Binary Search Trees (BST)

After studying this chapter, you should be able to

- Define and use the following terminology:
  - Binary tree
  - Binary search tree
  - Root
  - Parent
  - Child
  - Ancestor
  - Descendant
  - Level
  - Height
  - Subtree
- Define a binary search tree at the logical level
- Show what a binary search tree would look like after a series of insertions and deletions
- Implement the following binary search tree algorithms in C++:
  - Putting an element in the tree,
  - Deleting an element from the tree,
  - Getting an element from the tree,
  - Modifying an element in the tree,
  - Copying a tree,
  - Traversing a tree in preorder, inorder, and postorder
- Discuss the Big-O efficiency of a given binary search tree operation
- Describe an algorithm for balancing a binary search tree

Searching

- It is often necessary to access an element in a structure that matches a specific condition
  - Not just direct access (e.g., **`array[4]`**)
  - For example, find student in list with ID number 203
- Each structure must define its own methods for searching for elements

Search Specification

FindItem (item, location) (a member function)

- Function: Determine if an element of the list has a key that matches item's
- Precondition: The list has been initialized, item's key has been initialized
- Postcondition: Location = the position of the matching element, or NULL if no such element exists
  - "Position" could mean an array index or a pointer into a linked list

Linear Search

- Simple algorithm: Walk the list, checking each item to see if it matches
- Must be used if the list isn't sorted by key
- O(N) search, performs on average $N/2$ comparisons assuming an equal probability of searching for any element

High Probability Ordering

- Sometimes, a few list elements are in greater demand than others
- It would be very useful to optimize the ordering of elements to make search more efficient in this case
- Search would still be O(*N*) in the worst case, but the *average* case approaches O(1)
- This is called a *self-adjusting list*

High Probability Ordering (cont.)

- How should elements be reordered?
- Naive approach: Every time an element is searched for, move it to the front of the list
  - Inefficient for arrays; don't need to move an element to the front if it's only searched for once
- Better approach: Every time an element is searched for, swap it with its predecessor
  - In-demand elements bubble to the front

Key Ordering

- Sorted lists allow the use of more efficient search algorithms
- Linear search can stop as soon as it passes the position were the element should have been
- Inserting all elements in sorted order is O(N^2), but inserting then sorting is O($N \log_2 N$)
- With sorted lists, binary search may be used

Binary Search

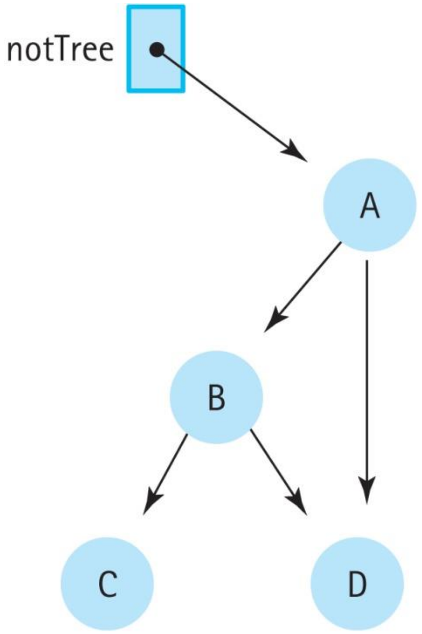
- Discussed previously in Chapters 4 and 7
- Uses divide-and-conquer approach to perform $O(\log_2 N)$ search
- Requires an array-based list that is already sorted
- Could binary search be used by linked lists?

Trees

- Linked lists are linear structures: each node has a unique successor (except for the last node)
- **Binary Trees:** A structure with a unique starting node (the **root**), in which every node has at most two *children* and there is a unique path from the root to every node
- Nodes without children are called **leaf nodes**

Unique Paths

This structure is not a tree:



There are two paths from the root (A) to the node D. Each node must have at most one parent.

A Recursive Structure

- The left and right children of a node are the roots of the left and right *subtrees* of that node
- That is, a binary tree can be recursively split into smaller binary trees
- This is a useful property that makes writing recursive routines on binary trees easy

Binary Tree

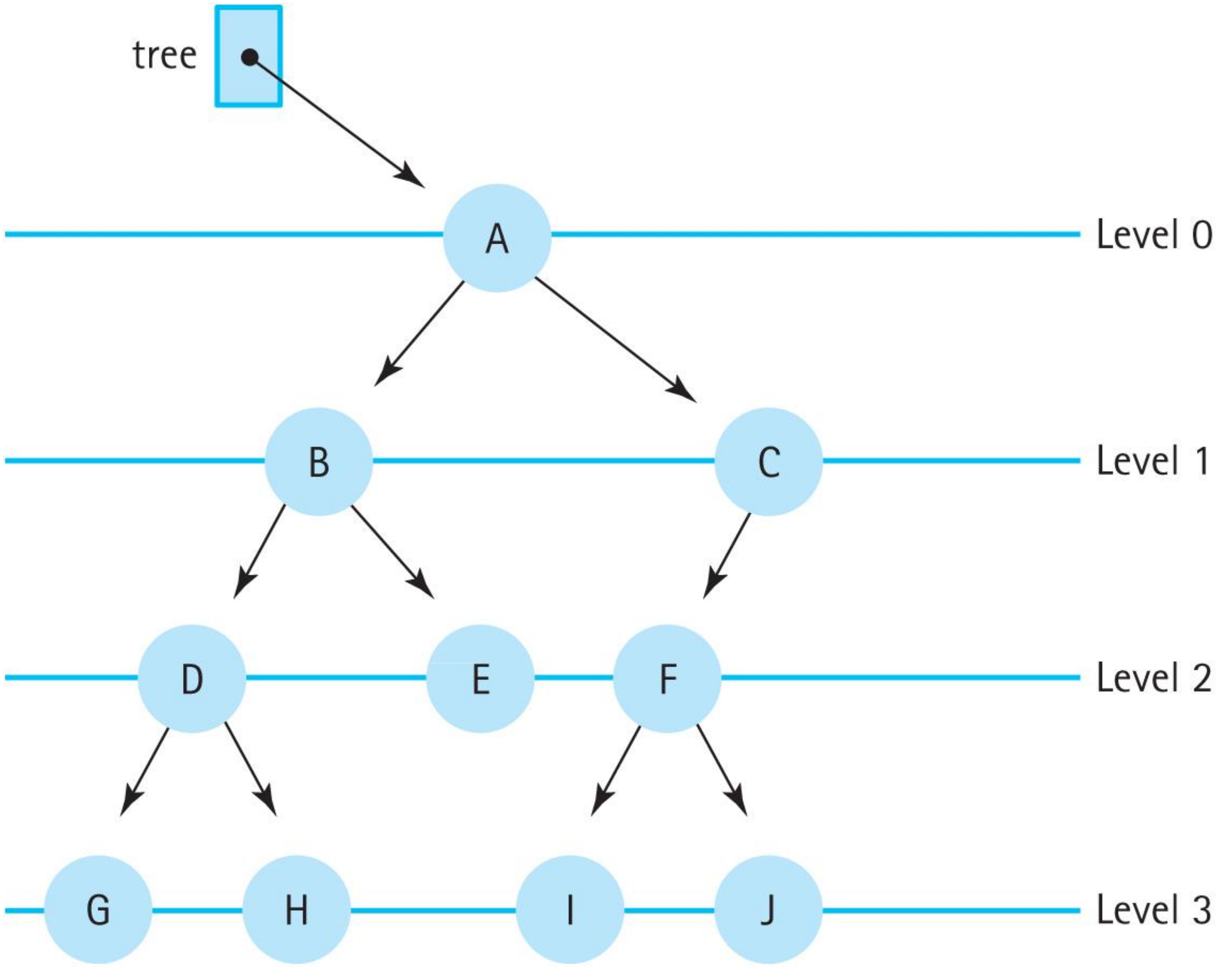


Figure 8.1 A binary tree

Level and Height

- **Level:** The distance of a node to the root
  - The number of nodes at level N is at most 2^N
- **Height:** The maximum level of the tree
  - With N nodes, the max height is N and the minimum height is $\log_2 N + 1$
- Height determines efficiency; a minimum height tree supports $O(\log_2 N)$ access of every element, but a max height tree is $O(N)$

Level and Height (cont.)

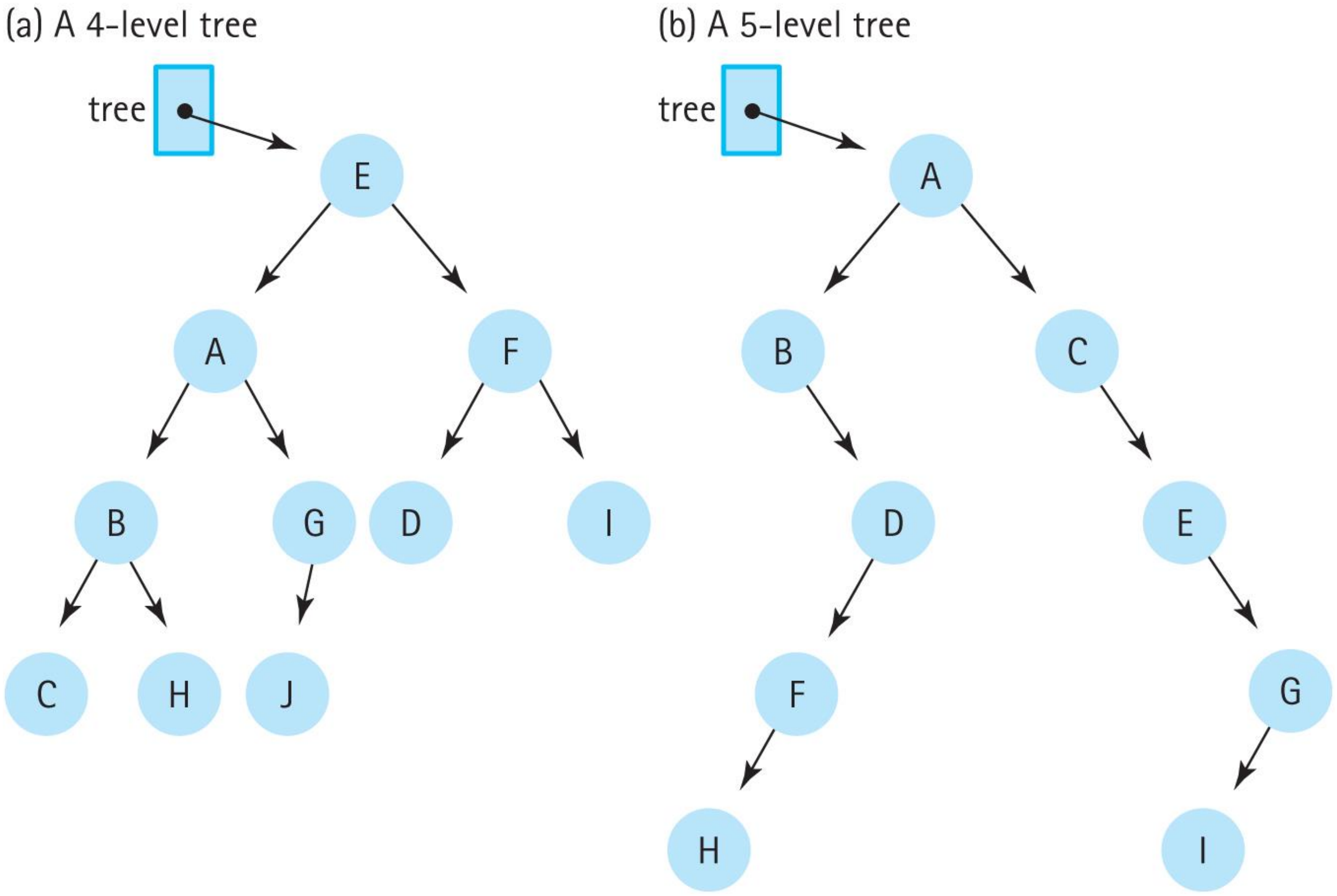


Figure 8.2 Binary trees with ten nodes (a) A 4-level tree (b) A 5-level tree

Binary Search Trees

- If elements are unordered, searching binary trees is still O(*N*)
- **Binary Search Property:** The left subtree of a node contains only values less than the node, and the right subtree contains only values greater than the node
- A tree with this property is called a **Binary Search Tree** (BST)

Binary Search Tree

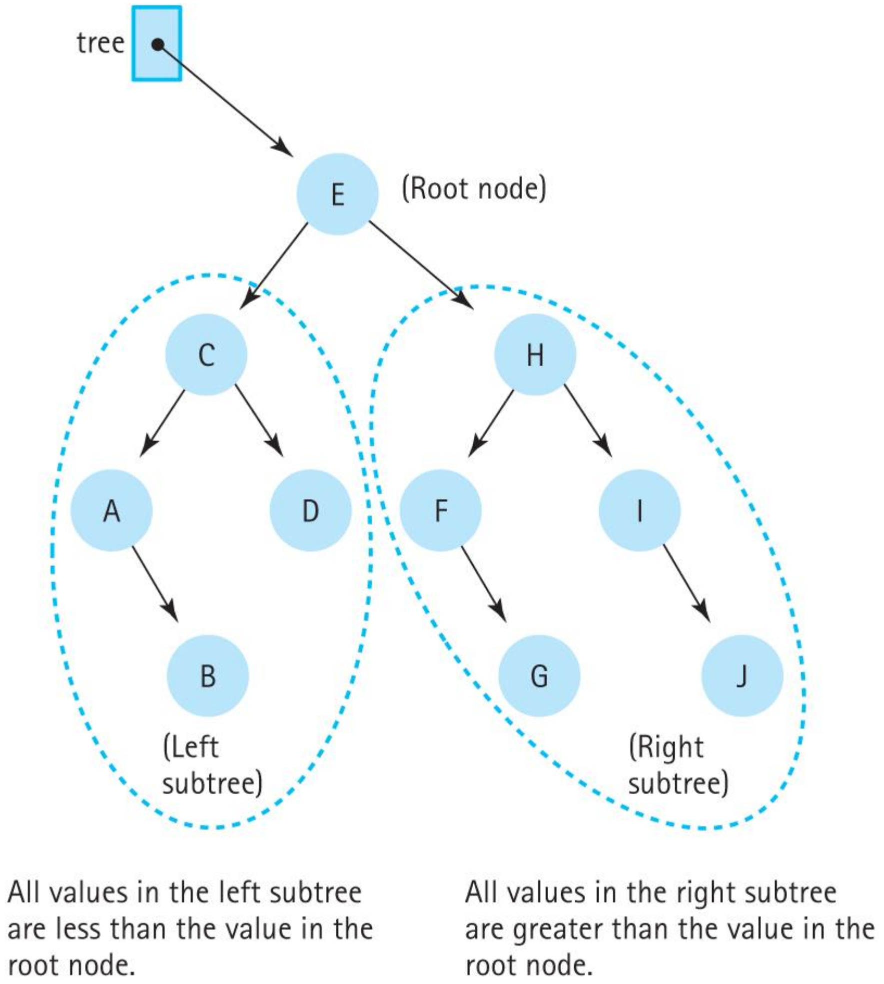


Figure 8.3 A binary search tree

BST: Logical Level

- Inserting and deleting (PutItem, DeleteItem)
- Observers: IsEmpty, IsFull, GetLength, GetItem
- Iterators: ResetTree and GetNextItem
  - Unlike lists, there is more than one way to iterate through a tree
  - These are called *tree traversals*
- Utility: MakeEmpty and Print

BST: Application Level

- Generally, binary search trees can be used in place of lists
- Like lists, they can be used to implement other data structures
- Binary search trees provide O(log*N*) insertion, O(log*N*) deletion, and O(log*N*) search (assuming the tree is close to min. height)

BST: Implementation Level

- BSTs are linked structures with dynamically allocated nodes
- Each node has a pointer to each child
- Since BSTs are inherently recursive (each node is the root of a tree), the algorithms will be implemented recursively

Recursive BST Operations

- The TreeType class contains a pointer to the root of the tree itself
- The recursive operations will operate on nodes
- Therefore, the member functions call recursive helper functions that take the root of the tree as parameters

IsFull and IsEmpty

- IsFull: BSTs are logically unbounded; IsFull attempts to allocate memory and returns “true” if a bad_alloc exception is thrown
- IsEmpty checks if the root of the tree is NULL
- These observers are identical to the linked list methods

GetLength

Number of nodes in tree = Number of nodes in left subtree + Number of nodes in right subtree + 1

- What are the base cases?
  - Left child is NULL
  - Right child is NULL
  - Both children are NULL
- Do we need a branch for each case?

GetLength (cont.)

- Simplify: If the tree is NULL, return 0
- Otherwise, return the count of the two child trees + 1
- This elegantly handles the base cases; if the children are NULL, the function will return 0

GetLength and CountNodes

```
int CountNodes(TreeNode* tree);
int TreeType::GetLength() const
// Calls the recursive function CountNodes to count the
// nodes in the tree.
{
    return CountNodes(root);
}
int CountNodes(TreeNode* tree)
// Post: Returns the number of nodes in the tree.
{
    if (tree == NULL)
        return 0;
    else
        return CountNodes(tree->left) +
          CountNodes(tree->right) + 1;
}
```

GetItem

- Because GetItem searches the tree, it can be programmed recursively
- As before, GetItem takes references to the item to search for and a Boolean flag and updates these parameters as necessary
  - If the item is found, the item reference is updated and the flag is set to true
  - If not, the flag is set to false

GetItem Specification

- Definition: Searches for an element with the matching key; if it is found, return the item
- Size: Number of nodes in the tree or number of nodes in the path from the root
- Base Cases: If the item is found, the pointers are set; if not, only the flag is set to false
- General Cases: Search the appropriate subtree if the key is less than or greater than the node

GetItem and Retrieve

```
ItemType TreeType::GetItem(ItemType item, bool& found) const
{
  Retrieve(root, item, found);
  return item;
}
void Retrieve(TreeNode* tree, ItemType& item, bool& found)
{
  if (tree == NULL)
    found = false; // item is not found.
  else if (item < tree→info)
    Retrieve(tree->left, item, found); // Search left subtree.
  else if (item > tree→info)
    Retrieve(tree->right, item, found); // Search right subtree.
  else
  {
    item = tree->info; // item is found.
    found = true;
  }
}
```

PutItem

- Conceptually similar to inserting into a sorted linked list
- PutItem must maintain the binary search property
- PutItem calls the recursive Insert helper method
- Insert takes a reference to a pointer so that it can update the pointer and the node

Insert

- Definition: Inserts an item into the tree
- Size: The number of elements in the path from the root to the point of insertion
- Base Case: If the tree is NULL, allocate a new node
- General Case: Insert into the left or right subtree as appropriate

Insert(13)

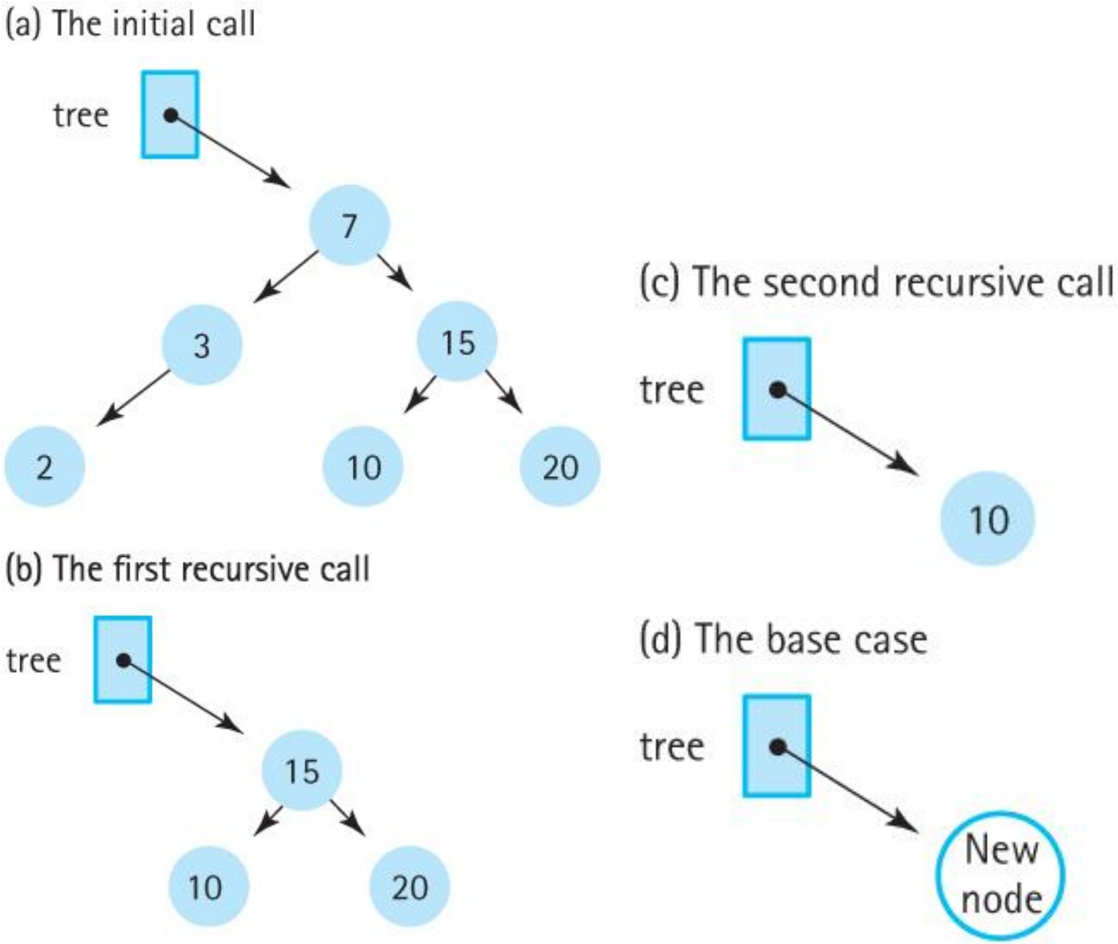


Figure 8.8 The recursive Insert operation (a) The initial call (b) The first recursive call (c) The second recursive call (d) The base case

Building Up a BST

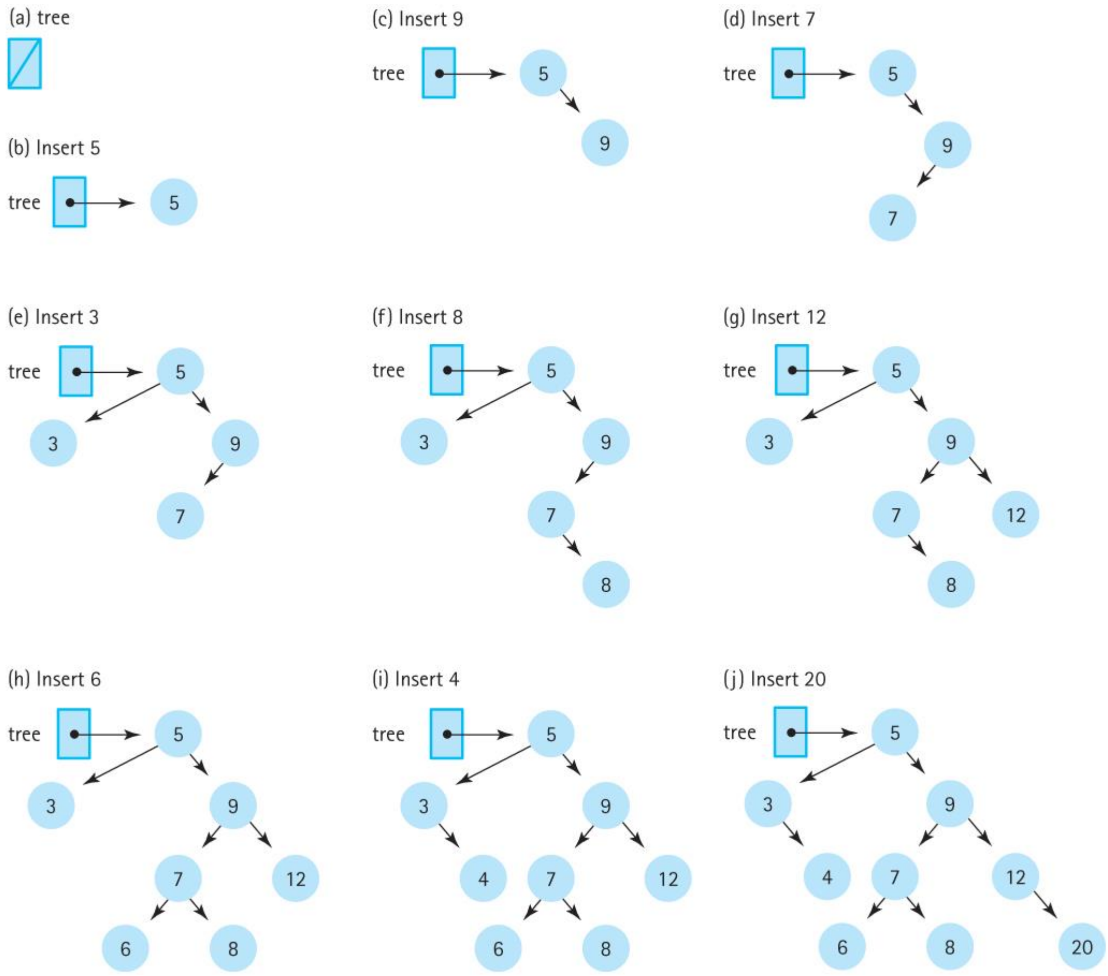


Figure 8.7 Insertions into a binary search tree (a) (b) Insert 5 (c) Insert 9 (d) Insert 7 (e) Insert 3 (f) Insert 8 (g) Insert 12 (h) Insert 6 (i) Insert 4 (j) Insert 20

PutItem and Insert

```
void Insert(TreeNode*& tree, ItemType item);
void TreeType::PutItem(ItemType item)
{
  Insert(root, item);
}
void Insert(TreeNode*& tree, ItemType item)
{
  if (tree == NULL)
  {// Insertion place found.
    tree = new TreeNode;
    tree->right = NULL;
    tree->left = NULL;
    tree->info = item;
  }
  else if (item < tree->info)
    Insert(tree->left, item); // Insert in left subtree.
  else
    Insert(tree->right, item); // Insert in right subtree.
}
```

Insertion Order and Tree Shape

- The order nodes are inserted determines the shape of the resulting tree
- Inserting nodes in order will result in a long, list-shaped tree
- Inserting nodes in **random** order results in a "bushy" and therefore more efficient tree
- It is possible to rearrange a tree after insertion

Insertion Order and Tree Shape (cont.)

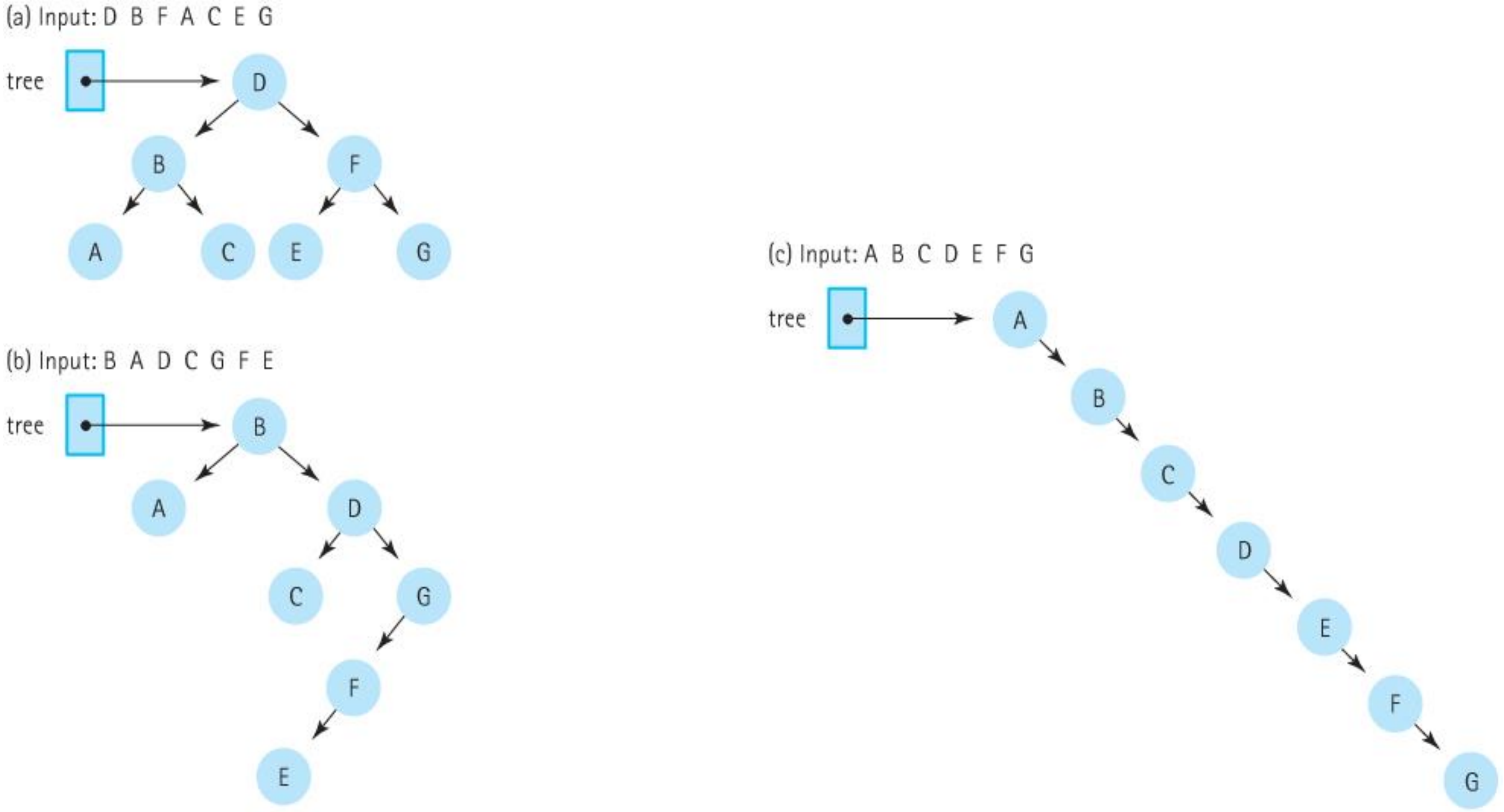


Figure 8.10 The input order determines the shape of the tree (a) Input: D B F A C E G (b) Input: B A D C G F E (c) Input: A B C D E F G

DeleteItem

- Delete (the recursive helper) searches the tree for the matching node and removes it
- Deletion has 3 cases:
  - Deleting a leaf: Set the link in its parent to NULL
  - Deleting a node with one child: Set the link in its parent to point to its child
  - Deleting a node with two children: Complex Updating the parent isn't enough!

Deleting a Node with Two Children

- The binary search property must hold
- The solution is to replace the deleted node with another node
- Can we replace the deleted node with one of its children?
  - No. What would happen to the children’s children?
- We need a node that’s greater than the left child but less than the right child

Logical Predecessor

- We’re looking for the *predecessor* of the deleted node: The node with next lowest value
- The predecessor is the furthest right child in the deleted node’s left subtree
- The predecessor may have a left child, so call Delete (left subtree, predecessor)
- Then replace the deleted node with predecessor

Delete

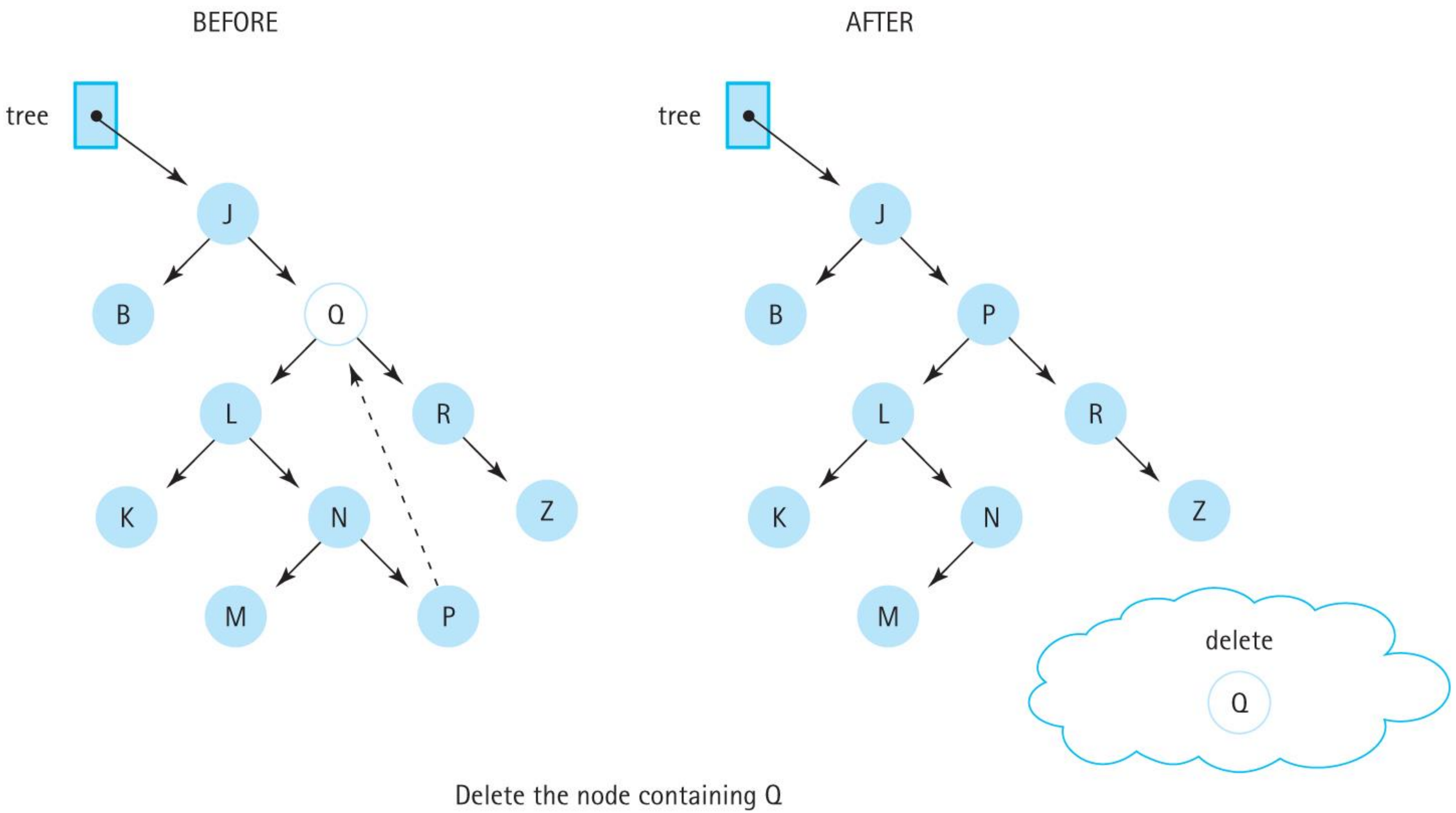


Figure 8.13 Deleting a node with two children

Deleting Multiple Elements

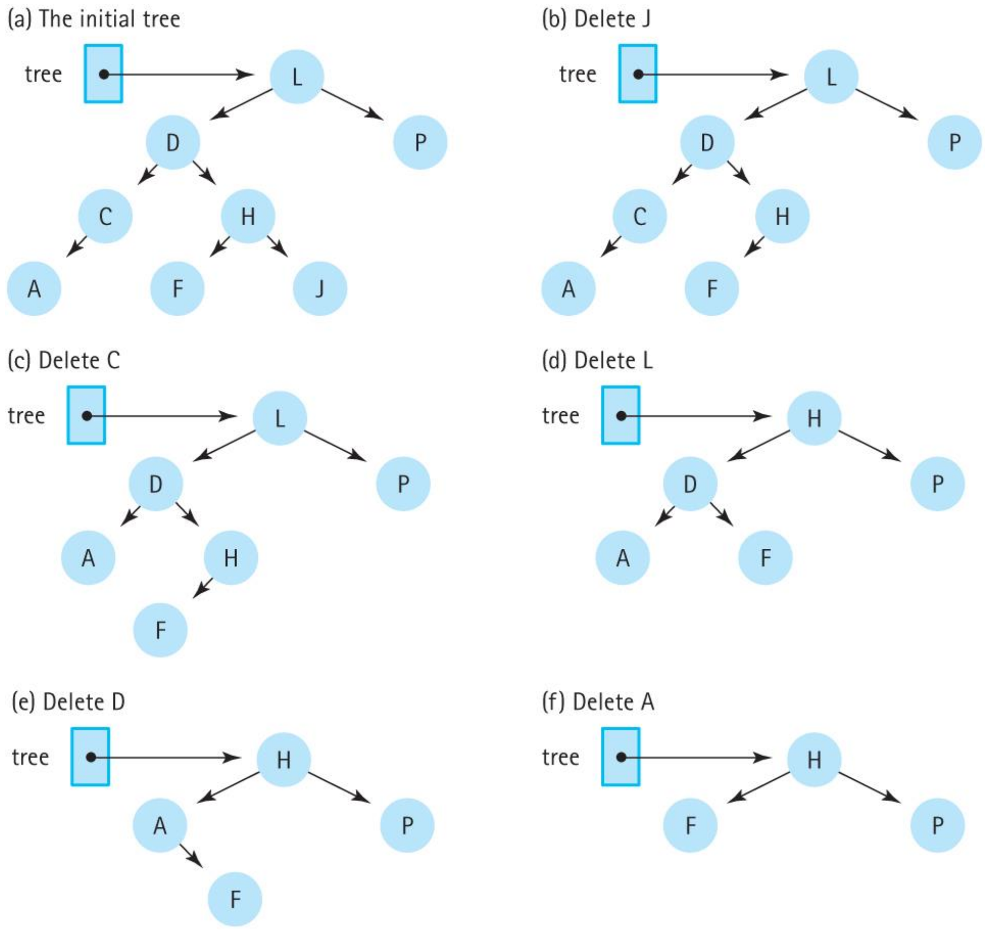


Figure 8.14 Deletions from a binary search tree (a) The intiial tree (b) Delete J (c) Delete C (d) Delete L (e) Delete D (f) Delete A

Print

- Printing a list is easy: Walk the list from beginning to end, printing each value
- Binary Search Trees are less straightforward
  - Do we go left or right first?
  - Should we print the subtrees or the root node first?
- The many ways to traverse a BST are useful in different contexts

Inorder Traversal

- Inorder Traversal accesses the elements of a tree from lowest to highest
- That is, first the left subtree is processed, then the root, and finally the right subtree

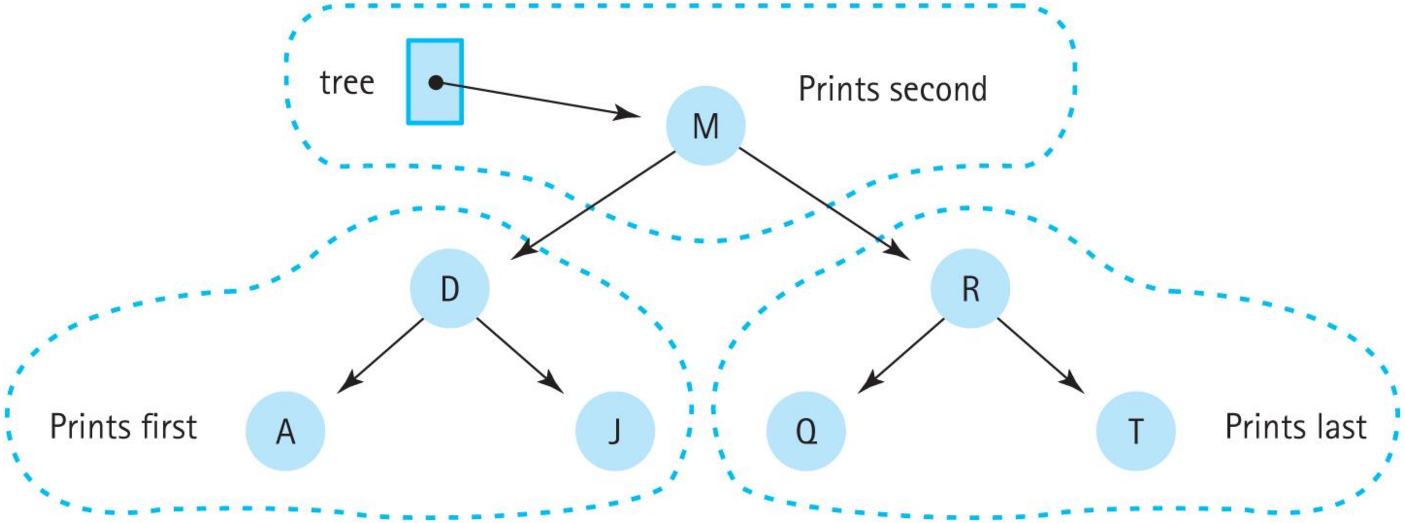


Figure 8.15 Printing all the nodes in order

Print

- Definition: Prints the items of a BST from smallest to largest
- Size: The number of items in the tree
- Base Case: If tree == NULL, do nothing
- General Case: Traverse the left subtree in order, print tree->info, traverse the right subtree in order

PrintTree

```
void PrintTree(TreeNode* tree, std::ofstream&
outFile)
// Prints info member of items in tree in
// sorted order on outFile.
{
  if (tree != NULL)
  {
    PrintTree(tree->left, outFile);
    outFile << tree->info;
    PrintTree(tree->right, outFile);
  }
}
```

Constructor and Destructor

- Constructor: Sets the root to NULL
- Destructor: Like Print, this must traverse the tree and access every node
  - Is inorder traversal the best way? Consider that leaves require less work to delete than nodes with children
  - *Postorder* traversal (traverse left subtree, then right subtree, then the node itself) will delete the children first, then the node (which is now a leaf)

Destroy Helper Code

```
void Destroy(TreeNode*& tree)
// Post: tree is empty and
// nodes have been deallocated.
{
  if (tree != NULL)
  {
    Destroy(tre->left);
    Destroy(tree->right);
    delete tree;
  }
}
```

Copying a Tree

- Both the data and the structure of the original tree are copied into the new tree
- The algorithm:
  - Create a new node and copy the original tree's information into the node
  - Copy (newTree->left, originalTree->left)
  - Copy (newTree→right, originalTree→right)
- Stops when originalTree is NULL

CopyTree

```
void CopyTree(TreeNode*& copy,
    const TreeNode* originalTree)
// Post: copy is the root of a tree that is a duplicate
// of originalTree.
{
  if (originalTree == NULL)
    copy = NULL;
  else
  {
    copy = new TreeNode;
    copy->info = originalTree->info;
    CopyTree(copy->left, originalTree->left);
    CopyTree(copy->right, originalTree->right);
  }
}
```

Copying a Tree

- Remember that taking a reference to a pointer (e.g., TreeNode*& copy) allows the function to update the pointer to point to a new object
- CopyTree will be called by both the copy constructor and the assignment operator overload function

More About Traversals

- **Inorder:** Visit the left subtree, then the node, then the right subtree. Processes values in order from smallest to largest.
- **Preorder:** The left and right subtrees of a node are processed before the node itself.
- **Postorder:** First the node is processed, then its left subtree, and finally the right subtree.
- Print, Delete, and Copy each used these.

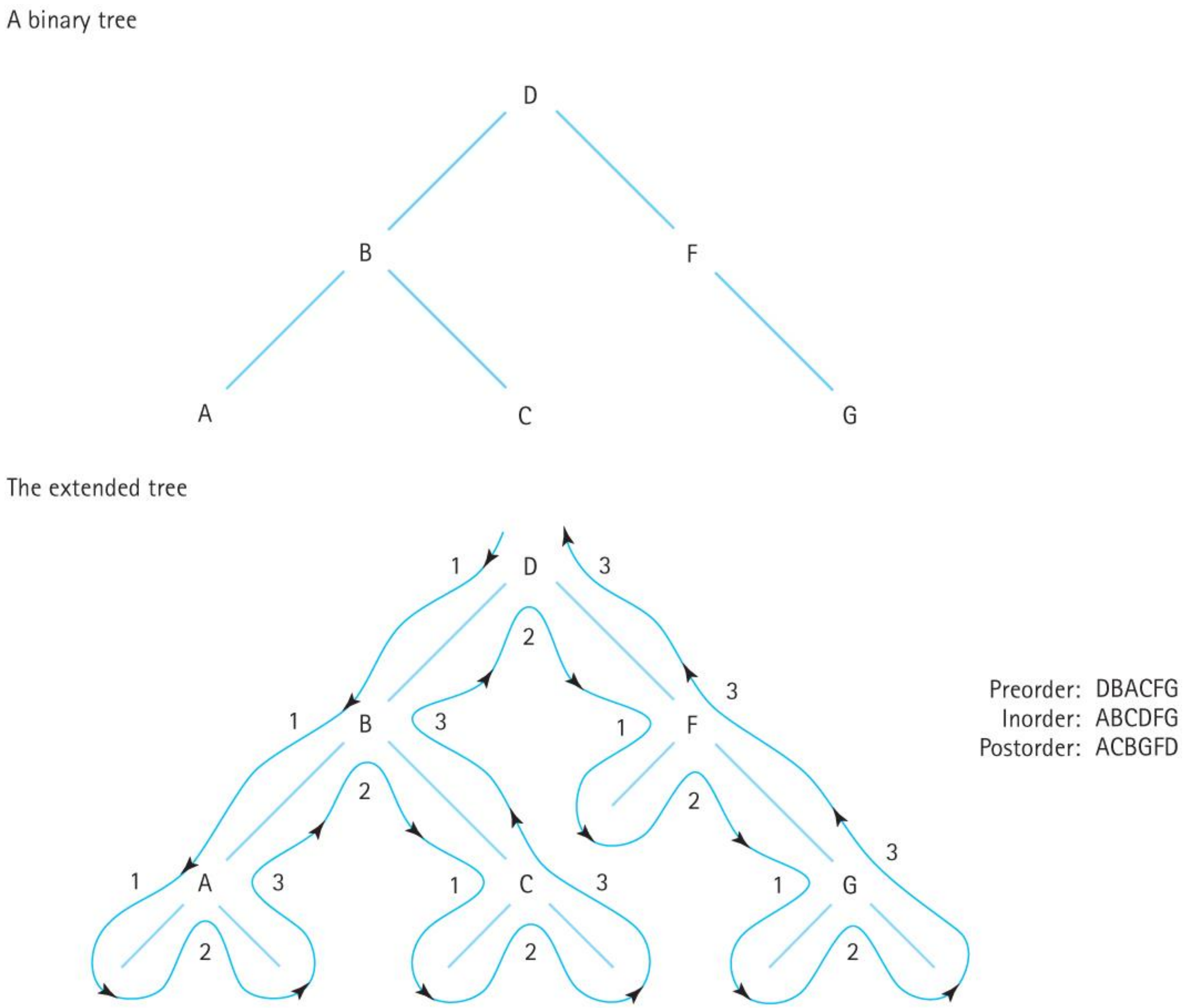


Figure 8.16 Visualizing binary tree traversals

ResetTree and GetNextItem

- Logically the same as in other ADTs: ResetTree prepares the tree for iteration and GetNextItem returns the next item.
- But which traversal should they support?
- All three traversal methods can be supported by these two functions by using a parameter to signal the traversal order.

Iterative Binary Search Trees

- It is possible to write the BST operations using iterative techniques
- Consider FindNode, which takes a tree and a node to find and returns a pointer to that node and its parent
- This can be used by PutItem and DeleteItem to find the insertion point or node to delete

FindNode

```
void FindNode(TreeNode* tree, ItemType item,
    TreeNode*& nodePtr, TreeNode*& parentPtr)
// Post: If a node is found with the same key as item's, then nodePtr points
// to that node and parentPtr points to its parent node.
// If the root node has the same key as item's, parentPtr is NULL.
// If no node has the same key, then nodePtr is NULL and parentPtr points
// to the node in the tree that is the logical parent of item
{
  nodePtr = tree;
  parentPtr = NULL;
  bool found = false;
  while (nodePtr != NULL && !found) {
    if (item < nodePtr->info) {
      parentPtr = nodePtr;
      nodePtr = nodePtr->left;
    } else if (item > nodePtr->info) {
      parentPtr = nodePtr;
      nodePtr = nodePtr→right;
    } else
      found = true;
  }
}
```

Iterative PutItem

- The task is the same: Create a new node, find the insertion point, and attach the new node.
- FindItem is used to find the insertion point
  - nodePtr should be NULL and parentPtr will point to the parent to attack the node to
- To attach the new node, set the appropriate child pointers on the parent
  - If the parent is NULL, the new node becomes the root of the tree

Iterative DeleteItem

- In the iterative version, DeleteItem uses FindItem to find the node to delete
- The rest of the deletion can be handled by DeleteNode, the helper function developed for the recursive version that is not itself recursive

Recursion or Iteration?

We can use the guidelines from Recursion Chapter to guide our decision:

- Is the recursion shallow? Yes, because well-balanced BSTs have $O(\log_2 N)$ height.
- Is the recursive version shorter or cleaner? Yes.
- Is the recursive version less efficient? No. They will always be $O(\log_2 N)$.

Binary Trees vs. Linked Lists

- BSTs combine the best features of the sorted array-based list and the linked list
- The main draw is the $O(\log_2 N)$ searching, insertion, and deletion operations
- The extra pointers do take up more memory, and the algorithms are a little more complex
- BSTs are better suited to random rather than sequential access of elements

Binary Trees vs. Linked Lists: Big-O

- Binary Search Tree efficiency relies on balanced trees; *degenerate* trees (those which look like linked lists) are not as efficient
- BSTs match the performance of array-based sorted list when searching for items
- GetLength can be O(*N*), though maintaining a length field will make it O(1)

Binary Trees vs. Linked Lists: Big-O

Table 8.2 Big-O Comparison of List Operations

	Binary Search Tree	Array-Based Linear List	Linked List
Class constructor	O(1)	O(1)	O(1)
Destructor	O(N)	O(1)*	O(N)
`MakeEmpty`	O(N)	O(1)*	O(N)
`GetLength`	O(N)	O(1)	O(1)
`IsFull`	O(1)	O(1)	O(1)
`IsEmpty`	O(1)	O(1)	O(1)
`GetItem`			
Find	O($\log_2 N$)	O($\log_2 N$)	O(N)
Process	O(1)	O(1)	O(1)
Total	O($\log_2 N$)	O($\log_2 N$)	O(N)
`PutItem`			
Find	O($\log_2 N$)	O($\log_2 N$)	O(N)
Process	O(1)	O(N)	O(1)
Total	O($\log_2 N$)	O(N)	O(N)
`DeleteItem`			
Find	O($\log_2 N$)	O($\log_2 N$)	O(N)
Process	O(1)	O(N)	O(1)
Total	O($\log_2 N$)	O(N)	O(N)

*If the items in the array-based list could possibly contain pointers, the items must be deallocated, making this an O(N) operation.

Table 8.2 Big-O Comparison of List Operations

The End!