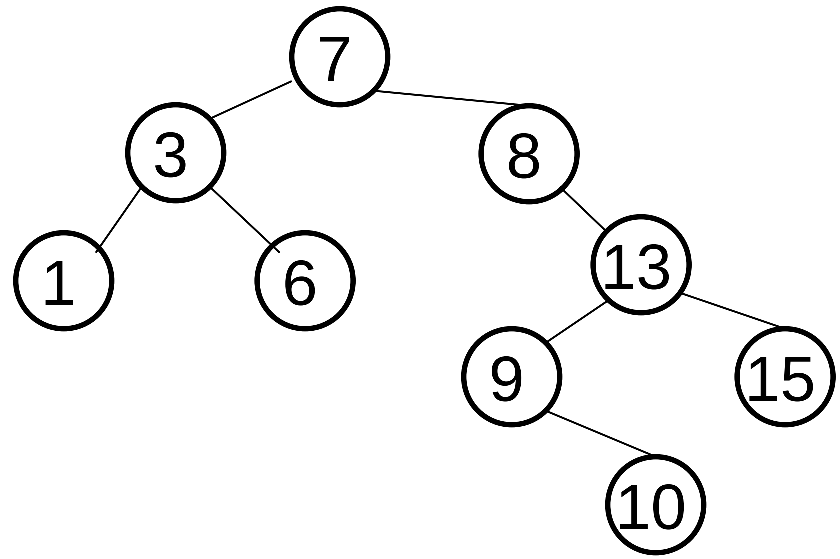
The Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

Figure 10-34 The order of the retrieval, insertion, deletion, and traversal operations for the pointer-based implementation of the ADT binary search tree

Saving and Restoring a BST

- Saving/restoring any data structure to/from a file requires us to serialize the data structure
- files store data linearly
- arrays and linked lists are linear



- Preorder, postorder and inorder traversals produce a linear tree listings
  - what order makes restoration easiest?
- Preorder: 7,3,1,6,8,13,9,10,15
- Insert nodes in an empty BST in this order and it reproduces the original

Applications

- Treesort
  - Uses the ADT binary search tree to sort an array of records into search-key order
    - Average case: O(n * log n)
    - Worst case: O(n2)

n-ary Trees

- An *n*-ary tree is a general tree whose nodes can have no more than *n* children each
  - a generalization of a binary tree

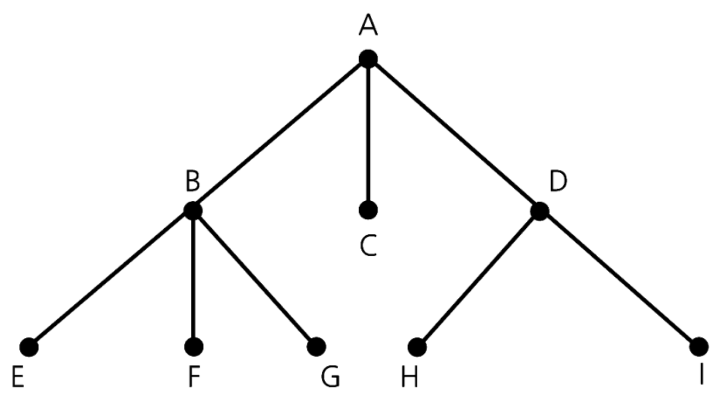


Figure 10-38 A general tree

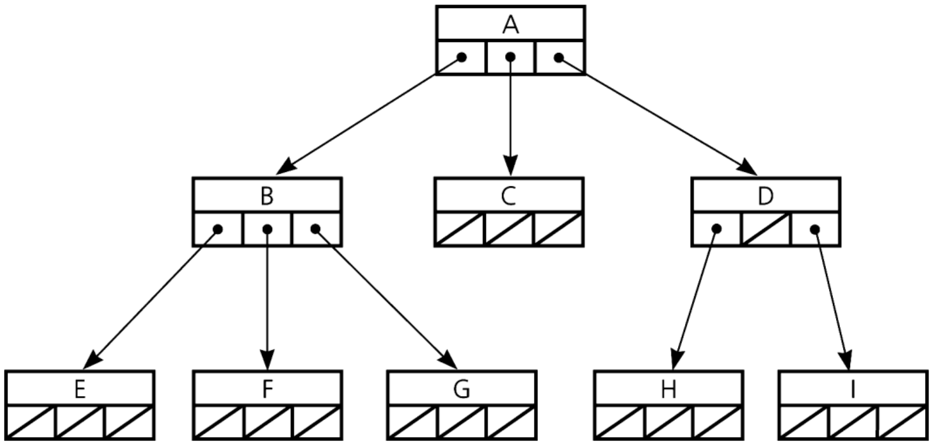


Figure 10-41

An implementation of the *n*-ary tree in Figure 10-38

n-ary Trees

- A binary tree can represent an *n*-ary tree
  - seems a bit odd, but good when the number of children is highly variable and especially when there is no upper bound on the number of children
- Lchild is used to point to the first of its children
  - Rchild pointers are used to link siblings together

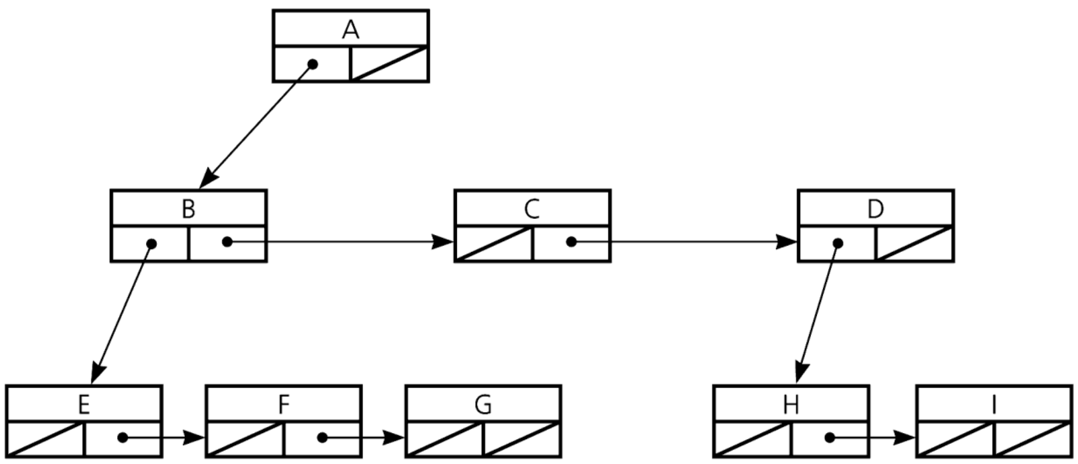


Figure 10-39 Another implementation of the tree in Figure 10-38

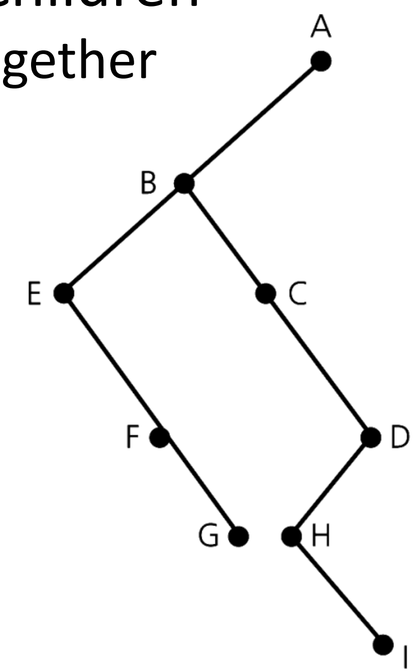


Figure 10-40 The binary tree that Figure 10-39 represents

Summary

- Binary trees provide a hierarchical organization of data
- The implementation of a binary tree is usually pointer-based
- If the binary tree is complete, an efficient array-based implementation is possible
- Traversing a tree to “visit”—that is, do something to or with—each node is useful
- You pass a client-defined “visit” function to the traversal operation to customize its effect on the items in the tree

Summary

- The binary search tree allows you to use a binary search-like algorithm to search for an item having a specified value
- Binary search trees come in many shapes
  - The height of a binary search tree with n nodes can range from a minimum of $\lceil \log_2(n + 1) \rceil$ to a maximum of n
  - The shape of a binary search tree determines the efficiency of its operations

Summary

- An inorder traversal of a binary search tree visits the tree's nodes in sorted search-key order
- The treesort algorithm efficiently sorts an array by using the binary search tree's insertion and traversal operations

Summary

- Saving a binary search tree to a file while performing
  - An inorder traversal enables you to restore the tree as a binary search tree of minimum height
  - A preorder traversal enables you to restore the tree to its original form