# CSCI 136:
# Data Structures
# and
# Advanced Programming

## Lecture 17

## Linear structures

Instructor: Dan Barowy

Williams

# Topics

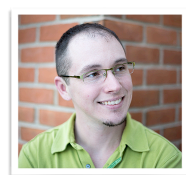
- Linear ADTs
- Stack ADT
- Queue ADT

# Your to-dos

1. Lab 6 (partner lab), **due Tuesday 4/12 by 10pm**.
2. Read **before Wed**: Bailey, Ch 10.

# Announcements

Colloquium on Friday.



**Friday, April 8 @ 2:35pm**

**Wege Hall – TCL 123**

**Perception and Context in Data Visualization**

**Jordan Crouser, Smith College**

Visual analytics is the science of combining interactive visual interfaces and information visualization techniques with automatic algorithms to support analytical reasoning through human-computer interaction. People use visual analytics tools and techniques to synthesize information and derive insight from massive, dynamic, ambiguous, and often conflicting data… and we exploit all kinds of perceptual tricks to do it! In this talk, we'll explore concepts in decision-making, human perception, and color theory as they apply to data-driven communication. Whether you're an aspiring data scientist or you're just curious about the mechanics of how data visualization works under the hood, stop by and take your pre-attentive processing for a spin.

# Announcements

Midterm exams back this week

# Practice Quiz

# 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.

## Linear ADT

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

In a linear ADT, **adding** and **removing** elements is **constrained**, meaning that the structure can only be modified according to certain rules.

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.”**

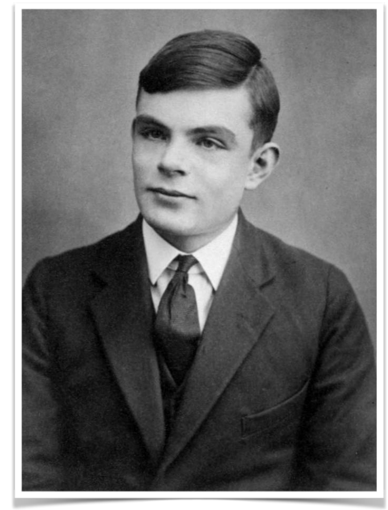
We also frequently include a "**peek**" operation that lets us look at an element on the top of a stack without removing it, and "**size**" and “**isEmpty**" operations that let us check how many elements are stored and whether a stack stores zero elements, respectively.

# Stack ADT

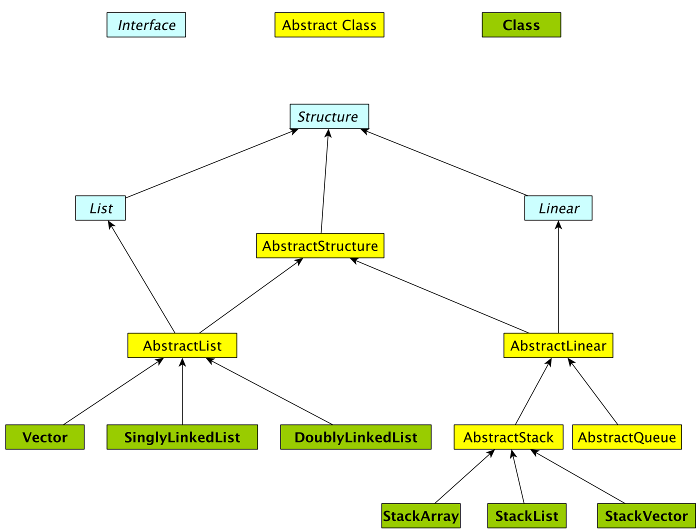
Interesting history: first appeared in print in a paper by Alan Turing (1946).

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 = 7

Small problem: order of operations in infix arithmetic depends on the operations themselves.

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

E.g., 1 2 3 * + (note: fixed the confusing class example)

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

# Activity: Arithmetic

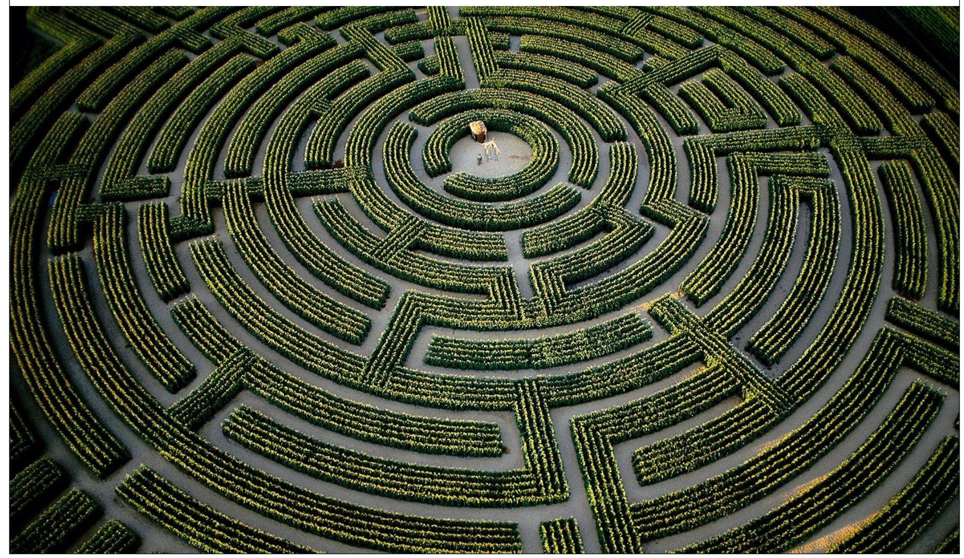
Convert infix to postfix: `x*y+z*w`

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*+`

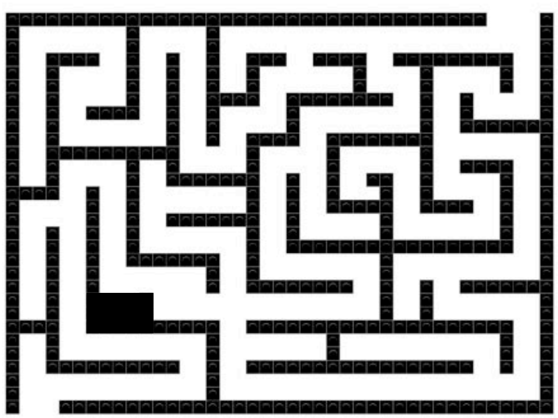
Evaluate these using a stack:

1. **`4 + 1 * 8`**
2. **`5 * (6 + 2) - 12 / 4`**

# Cool application: backtracking search



# Cool application: backtracking search



# Recap & Next Class

## Today:

Linear ADTs

Stack

## Next class:

Queue, etc.