CSCI 136:
Data Structures
and
Advanced Programming

Lecture 17

Linear structures

Instructor: Dan Barowy

Williams

Announcements

I hope you had a great spring break!

Midterm exam: will return graded exams via GLOW, Tues or Wed

PRE-LAB 0: due today by 5pm

Outline

1. Abstract Data Types
2. Linear ADTs
3. Stack
4. Queue

Abstract Data Type

An **abstract data type** is a mathematical formulation of a data type. ADTs abstract away **accidental** properties of data structures (e.g., implementation details, programming language). Instead, ADTs contain only **essential** properties and are **concisely defined by their logical behavior** over a **set of values** and a **set of operations**.

In an ADT, precisely how data is **represented** on a computer **does not matter**.

By contrast: data structure

A **data structure** is the physical form of a data type, i.e., it is an implementation of an ADT. Generally, data structures are designed to efficiently support the logical operations described by the ADT.

For data structures, precisely how data is **represented** on a computer **matters a lot**. Simple data structures are often composed of simple representations, like primitives, while more complex data structures are composed of other data structures.

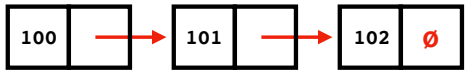
ADT example: Linked List

A **linked list** is a linear collection of data elements, whose order is not necessarily given by their placement in memory. Each element is stored in a node that points to the next node. Elements may store **any type of value**. A list supports **inserting**, **searching** for, and **deleting** any value in a list, although not necessarily efficiently.

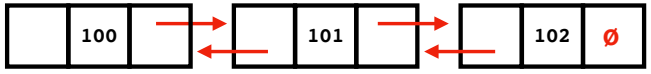
DS example: Linked List

We have seen many **implementations** of linked lists and discussed their performance tradeoffs. For example:

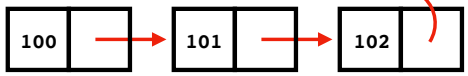
- `SinglyLinkedList`



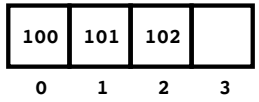
- `DoublyLinkedList`



- `CircularList`



- `Vector`



ADTs cannot be expressed in Java

At least **not directly**.

Instead, Java uses **types** to stand in for ADTs.

Because types in Java are often bound to an implementation, Java provides two mechanisms for programmers to use a type without depending on a mechanism: **interfaces** and **abstract classes**.

Interface

An **interface** defines boundary between two systems across which they share information. An interface is a **contract**: calling a method defined in an interface returns the data as promised.

Because an interface **contains no implementation**, programmers who use them **cannot rely on accidental implementation details**.

E.g., the `List` interface states that there must be an `add` method but does not say how it is implemented.

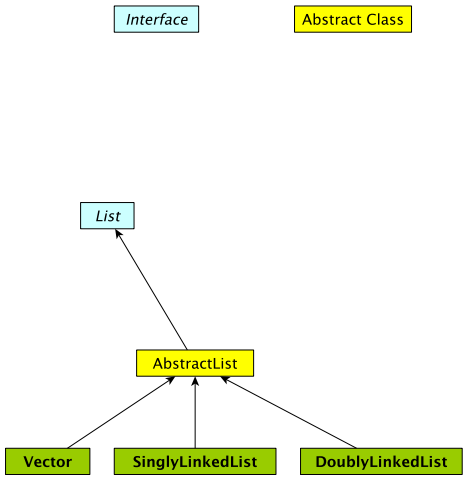
Abstract class

An **abstract class** is a partial implementation, mainly used as a **labor-saving device**.

E.g., many `List` implementations will implement methods the same way. Why duplicate all that work?

`isEmpty()` can always be implemented by checking that `size() == 0`.

structure5 List implementations



Missing from Java: ADT behavior

Java provides no way of specifying behavior independently of implementation.

E.g., a `List` interface might require

```
public void prepend(T elem)
```

But there's no way to **require** that the implementation actually place the element at the head of the list.

Next best thing: `assert` statements

This is why we encourage you to write pre- and post-conditions.

E.g.,

```
public void prepend(T elem) {
  T oldHead = head();
  …
  Assert.post(head().next() == oldHead)
}
```

Linear ADT

A **linear ADT** is one that presents elements **in a sequence**, even if the elements are **not actually stored that way**.

We will talk about two today: **stack** and **queue**.

Stack ADT

A **stack** is an **abstract data type** that stores a collection of **any type of element**. A stack **restricts which elements are accessible**: elements may only be added and removed from the "**top**" of the collection. The "**push**" operation places an element onto the top of the stack while a "**pop**" operation removes an element from the top.

Stack ADT

Stack ADT

Also sometimes referred to as a **LIFO**: **"last in, first out."**

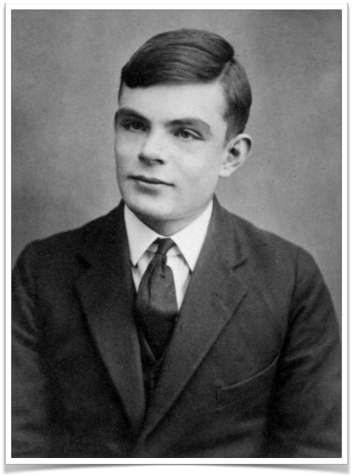
We also frequently include a "**peek**" operation that lets us look at an element on the top of a stack without removing it, and "**size**" and "**isEmpty**" operations that let us check how many elements are stored and whether a stack stores zero elements, respectively.

Stack ADT

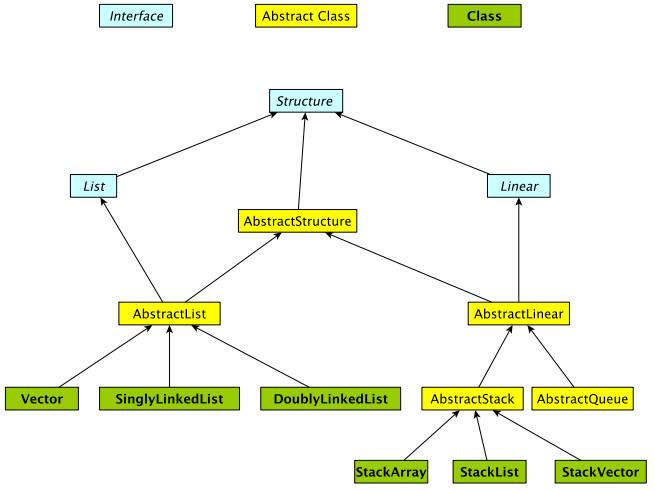
Interesting history: first appeared in print in a paper by Alan Turing (1946).

Unclear if he actually invented it.

push = **bury**,
pop = **unbury**.



structure5 Stack implementations



Application: Arithmetic

A computer can perform arithmetic using a stack.

E.g., 1 * 2 + 3 = 5

Small problem: order of ops in infix arithmetic depends on ops.

In postfix arithmetic, order is always the same: left to right

E.g., 1 2 * 3 +

Once in this form, processing is easy. (Example)

Activity: Arithmetic

Convert infix to postfix: `x*y+z*w`

1. Add parens to preserve order of operations:
   `((x*y)+(z*w))`
2. Move all operators to the end of each parenthesized expression:
   `((xy*)(zw*)+)`
3. Remove parens:
   `xy*zw*+`

Evaluate these using a stack:

1. `1 + 2 * 3`
2. `5 * (6 + 2) - 12 / 4`

Cool application: backtracking search

Stack implementations

StackArray

A StackArray is a stack implemented using an array for element storage.

Pros: push and pop are O(1) operations.

Cons: data structure has a maximum capacity.

Stack implementations

StackVector

A StackVector is a stack implemented using a Vector for element storage.

Pros: push and pop are amortized O(1) operations. There is no maximum capacity.

Cons: Most of the time, they take O(1) time, but occasionally--when the underlying array needs to grow--an O(n) cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice. Also, unless the underlying array is completely full, Vectors waste some space.

Stack implementations

StackList

A StackList is a stack implemented using a List (usu. SLL) for element storage.

Pros: push and pop are O(1) operations. There is no maximum capacity. Push and pop costs are predictable (always the same), unlike StackVector.

Cons: because of the way computer hardware is implemented, a StackList's constant-time cost is likely to be much higher than a StackVector's. So a StackList's performance may be more predictable than a StackVector, but it will likely be slower on average.

Queue ADT

A **queue** is an **abstract data type** that stores a collection of **any type of element**. A queue **restricts which elements are accessible**: elements may only be added to the "**end**" of the collection and elements may only be removed from the "**front**" of a collection. The "**enqueue**" operation places an element at the end of a queue while a "**dequeue**" operation removes an element from the front.

Queue ADT

Queue ADT

Also sometimes referred to as a **FIFO**: **"first in, first out."**

(a stack would be an annoying way to process a line at Starbucks!)

Frequently used as a **buffer** to hold work **to do later**.

We also frequently include a "**peek**" operation that lets us look at an element on the top of a queue without removing it, and "**size**" and "**isEmpty**" operations that let us check how many elements are stored and whether a queue stores zero elements, respectively.

Queue implementations

QueueArray

A QueueArray is a queue implemented using an array for element storage.

Pros: enqueue and dequeue are O(1) operations.

Cons: data structure has a maximum capacity.

Queue implementations

QueueVector

A QueueVector is a queue implemented using a Vector for element storage.

Pros: enqueue and dequeue are amortized O(1) operations. There is no maximum capacity.

Cons: Most of the time, they take O(1) time, but occasionally--when the underlying array needs to grow--an O(n) cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice. Also, unless the underlying array is completely full, Vectors waste some space.

Queue implementations

QueueList

A QueueList is a queue implemented using a List (usu. DLL or CL) for element storage.

Pros: enqueue and dequeue are O(1) operations. There is no maximum capacity. Enqueue and dequeue costs are predictable (always the same), unlike QueueVector.

Cons: because of the way computer hardware is implemented, a QueueList's constant-time cost is likely to be much higher than a QueueVector's. So a QueueList's performance may be more predictable than a QueueVector, but it will likely be slower on average.

Recap & Next Class

Today we learned:

- ADTs
- Linear ADTs
- Stack
- Queue

Next class:

- Iterators, etc.