CSE 373: The ADT Toolbox

Miscellaneous

What We’ve Done

Up to this point, we’ve looked at a variety of fundamental data structures, each with its own unique strengths and limitations:

- **List:** general-purpose storage
- **Stack:** FIFO ordering
- **Queue:** LIFO ordering
- **Tree:** hierarchical organization
- **BST:** searchable storge
- **Hash Table:** quick storge
- **Heap:** quick location of minimum
The Toolbox

- These data structures are not the only ones you’ll ever use or need
- Rather, think of them as a *basis set* from which you can build other data structures
  - by mixing multiple data structures
  - by adding additional functionality
  - by relaxing the “pure” version of the data structure

Mixing Data Structures

As we saw on day one, C’s arrays and structures can be mixed and matched:

- **array of arrays**
- **array of structures**
- **structure of arrays**
- **structure of structures**
More Mixing Data Structures

This same thing can be done for all the data structures we’ve used in this class:

- The Bash interpreter was a simple example of this (hash table of structs of trees...)

Adding Additional Functionality

In addition to the usual implementation of the data structure, add some more information

e.g., maze-solving example:

- most people kept a list of the moves they made to get to their current position
- once the goal was found, you could iterate over this list to find out its length (and if it was the shortest path)
- OR you could add a new field (inside or outside the List ADT) that would keep the length as the list grew
- though this doesn’t change the algorithm asymptotically (this part of it is still $O(n)$), it may improve elegance
Relaxing the “Pure” ADT

We’ve already seen several examples of this:
- Microsoft’s “recent documents list” is queue-like
  - but it only holds \( n \) elements at a time (breaks arbitrary size property)
  - if an document in the queue is accessed, it is removed and re-inserted at the back (breaks LIFO property)
- Avoiding some operators can change properties
  - \texttt{findMin/findMax} cheap on hash table if no deletes
- Iteration over hash tables can be useful
  - to implement a general \texttt{findMin/findMax}, e.g.

Application: Min/Max Heap

I’d like a heap that supports both \texttt{deleteMin()} and \texttt{deleteMax()} efficiently

How could I do this?
Application: Multi-sorted List

It’s easy to imagine writing a List ADT that always inserts data in sorted order.

What if I wanted to have a list “sorted” by all of its fields?
  e.g., I’d like to print it out sorted by last name, by first name, by student ID, by grade, etc.

Sample Application: UW Registry

Our naive implementation was to declare an array of size `# students × # classes`
  – this used way too much memory for the complexity of data we were storing
  – we oversimplified “indexing by UWID” since they start at 9?????? and C arrays start at 0

What else could we do?
Next Assignment: Sparse Arrays

Sparse Array: an array in which most values are identical
- thus, it is better to store only those values that differ
- a single copy of the unrepresented value (URV) is stored

Examples:
- UW registry (most students are not taking most classes)
- Many scientific computations/simulations

Sample Sparse Array Operations

(We’ll be doing Sparse 2D arrays…)

Main operations:
- **Object Read(i, j)** – return the value at (i, j) if it’s stored, the unrepresented value otherwise
- **void Store(i, j, Object)** – store val at index (i, j); stop storing a value for that position if val == the URV

- **Iterators**: allow the user to iterate over the data by row or column
- **I/O**: support read/write of sparse arrays