Chapter 8

Abstract Data Types and Subprograms
Do you remember what is the most powerful tool for managing complexity?
Remember from Ch.7: Three types of abstraction

**Data abstraction**
Separation of the logical view of data from their implementation

**Procedural abstraction**
Separation of the logical view of actions from their implementation

**Control abstraction**
Separation of the logical view of a control structure from its implementation

Unsigned integers can be implemented on 8, 16, 32, or 64 bits!

Subprograms do this!

A for loop is the same in pseudocode, but can have different syntax details in different languages! Compilers/interpreters do it.
Abstract data type = A data type whose properties (data and operations) are specified independently of any particular implementation.

The terms container and black-box are used to describe ADTs.

Python strings and lists have their own methods attached to them.
Three Views of Data

Application (user) level view = View of the data within a particular problem
The data is seen in terms of its physical properties and behaviors.

Example: A robot has 6 SONAR sensors around its body. Each SONAR measures a distance to the nearest obstacle in front of it.
Three Views of Data

Logical (abstract) level view

View sees data objects as groups of objects with similar properties and behaviors.

Example: The 6 distances measured by SONARs are stored in an array of integers. The array has a length of 6, and it supports initialization, printing, searching, and sorting.
Three Views of Data

Implementation level view = A specific representation of the structure that hold the data, and the coding of the operations in a programming language.

View sees the properties represented as specific data fields and behaviors represented as functions or methods implemented in code.

Example: The array is implemented as a Python list, with all the operations, functions and methods (e.g. slicing, `len()`, `append()`) that lists have in Python.

```python
data = [0]
data = data * 20

length = 6
data[0] = 60
data[1] = 75
data[2] = 95
data[3] = 80
data[4] = 65
data[5] = 90

print data[0:length]
```
Describe how a **word processor** can be seen from the three views:

• user/application
• logical/abstract
• implementation (in Python)
Describe how a **thermostat** can be seen from the three views:

- user/application
- logical/abstract
- implementation (in Python)
Stacks

**Stack** = An ADT in which accesses are made at only one end

- LIFO = Last In First Out

- The insert is called **Push** and the removal is called **Pop**

Name three everyday structures that are stacks
Stack algorithms

Placing elements in the stack:

WHILE (more data)
  Read value
  \textbf{push}(myStack, value)

Removing elements from the stack:

WHILE (NOT IsEmpty(myStack))
  \textbf{pop}(myStack, value)
  Write value
A stack is initially empty. Draw the stack after each of these operations:

- push(42)
- push(15)
- push(10)
- push(10)
- pop()
- push(21)
- pop()
Queues

**Queue** = An ADT in which items are inserted at one end and removed from the other end

– FIFO = 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 removal operation.

Name three everyday structures that are queues
Interesting trivia: **queueing** is one of only two words in the English language that has 5 vowels in a row!

E.g. Queueing Theory

4 vowels in a row is more common:
queue (!), aqueous, onomatopoeia, sequoia, archaeoastronomy, etc.
Queue algorithms

Placing elements in the queue:

WHILE (more data)
  Read value
  enqueue(myQueue, value)

Removing elements from the queue:

WHILE (NOT IsEmpty(myQueue))
  dequeue(myQueue, value)
  Write value
A queue is initially empty. Draw the queue after each of these operations:

- enqueue(42)
- enqueue(15)
- enqueue(10)
- dequeue()
- dequeue()
- enqueue(21)
- dequeue()
- dequeue()
Two main implementations of ADTs

Array-based implementation
Objects in the container are kept in an array, i.e. physically next to each other in memory.

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 tells you where to go to get the next clue?
Stacks and Queues in the linked implementation

(a) A linked stack

(b) A linked queue
Why access elements only at the ends?
Lists

List are more general ADTs than stacks and queues

Logical operations that can be applied to lists:

- There is an index for the *current item* *(crt)*
- **Add item** Put an item into the list (before *crt*)
- **Remove item** Remove the *crt* item from the list
- **Get next item** Get (look at, copy, print) the next item (after *crt*)
- **Get *crt* item**
- **Move to next item** *crt* points to the next item
- **More items** Are there more items beyond *crt*?
Linked-list implementation
QUIZ: Show the list and the outputs (if any) after each of the following operations:

moveNext()  
moveNext()  
getCrt()  
getNext()  
remove()  
insert(42)  
moreItems()  
moveNext()  
getNext()
Array-based implementation of a list

crt
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, " "

Brings crt back to the head of the list
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, '

Trick question:
Which implementation is being used (array or linked)?
Logical Level

The algorithm that uses the list does not need to know how the data in the list is stored (array or linked), or how the various operations (Insert(), Reset(), moreItems()) are implemented!

We have written algorithms using a stack, a queue, and a list without ever knowing the internal workings, i.e. the implementation of these containers.
8.5 Trees

Structure such as lists, stacks, and queues are **linear** in nature; only **one relationship** is being modeled.

Other (more or less complex) relationships require different structures.
Binary Tree (BT)

- Root node
- Node with two children
- Node with right child
- Leaf node
- Node with left child
**Binary tree** = A linked container with a unique starting node called the **root**, in which each node can have up to **two child nodes**

- A node can have **0, 1, or 2** children

A **unique path** (series of nodes) exists from the root to every other node.

- There are **no loops!**
Write the (unique) path from the root to the node containing:

- 7
- 8
- 6
List all the nodes having:

- 0 children
- 1 child
- 2 children
Putting a BT to work: Binary Search Tree (BST)

Do you notice a pattern in this BT?
Binary Search Tree = A tree with the following properties:

1. **Shape property** → It is a binary tree
2. **Semantic property** → For every node, the value stored in the node is
   - greater than the value in any node in its left subtree and
   - smaller than the value in any node in its right subtree
We have already encountered the null pointer in linked lists
Algorithm for searching an item in a BST

```
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)

Recursive!
```
QUIZ BST:
Search for item 18

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)
Extra-credit QUIZ:
Can you spot a similarity between this “tree search” and binary search?

```
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)
```

```
15

7

5

1

6

8

10

16

18

19

21
```

```
tree
```
In order for tree search to be efficient, the tree has to be balanced.

Best case

Worst case

Balanced

We do not cover balancing algorithms in this class (see CS 241 - Data Structures)
Inserting an item in a BST: first we have to search for it!

Search for Kyrsten in this tree ...
Inserting an item in a BST: first we have to search for it!

Search for Kyrsten in this tree ...

... and insert the item where the search ended!
**Insert**(tree, item)

IF (tree is null)
   *Put item in a new tree*
ELSE
   IF (item < info(tree))
      **Insert** (left(tree), item)
   ELSE
      **Insert** (right(tree), item)

*Note how similar this algorithm is to the search algorithm!*
QUIZ on BST insertion

Problem 47/280:
The following elements are inserted in an initially empty BST:
50, 72, 96, 107, 26, 12, 11, 9, 2, 10, 25, 51, 16, 17, 95
Find the final tree.
Printing/traversing a BST

**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 that recursive algorithms can be very powerful
Printing/traversing a BST

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

This is called “in-order” traversal
Other tree algorithms

Finding the **depth** of a tree:

\[
\text{Depth}(\text{tree}) \\
\text{IF (tree is null)} \\
\text{RETURN 0} \\
\text{ELSE} \\
\text{RETURN max(\text{Depth(left(tree)), \text{Depth(right(tree))}}))}
\]

Finding the **length** of a tree:

\[
\text{Length}(\text{tree}) \\
\text{IF (tree is null)} \\
\text{RETURN 0} \\
\text{ELSE} \\
\text{RETURN 1 + \text{Length(left(tree))} + \text{Length(right(tree))}}
\]
8.6 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.

Note: The “unique path” condition from trees has been removed; graphs are more general than trees!
Graphs: The non-uniqueness of paths generates cycles and unconnected parts.

(a) Vertices: People
Edges: Siblings
QUIZ: Find 5 cycles in this graph
QUIZ: In each case, decide if the data structure is a tree or just a graph.
Graphs: directed and undirected

(b) Vertices: Cities
Edges: Direct Flights
Graphs

(c) Vertices: Courses
Edges: Prerequisites
Creating a Graph: adjacency matrix

See Table 8.3/262
QUIZ: Draw the adjacency matrix for this graph
Extra-credit QUIZ:
If in a graph all edges are bi-directional, what property does the adjacency matrix have?
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 exploring as far down one path before taking alternative choices at earlier branches
Graph Algorithms

We are covering only the Depth-First Search (DFS) algorithm

... but what exactly are we searching for?

An item stored in a node? Cycles? Spanning trees? Hamiltonian cycles?
Searching for paths between given nodes

Can you find a path from Austin to Washington?
**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 vertices adjacent to tempVertex

        Mark tempVertex as visited

    END IF

IF (found)

    Write "Path has been printed"

ELSE

    Write "Path does not exist")
Using DFS to find a path from Austin to Washington

startVertex ———

Austin ——— Dallas ——— Washington
——— 200 ——— 1300 ———
——— 780 ———
——— 900 ——— 1400
——— 1000 ———

Denver ——— Atlanta
——— 800 ———
——— 800

Houston ———

endVertex

Figure 8.11 Using a stack to store the routes

Stack

Austin

Stack

Houston

Stack

Atlanta

Stack

Washington

a. b. c. d.
Using DFS to find a path from Austin to Washington

Why don’t we push Houston here?
Is this the only path from Austin to Washington?
Is this the shortest path from Austin to Washington?
Conclusion on DFS: The algorithm is guaranteed to find a path (if at least one exists) between the given vertices.

To find the shortest path, we need to use different algorithms, e.g. BFS or SSSP (not covered)
8.7 Subprogram Statements

**Value parameter** = A parameter that expects a copy of its argument to be passed by the calling unit
• Changes made to it are not visible outside

**Reference parameter** = A parameter that expects the address of its argument to be passed by the calling unit
• Changes made to it are visible outside
Subprogram Statements

Think of arguments as being placed on a message board
In Python, the scalar and list arguments are being passed as value arguments …

… but the individual list elements are reference arguments!
Chapter Review Questions

• Distinguish between an array-based visualization and a linked visualization
• Distinguish between an array and a list
• Distinguish between and 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 Review Questions

• 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
Who am I?

In the 1940s, the new computer architecture I developed revolutionized the design of computers. Today’s computers are referred to as [my last name] machines because the architectural principles I described have proven very successful.
Do you know?

What are some of the economic stimulus scams uncovered by the IRS in 2007?

What is the Open Source Hardware Bank (OSHB)? Explain its connection to improving the function and expanding the capabilities of hardware

How does graph theory relate to terrorist detection?
Homework for Ch.8 (last hw!), due Friday, April 27

1 through 10, 37, 51 through 55, 56, 57