1 Abstract Data Types and Subprograms
CST111—Introduction to Information Technology

2 Abstract Data Types
- Abstract data type—a data type whose properties (data and operations) are specified but nothing is known of their implementation
- Defined indirectly only be the operations that may be performed on it and mathematical constraints on the effects of those operations
- They are called containers because their only job is to hold other objects

3 Three Views of Data (Page 1)
- There are three ways to look at data:
  - Application (user) level—
    - View of the data within a particular problem
    - This view sees data objects in terms of properties and behaviors
  - Logical (abstract) level—
    - Abstract view of the data and the set of operations to manipulate them
    - View sees data objects as groups of objects with similar properties and behaviors

4 Three Views of Data (Page 2)
- There are three ways to look at data (con.):
  - Implementation level—
    - A specific representation of the structure that holds the data items and the coding of the operations in a programming language
    - View sees the properties represented as specific data fields and behaviors which are represented as methods implemented in code
    - It is concerned with data structures, i.e. the implementation of composite data field in an abstract data type

5 Abstract Data Type Features
- Composite data type—a data type in which a name is given to a collection of data values
  - Both arrays and data structures are collections
- Data structures—the implementation of a composite data fields in an abstract data type
- Containers—objects whose whole role is to hold and manipulate other objects

6 Logical Implementations
- Two logical implementations of containers:
  - Array-based implementation—objects in the container are kept in an array
  - Linked-based implementation—related 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

7 Stacks (Page 1)
- Stack—an abstract data type in which accesses are made at only one end
  - Called LIFO, which stands for Last In First Out
- Insert is called Push and delete is called Pop

8 Stacks (Page 2)
WHILE (more data)
  Read value
  Push(myStack, value)
WHILE (NOT isEmpty(myStack))
  Pop(myStack, value)
  Write value

9 Queues (Page 1)
- Queue—an abstract data type in which items are entered at one end and removed from the other
Queues (Page 2)

- 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.

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

Lists (Page 1)

- A list as a container of items, such as guest lists, grocery lists, class lists, things-to-do list, etc.
- Three properties of lists:
  - Homogeneous—items in the list are logically related to each other
  - Linear—except for the first item, all items have a unique component that comes before it; and except for the last item, all items have a unique component that comes after it
  - Lists have varying lengths

Lists (Page 2)

- Logical operations that can be applied to lists:
  - Insert—put an item into the list
  - Delete—remove an item from the list
  - IsThere—whether an item exists in the list
  - GetLength—report number of items in the list

Lists (Page 3)

- Additional mechanisms to view items in sequence:
  - Reset—start over at the first item in the list
  - GetNext—get (look) at the next item
  - MoreItems—are there more items?

Lists (Page 4)

- A linked list structures is based on the concept of a node which consists of two pieces of information
  - The user’s data
  - A link or pointer that indicates where to find next node (the end of the list is a link that contains null)

Algorithm for Creating and Printing 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, " 

Trees (Page 1)

- A hierarchical structure that begins with a starting node (the root) at the top, i.e. an organizational chart or genealogy chart
It is not the child of any other node
- Each node may have one or more child nodes

Trees (Page 2)
- Binary tree—linked tree container in which each node is capable of having two child nodes
  - A unique path (series of nodes) exists from the root to every other node; thus every node has a unique (single) parent

Trees (Page 3)
- Binary tree (con.)
  - Node to the left (if one exists) is called the left child
  - Node to the right (if one exists) is called the right child
  - If there is only one child it may appear on either side; however it will be on the same side for all nodes in the tree that have just one child
  - A node in the tree with no child is called a leaf
  - Each child is the root of a smaller binary tree (subtree)

Trees (Page 4)
- Binary tree (con.)
  - A node is the ancestor of another node if it is the parent of the node, or the parent of some other ancestor of that node; the root is the ancestor of every node in the tree

Binary Search Trees
- Binary search tree (BST)—has the shape property of a binary tree as well as the semantic property that characterizes the values in a node of a tree:
  - A binary tree has similarities to a sorted list
    - The value in any node is greater than the value in any node in its left subtree
    - The value in any node is less than the value in any node in its right subtree

Binary Search Tree
IsThere(tree, item)
IF (tree is null)
  RETURN FALSE
ELSE
  IF (item equals info(tree))
    RETURN TRUE
  ELSE
    IF (item < info(tree))
      IsThere(left(tree), item)
    ELSE
      IsThere(right(tree), item)

Building a Binary Search Tree
- If you follow the search algorithm for an item you want to add to a binary search tree, insert the item when you reach the null node

Building a 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)
31 □ Print a Binary Search Tree

Print(tree)
If (tree is not null)
Print (left(tree))
Write info(tree)
Print (right(tree))

33 □ 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

37 □ 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
  • Called a depth-first search because we start at a given vertex and go to the deepest branch and exploring as far down one path before taking alternative choices at earlier branches

38 □ A Depth-First Searching Algorithm

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")

41 □ Breadth-First Search (Page 1)
- 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 the question above. Why?

42 □ Breadth-First Search (Page 2)

Breadth First Search(startVertex, endVertex)
Set found to FALSE
Enqueue(myQueue, startVertex)
WHILE (NOT IsEmpty(myQueue) AND NOT found)
  Dequeue(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

Breadth-First Search (Page 3)

IF (found)
    Write "Path has been printed"
ELSE
    Write "Path does not exist"

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
- Methodology that takes the “divide and conquer” approach—modularization

Parameters and Arguments
- But what if the subprogram needs data from the calling unit?
- Parameters—identifiers listed in parentheses beside the subprogram declaration (header)
  - Sometimes called formal parameters
- Arguments—identifiers listed in parentheses on the subprogram call
  - Sometimes called actual parameters

Passing by Value or Reference
- Value parameter—a parameter that expects a copy of its argument to be passed by the calling unit (previous example)
- Reference parameter—a parameter that expects the address of its argument to be passed by the calling unit (next example)