Chapter 10:
Trees

Data Abstraction & Problem Solving with C++

Fifth Edition

by Frank M. Carrano

Categories of Data-Management Operations

- General: Operations that
  - Insert data into a data collection
  - Delete data from a data collection
  - Ask questions about the data in a data collection
- Position-oriented ADTs: Operations that
  - Insert data into the i^{th} position
  - Delete data from the i^{th} position
  - Ask a question about the data in the i^{th} position
  - Examples: list, stack, queue, binary tree

Categories of Data-Management Operations

- Value-oriented ADTs: Operations that
  - Insert data according to its value
  - Delete data knowing only its value
  - Ask a question about data knowing only its value
  - Examples: sorted list, binary search tree

Terminology

- 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

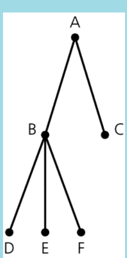


Figure 10-1 A general tree

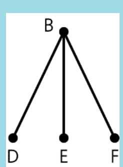


Figure 10-2

A subtree of the tree in Figure 10-1

Terminology

- General tree
  - A general tree T is a set of one or more nodes such that T is partitioned into disjoint subsets:
    - A single node r, the root
    - Sets that are general trees, called subtrees of r

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

7

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

8

A Binary Tree

- A binary tree is a set *T* of nodes such that either
  - *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*

9

A General Tree & A Binary Tree

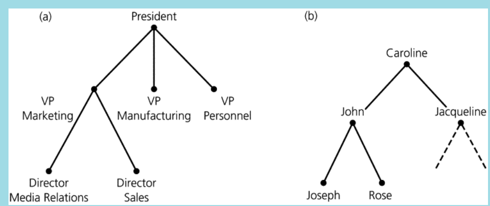


Figure 10-3 (a) An organization chart; (b) a family tree

10

More Binary Trees

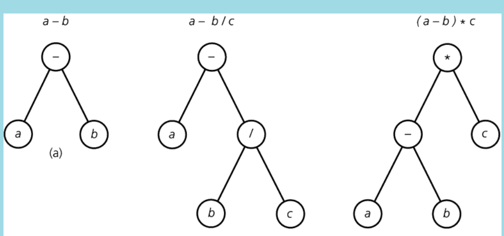


Figure 10-4 Binary trees that represent algebraic expressions

11

A Binary Search Tree

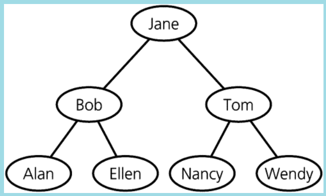
- A binary search tree
  - A binary tree that has the following properties for each node *n*
    - *n*'s value is > all values in *n*'s left subtree T_L
    - *n*'s value is < all values in *n*'s right subtree T_R
    - Both T_L and T_R are binary search trees

12

A Binary Search Tree

Figure 10-5

A binary search tree of names



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

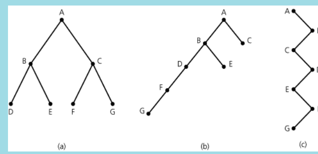


Figure 10-6

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,

    $height(T) = 1 + \max\{height(T_L), height(T_R)\}$

```
    r
   / \
 T_L   T_R
```


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 $h - 1$

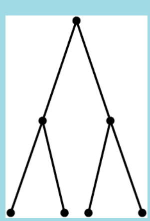


Figure 10-7

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

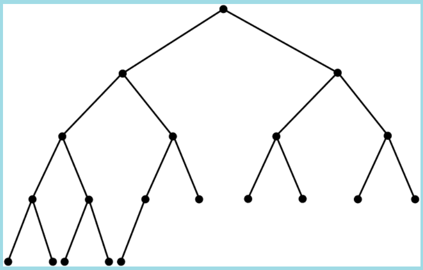


Figure 10-8 A complete binary tree

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

The ADT Binary Tree

Figure 10-9

UML diagram for the class `BinaryTree`

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)
```


Traversals of a Binary Tree

- A traversal visits each node in a tree
- You 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

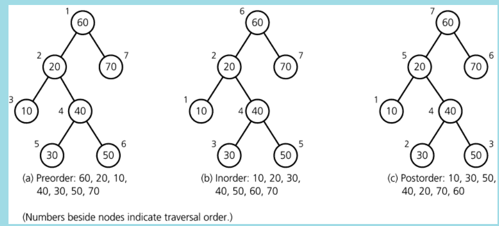


Figure 10-10
Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

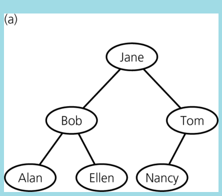
Traversals of a Binary Tree

- 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

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
- A pointer-based representation
  - Nodes have two pointers that link the nodes in the tree

Array-based ADT Binary Tree



(b) tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	–1
3	Alan	–1	–1
4	Ellen	–1	–1
5	Nancy	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
⋮	⋮	⋮	⋮

root: 0

free: 6

Rows 6 onward: Free list

Figure 10-11 (a) A binary tree of names; (b) its array-based implementation

Array-based Representation of a Complete Binary Tree

- If a binary tree is complete and remains complete
  - A memory-efficient array-based implementation is possible AND attractive

Array-based Representation of a Complete Binary Tree

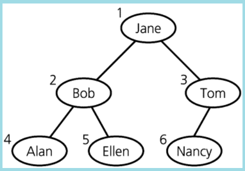


Figure 10-12 Level-by-level numbering of a complete binary tree

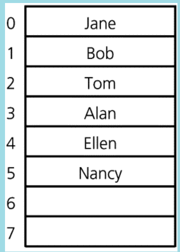


Figure 10-13 An array-based implementation of the complete tree in Figure 10-12

Pointer-based ADT Binary Tree

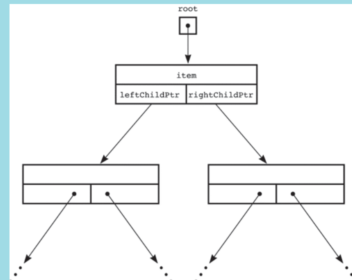


Figure 10-14 A pointer-based implementation of a binary tree

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

Pointer-based ADT Binary Tree

- `BinaryTree` class (continued)
  - `isEmpty`, `getRootData`, `setRootData`
  - `attachLeft`, `attachRight`
  - `attachLeftSubtree`, `attachRightSubtree`
  - `detachLeftSubtree`, `detachRightSubtree`
  - `getLeftSubtree`, `getRightSubtree`
  - Overloaded assignment operator

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`

Pointer-based ADT Binary Tree: Recursive Inorder Traversal

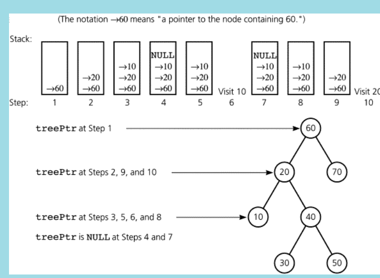


Figure 10-15 Contents of the implicit stack as `treePtr` progresses through a given tree during a recursive inorder traversal

Pointer-based ADT Binary Tree: Nonrecursive Inorder Traversal

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

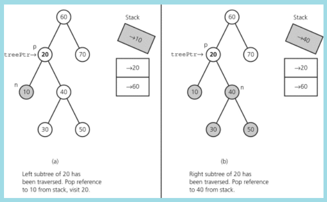


Figure 10-16

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
- The ADT binary search tree is suitable for this task
- Record
  - A group of related items, called fields, that are not necessarily of the same data type
- Field
  - A data element within a record

The ADT Binary Search Tree

- A data item in a binary search tree has a specially designated search key
  - A search key is the part of a record that identifies it within a collection of records
- `KeyedItem` class
  - Contains the search key as a data field and a method for accessing the search key
  - Prevents modification of the search-key value once an item is created
    - Has only a constructor for initializing the search key

The ADT Binary Search Tree

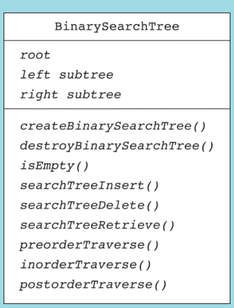


Figure 10-18 UML diagram for the class *BinarySearchTree*

ADT Binary Search Tree: Search Algorithm

- Search the binary search tree `bst` for the item whose search key is `searchKey`

```
search(in bst:BinarySearchTree,
           in searchKey:KeyType)
 if (bst is empty)
   Item not found
 else if (searchKey equals root item)
   Item is found
 else if(searchKey < root item)
   search(bst's left subtree,searchKey)
 else // searchKey > root item
   search(bst's right subtree,searchKey)
```

ADT Binary Search Tree: Insertion

- Insert `newItem` into the binary search tree to which `treePtr` points

```
insertItem(in treePtr:TreeNodePtr,
           in newItem:TreeItemType)
  if (Search stops at n’s left subtree)
    Make n’s leftChildPtr point to newItem
  else
    Make n’s rightChildPtr point to newItem
```


ADT Binary Search Tree: Insertion

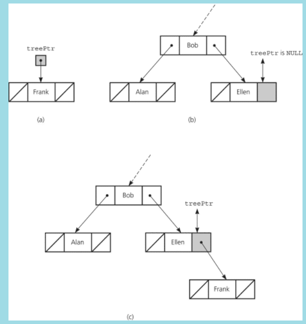


Figure 10-23
(a) Insertion into an empty tree;
(b) search terminates at a leaf;
(c) insertion at a leaf

ADT Binary Search Tree: Deletion

- Three possible cases for deleting the item in node *N*
  - *N* is a leaf
    - Set the pointer in *N*’s parent to `NULL`
  - *N* has only one child
    - Let *N*’s parent adopt *N*’s child
  - *N* has two children

ADT Binary Search Tree: Deletion

- 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: Retrieval and Traversal

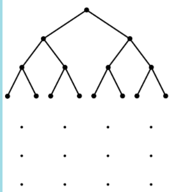
- The retrieval operation can be implemented by refining the `search` algorithm
  - Return the item with the desired search key if it exists
  - Otherwise, throw `TreeException`
- Traversals for a binary search tree are the same as the traversals for a binary tree
- Theorem 10-1

  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 $2^h - 1$ nodes
- Theorem 10-3
  - The maximum number of nodes that a binary tree of height h can have is $2^h - 1$
- Theorem 10-4
  - The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil$
  - 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

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(n^2)

Applications

- Algorithms for saving a binary search tree
  - Saving a binary search tree and then restoring it to its original shape
    - Uses preorder traversal to save the tree to a file
  - Saving a binary search tree and then restoring it to a balanced shape
    - Uses inorder traversal to save the tree to a file
    - To restore, need the number of nodes in the tree

The STL Search Algorithms

- *binary_search*
  - Returns true if a specified value appears in the given sorted range
- *lower_bound; upper_bound*
  - Returns an iterator to the first (one past the last) occurrence of a value
- *equal_range*
  - Returns a pair of iterators: One is to the first occurrence of a value in the given sorted range, the other is to one past the last occurrence

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

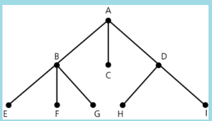


Figure 10-38 A general tree

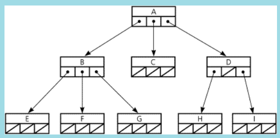


Figure 10-41

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

n-ary Trees

- A binary tree can represent an *n*-ary tree

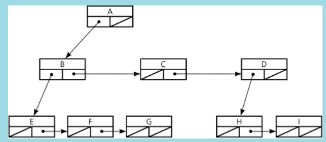


Figure 10-39 Another implementation of the tree in Figure 10-38

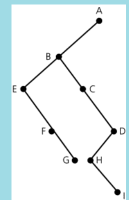


Figure 10-40 The binary tree that Figure 10-39 represents

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