COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 7: 13 February 2019

Lecturer: Dr. Andrew Hines *Scribes: Candis Anderson and Sarah Pender*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

7.1 Outline

This lecture introduces the idea of Abstract Data Types and Concrete Data Structures. We will cover the definition of these terms as well as how to use ADTs and CDSs in the confines of data structure and algorithms.

7.1.1 Feeling Puzzled?

Jigsaw puzzles are a great way to reflect on what we are trying to do with algorithms. A jigsaw puzzle is a great representation of a set in action.

How to you put them together?

You execute a sorting algorithm! Either you look at the pieces that match structurally (start with the edges!) or you look at the colors - then put them in groups using iteration.

If you think about piecing a puzzle together using set theory and Venn diagrams with the edges in one category and the middles in another... -the union between gives us all of the pieces in the puzzle -the intersection would give us pieces that are both edges and middles (which do not exist) -the difference will show pieces that are an edge but not a middle - or vice versa - so that we can match them up

aside jigsaw puzzles are an example of a set - different from a sequence because a set is unordered and does not contain duplicates.

7.2 Abstract Data Types (ADTs)

Independent of any coding language and how well we have implemented it, the Abstract Data Type gives us a representation of how the data can be interacted with. In programming, we use pseudo-code in order to abstract ourselves away from the specific implementations of the particular language we are using - we use it to describe what we are wanting to do at a high level. Similarly, we use Abstract Data Types to abstract data structures to a higher level. Another way to think of an Abstract Data Type is as a class definition in Python - you have the data itself and must list how you can interact with the data. Abstract data types have been given many names all with a slight variation in their definition, these names include Abstraction, Encapsulation, Modularity, and Information Hiding.

7.2.1 Sequences

Sequences are an example of an Abstract Data Type - a sequence is merely a set of objects (of any type) that are in sequential order.

The sequence is *implemented* as a string or list or tuple (depending on the programming language).

The sequence is one of the most basic and fundamental types used to understand Abstract Data Types and there are many variations of a sequence, but in essence a sequence is an ordered list. A sequence can be represented as an array or linked-list. The sequence will have an index that is ordered from 0 and the data set inside the sequence will be in sequential order.

ex: Index 0 1 2 3 4 Set 2 6 9 17 33

7.2.2 ADT Sequence Operations

There are rules for interacting with a sequence, and we call those the main operations and they are listed below. Note that an insert will shift the set values (in relation to the index) to the right and delete will shift the values to the left.

Main Operations
`get elem at rank(r)`: Return the element of S with rank r; an error occurs if r¡0 or r¿ n - 1
Input: Integer; Output: Object
`set elem at rank(r,e)`: Replace the element at rank r with e; an error occurs if r ¡ 0 or r ¿ n - 1.
Input: Integer r, Object e; Output: none
`insert elem at rank(r,e)`: Insert a new element into S which will have rank r; an error occurs if r ¡ 0 or r ¿ n - 1.
Input: Integer r, Object e; Output: none
`remove elem at rank(r)`: Remove from S the element at rank r; an error occurs if r¡0 or r¿n-1.
Input: Integer; Output: none

Secondary operations make it easier for you to interact with the sequence - they break transactions that would take 2-3 operations into just one operation.

Secondary Operations
`insert first(e)`, `insert last(e)`: Insert an element at one of the ends of the sequence
Input: element; Output: none
`insert after(p, e)`, `insert before(p, e)`: Insert an element after or before a position
Input: position and element; Output: none

Supporting Operations
`size()`: Returns the number of objects in the sequence
Input: none; Output: integer
`is empty()`: Return a boolean indicating if S is empty.
Input: none; Output: boolean

7.2.3 Data Storage

The size of data has an impact on how functions work. For example, a float takes up more storage space than an integer. A string, depending on its size, could take up more space than a float.

7.3 Array-based Data Structures

An array is used to store multiple items in one variable. Elements of an array can be accessed using an index and are easily accessed. There are a fixed number of values in an array and they are all of the same value type.

7.4 Linked List Characteristics

A linked-list is a linear collection of elements or nodes pointing to each other. Each element in a linked-list is linked to its successor using a pointer. The starting node of a linked-list is called a header. Linked-lists are not part of a standard Python library.

Linked-lists are more memory efficient than arrays as with an array you must specify its size before adding values, whereas a linked-list only uses the amount of memory that is needed depending on the number of values that are entered.

7.5 Array-Based Sequence

ADTs often follow specific patterns such as Get and Set patterns, Insert patterns, and Delete patterns. Get and Set patterns are used for error checking and editing data. Insert and delete patterns are often quite complex in their validation, error checking and memory management.