ABSTRACT DATA TYPES AND SUBPROBLEMS
Where are we?

Chapter 1: The Big Picture
Chapter 2: Binary Values and Number Systems
Chapter 3: Date Representation
Chapter 4. Gates and Circuits
Chapter 5. Computing Components
Chapter 6. Low-Level Programming Languages and Pseudocode
Chapter 7. Problem Solving and Algorithms
Chapter 8. Abstract Data Types and Subproblems
Chapter 9. Object-Oriented Design and High-Level Programming languages
Chapter 10. Operating Systems
Chapter 11. File Systems and Directories
Chapter 12. Information Systems
Chapter 13. Artificial Intelligence
Chapter 14. Simulation, Graphics, Gaming, and Other Applications
Chapter 15. Networks
Chapter 16. The World Wide Web
Chapter 17. Computer Security
Chapter 18. Limitations and Computing
Chapter Goals

- Distinguish between an array-based visualization and a linked visualization
- Distinguish between an array and a list
- Distinguish between a unsorted list and a sorted list
- Distinguish between the behavior of a stack and a queue
- Distinguish between the binary tree and a binary search tree
Chapter Goals

- Draw the binary search tree that is built from inserting a series of items
- Understand the difference between a tree and a graph
- Explain the concept of subprograms and parameters and distinguish between value and reference parameters
Abstract Data Types

- A data type whose properties (data and operations) are specified independently of any particular implementation
- Remember what the most powerful tool is for managing complexity?
- **Composite data type**
  - A data type in which a name is given to a collection of data values
- **Data structures**
  - The implementation of composite data fields in an abstract data type
- **Containers**
  - Object’s whole role is to hold and manipulate other objects
Logical Implementations

- Two logical implementations of containers:
  - Array-based implementation
    - Objects in the container are kept in an array
  - Linked-based implementation
    - Objects in the container are not kept physically together, but each item tells you where to go to get the next one in the structure

Did you ever play treasure hunt, a game in which each clue told you where to go to get the next clue?
Stacks

- An abstract data type in which accesses are made at only one end
  - LIFO, which stands for Last In First Out
  - The insert is called Push and the delete is called Pop

Name three everyday structures that are stacks
Stacks

WHILE (more data)
    Read value
    Push(myStack, value)
WHILE (NOT IsEmpty(myStack))
    Pop(myStack, value)
    Write value

Can you hand simulate this algorithm?
Queues

- An abstract data type in which items are entered at one end and removed from the other end
- FIFO, for First In First Out
- No standard queue terminology
  - *Enqueue, Enque, Enq, Enter, and Insert* are used for the insertion operation
  - *Dequeue, Deque, Deq, Delete, and Remove* are used for the deletion operation

Name three everyday structures that are queues
Queues

WHILE (more data)
  Read value
  Enque(myQueue, value)
WHILE (NOT IsEmpty(myQueue))
  Deque(myQueue, value)
  Write value

Can you hand simulate this algorithm?
Stacks and Queues

Stack and queue visualized as linked structures

(a) A linked stack

(b) A linked queue
Lists

- Think of a list as a container of items
- Here are the logical operations that can be applied to lists
  - Add item: Put an item into the list
  - Remove item: Remove an item from the list
  - Get next item: Get (look) at the next item
  - more items: Are there more items?
Array-Based Implementations
Linked Implementations
Algorithm for Creating and Print Items in a List

WHILE (more data)
    Read value
    Insert(myList, value)
Reset(myList)
Write "Items in the list are "
WHILE (moreItems(myList))
    GetNext(myList, nextItem)
    Write nextItem, ' ' 
Which implementation?
Logical Level

- The algorithm that uses the list does not need to know how it is implemented.
- We have written algorithms using a stack, a queue, and a list without ever knowing the internal workings of the operations on these containers.
Trees

- Structure such as lists, stacks, and queues are linear in nature; only one relationship is being modeled.
- More complex relationships require more complex structures.
- Can you name three more complex relationships?
Trees

- A binary tree
  - A linked container with a unique starting node called the root, in which each node is capable of having two child nodes, and in which a unique path (series of nodes) exists from the root to every other node
What is the unique path to the node containing 5? 9? 7? …
Binary Search Trees

- Binary search tree (BST)
  - A binary tree (shape property) that has the (semantic) property that characterizes the values in a node of a tree
  - The value in any node is greater than the value in any node in its left subtree and less than the value in any node in its right subtree
Each node is the root of a subtree made up of its left and right children.

Prove that this tree is a BST.

**Figure 8.7** A binary search tree
Binary Search Tree

User's data

Pointer to the root of the left subtree  Pointer to the root of the right subtree
Boolean IsThere(current, item)

If (current is null)
    return false

Else
    Set result to item.compareTo(info(current))
    If (result is equal to 0)
        return true
    Else
        If (result < 0)
            IsThere(item, left(current))
        Else
            IsThere(item, right(current))
Binary Search Tree

Trace the nodes passed as you search for 18, 8, 5, 4, 9, and 15.

What is special about where you are when you find null?
**Binary Search Tree**

\[ \text{IsThere}(tree, \text{item}) \]

*IF (tree is null)*

\[ \text{RETURN FALSE} \]

*ELSE*

*IF (item equals info(tree))*

\[ \text{RETURN TRUE} \]

*ELSE*

*IF (item < info(tree))*

\[ \text{IsThere(left(tree), item)} \]

*ELSE*

\[ \text{IsThere(right(tree), item)} \]
Building Binary Search Tree
Building Binary Search Tree

**Insert(tree, item)**

IF (tree is null)

*Put item in tree*

ELSE

IF (item < info(tree))

*Insert (left(tree), item)*

ELSE

*Insert (right(tree), item)*
Print(tree)
If (tree is not null)
    Print (left(tree))
    Write info(tree)
    Print (right(tree))

Is that all there is to it? Yes!
Remember we said that recursive algorithms could be very powerful!
Graphs

- **Graph**
  - A data structure that consists of a set of nodes (called vertices) and a set of edges that relate the nodes to each other

- **Undirected graph**
  - A graph in which the edges have no direction

- **Directed graph (Digraph)**
  - A graph in which each edge is directed from one vertex to another (or the same) vertex
Graphs

Figure 8.10
Examples of graphs

(a) Vertices: People
   Edges: Siblings
Graphs

Figure 8.10
Examples of graphs

(b) Vertices: Cities
Edges: Direct Flights
Graphs

Figure 8.10
Examples of graphs

(c) Vertices: Courses
Edges: Prerequisites
Graph Algorithms

- A Depth-First Searching Algorithm
  - Given a starting vertex and an ending vertex, we can develop an algorithm that finds a path from startVertex to endVertex
  - This is called a depth-first search because we start at a given vertex and go to the deepest branch and explore as far down one path before taking alternative choices at earlier branches.
**Depth First Search**(startVertex, endVertex)

Set found to FALSE

Push(myStack, startVertex)

WHILE (NOT IsEmpty(myStack) AND NOT found)
    Pop(myStack, tempVertex)
    IF (tempVertex equals endVertex)
        Write endVertex
        Set found to TRUE
    ELSE IF (tempVertex not visited)
        Write tempVertex
        Push all unvisited vertexes adjacent with tempVertex
        Mark tempVertex as visited
    IF (found)
        Write "Path has been printed"
    ELSE
        Write "Path does not exist")
Can we get from Austin to Washington?

Figure 8.11 Using a stack to store the routes
Can we get from Austin to Washington?
Breadth-First Search

- What if we want to answer the question of how to get from City X to City Y with the fewest number of airline stops?
- A Breadth-First Search answers this question
- A Breadth-First Search examines all of the vertices adjacent with startVertex before looking at those adjacent with those adjacent to these vertices
- A Breadth-First Search uses a queue, not a stack, to answer this above question Why??
Breadth First Search(startVertex, endVertex)

Set found to FALSE

Enque(myQueue, startVertex)

WHILE (NOT IsEmpty(myQueue) AND NOT found)
    Deque(myQueue, tempVertex)
    IF (tempVertex equals endVertex)
        Write endVertex
        Set found to TRUE
    ELSE IF (tempVertex not visited)
        Write tempVertex
        Enque all unvisited vertexes adjacent with tempVertex
        Mark tempVertex as visited

IF (found)
    Write "Path has been printed"
ELSE
    Write "Path does not exist"
Breadth-First Search Traveling from Austin to Washington, DC
Subprogram Statements

- We can give a section of code a name and use that name as a statement in another part of the program.
- When the name is encountered, the processing in the other part of the program halts while the named code is executed.
What if the subprogram needs data from the calling unit?

Parameters

Identifiers listed in parentheses beside the subprogram declaration; sometimes called formal parameters

Arguments

Identifiers listed in parentheses on the subprogram call; sometimes called actual parameters
Subprogram Statements

- **Value parameter**
  - A parameter that expects a copy of its argument to be passed by the calling unit

- **Reference parameter**
  - A parameter that expects the address of its argument to be passed by the calling unit
Subprogram Statements

Think of arguments as being placed on a message board
Subprogram Statements

Insert(list, item)  // Subprogram definition
Set list.values[length-1] to item
Set list.length to list.length + 1

Insert(myList, value)  // Calling statement

Which parameter must be by reference?