Abstract data types (ADTs)

- an ADT gives us a way to think about the design of our data without focusing on the code details
- top-down design gave us a way to logically decompose our code into logical subunits
- ADTs let us do something similar with our data
- for any given kind of information we want to store/manipulate, we can think about (i) the other kinds of information it is made up of, and (ii) the kinds of operations we would like to perform on the data

Example: StudentRecord

- one sample ADT we might consider is a student record
- in the context of our program, what information do we want to store about a student?
  - name: as one field (whole name), or multiple (family name, given name, etc)
  - student number: as a number, or some form of text?
  - fees owing: as one field (dollars), or two (dollars/cents)?
- what operation do we want to perform on student data?
  - get/set the name/student number/fees owing
  - print the information

Design, abstraction

- in the ADT we aren't worried about the specific language features/types we'll use to implement the student record
  - we might not even have decided which language to use!
- instead, we focus on the nature of the information we want to work on, and the kinds of things we want to do with it
- this abstract view helps us simplify and describe our design, and avoid getting bogged down in details that aren't important (yet)

Design vs implementation

- at the ADT level we are focused on the abstract data and operations
- at the implementation level, after we've agreed on our design, we'll have to develop a solution in code
- there are always many ways to implement any given design, we'll want to think about possible different implementation approaches and the pros/cons of each

Example: list of numbers

- suppose we want to store a list of numbers, initially empty but could grow to any size
- we want to be able to add new numbers, one at a time, each number added must be the “last” one to that point
- we want to be able to search the list from beginning to end to find a specific value (if it has been stored)
- we want to be able to print the list from beginning to end
- we want to be able to remove an item currently in the list, while keeping the others in their current order

Implementation idea #1

- we could implement using an array, using a counter to keep track of how many values we've entered so far and always inserting in the last position
- pros:
  - simple to code, fast lookup if we know an items position
- cons:
  - fixed size array can be a problem (too big means wasted memory, but too small means the program may not work)
  - when removing an item we'll have to shift everything after it forward one spot, or be left with gaps in the array

Implementation idea #2: linked list

- each “item” has a number and a pointer to the next and previous items, overall we keep pointers to the front and back
- to insert, we dynamically allocate an item and attach it to the back, then make it the new back
- to print, we go through each item from front to back, printing as we go
- to search, we go through each item from front to back, returning the item (pointer) if found, NULL if never found
- to remove, we search for the item, get the items before and after it to bypass the item (cutting it out), update front and back if necessary, then delete the removed item

Linked list pros and cons

- pros:
  - initially takes almost no space, and can grow to any size: we don't have to pick a size up front
- cons:
  - extra coding complexity for structs, pointers, memory handling
  - extra memory overhead for all the pointers
  - can only get to an item by (linear) search from front or back, even if we know its position in the list, which could be slow for items deep in big lists

Making a choice

- which implementation approach we choose will depend on
  - how much time we've got to develop a solution (array version quicker to code)
  - how wide a range of stored values are we expecting (if we have a good idea of roughly how many numbers will be entered then an array might be sensible, but if there is huge variation then the linked list might be better)
  - how frequently we'll perform removes from early/mid sequence (removes are much slower in the array approach if we have to shift elements forward afterward)