Lecture 17

Today

- Abstract Data Types
- Interfaces
- Dynamic Arrays
- Linked Lists
Stacks (Review)

A *stack* is an ordered collection of items, to which you may insert an item (a *push*) or remove an item (a *pop*), where removal follows a last-in-first-out order (LIFO).

- the definition of a stack was independent from its implementation
- the first example of an *abstract data type*

**Abstract data type (ADT):** a collection of data and a set of allowed operations on that data.
- describes data + operations, not how the data are stored or how operations are carried out
Stacks (Review)

A *stack* is an ordered collection of items, to which you may insert an item (a *push*) or remove an item (a *pop*), where removal follows a last-in-first-out order (LIFO).

- a stack of plates
- a stack of books
- a stack of passengers
Abstract Data Types

Abstract data type (ADT): a collection of data and a set of allowed operations on that data.

- specifies **data and operations**, not how the data are stored or how operations are carried out
- different from the **data structure**, which deals with the implementation

**Data structure**
- Implementation of the ADT

**Data + operations**
- Usage of the ADT

Interact via an interface vs.
Another Common ADT

Queue ADT: A *queue* is another sequence of data, but the insert / remove operations work on opposite ends of the sequence.

- order is first-in-first-out (FIFO)
- like a line-up

queue for service

queue of traffic

queue of food
Interfaces

An *interface* refers to an expected collection of data and behaviours

- parametrized by inputs
- serves as a contract

Q. What interfaces have you seen in CMPT 125?
- functions, pre-, post-conditions, invariants
- collections of functions, typedefs, constants
- header files
Why use interfaces?

Code re-usage

Code independence

A tale of three programmers

Algorithm that implements a resizable ADT

Interface:
- a sequence of data
- read element
- set element
- append

Algorithm that requires a resizable data type
Why use interfaces?

Code re-usage

Code independence

A tale of three programmers

Resizable ADT, implemented using arrays

Interface:
- a sequence of data
- read element
- set element
- append

Code that instantiates and uses a resizable ADT
Why use interfaces?

Code re-usage

Code independence

A tale of three programmers

Resizable ADT, implemented using arrays

Resizable ADT, implemented by linked lists

Interface:
- a sequence of data
- read element
- set element
- append

Code that instantiates and uses a resizable ADT
Why use interfaces?

Code re-usage
Code independence
A tale of three programmers

Implementation of an ADT

Interface:
- function spec
- preconditions
- return codes
- ...

Code that uses an ADT, according to its interface
Why use interfaces?

Code re-usage
Code independence
A tale of three programmers
Why use interfaces?

Code re-usage

Code independence

A tale of three programmers

- Implementation of an ADT
- Interface:
  - function spec
  - preconditions
  - return codes
  - ...
- Code that uses an ADT, according to its interface
Why use interfaces?

Code re-usage
Code independence
A tale of three programmers

Interface:
- function spec
- preconditions
- return codes
- ...

Code that uses an ADT, according to its interface
Software Engineering Principles

Encapsulation
- bundle related data and operations together

Modularity
- break up the problem into smaller, manageable programming tasks

Information Hiding
- keep the implementation details private
- keep the interface stable

Finding a good selection of interfaces is the foundation for writing large scale software
Fleshing out some ADTs

Q. What sort of data (properties) and operations (functions) would apply to:

<table>
<thead>
<tr>
<th>Stack ADT:</th>
<th>Appendable array ADT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>● a sequence of data</td>
<td>● a sequence of data</td>
</tr>
<tr>
<td>● last in first out order</td>
<td>● append (to the end)</td>
</tr>
<tr>
<td>● insert (push)</td>
<td>● size (length)</td>
</tr>
<tr>
<td>● remove (pop)</td>
<td>● access (get)</td>
</tr>
<tr>
<td>● isEmpty</td>
<td>● change (set)</td>
</tr>
<tr>
<td>● top</td>
<td></td>
</tr>
<tr>
<td>● size (length)</td>
<td></td>
</tr>
</tbody>
</table>
Appendable Array ADT

One possible implementation is an array

- keep track of current length
- keep a pointer to the array
- **access** - trivial + bounds check
- **change** - trivial + bounds check
- **append** - not so trivial - malloc and copy

Q. What’s the total running time for $N$ appends?
Linked Lists

Another Idea: malloc one item on each append
- items might not be contiguous anymore
- Q. How to find next item in the sequence?
- use a sequence of pointers

Refined Idea: malloc one item + one pointer on each append.
Linked Lists

Another Idea: malloc one item on each append
- items might not be contiguous anymore
- Q. How to find next item in the sequence?
- use a sequence of pointers

Refined Idea: malloc one item + one pointer on each append.
Linked Lists

Another Idea: malloc one item on each append
  ● items might not be contiguous anymore
  ● Q. How to find next item in the sequence?
  ● use a sequence of pointers

Refined Idea: malloc one item + one pointer on each append.
Heap Memory vs. Stack Memory

- **Heap memory**
  - Special command needed to add (malloc, new) and remove (free, delete) variables
  - Useful for ADTs that vary in size
  - Different variables typically do not occupy contiguous memory locations

- **Stack memory**
  - Memory used to hold the function call stack
    - Includes local variables and function parameters
  - No special commands or manual maintenance needed
  - Cannot resize easily, since everything is in the function call stack