CSE 373: Breaking Out

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 storage*
Hash Table: *quick storage*
Heap: *quick location of minimum*

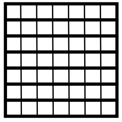
The Toolbox

- 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

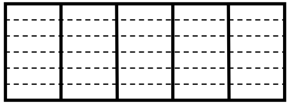
Mixing Data Structures

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

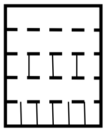
array of arrays



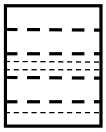
array of structures



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:

$$a\ldots\left\{\begin{array}{l}\text{List}\\ \text{Stack}\\ \text{Queue}\\ \text{Tree}\\ \text{BST}\\ \text{Hash Table}\\ \text{Heap}\end{array}\right\}\ldots of\ldots\left\{\begin{array}{l}\text{Lists}\\ \text{Stacks}\\ \text{Queues}\\ \text{Trees}\\ \text{BSTs}\\ \text{Hash Tables}\\ \text{Heaps}\end{array}\right.$$

- Our simple database was an example of this
- So is the BEL++ interpreter

Adding Additional Functionality

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

e.g., mode example:

- several people kept track of the current mode as they added each integer to their data structure
- once the mode was requested, they could just return the stored mode
- this usually did not improve the algorithm asymptotically, but may be worthwhile for other reasons (secondary performance impact or more elegant code)

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 deleted and moved to the back (breaks LIFO property)
- Avoiding some operators can change properties

  `FindMin`/**`FindMax`** cheap on hash table if no deletes
- Iteration over hash tables can be useful

  mode example: store all then iterate to find mode

Sample Application: UW Registry

Our naive implementation was to declare an array of size # *students* × # *classes*

- this used way too much memory
- we oversimplified “indexing by UWID” since they start at 9?????? and C arrays start at 0

What else could we do?

Application: Multi-sorted List

On one of the homework assignments, you implemented a sorted list using an `InsertSorted()` operation

What if I wanted to have a list sorted by multiple fields?

e.g., I'd like to print it out sorted by last name, by first name, by student ID, by grade, etc.

Application: Min/Max Heap

I'd like a heap that supports both `DeleteMin()` and `DeleteMax()` efficiently

How could I do this?

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 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 of doubles…)

- **Main operations:**
  - `double* Access(i,j)` – return a pointer to element (*i*,*j*) if it exists, NULL otherwise
  - `double Read(i,j)` – return the value at (*i*,*j*) if it exists, the unrepresented value otherwise
  - `void Write(i,j,val)` – store val at index (*i*,*j*)
- **Iterators:** allow the user to iterate over the data by row or column
- **I/O:** support read/write of sparse arrays