

# COMP90038 Algorithms and Complexity

## Algorithms and Data Structures

Michael Kirley

Lecture 2

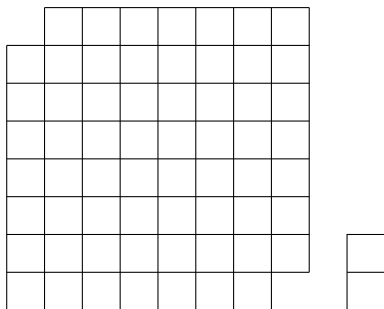
Semester 1, 2016

# Approaching a Problem

Can we cover this board with  
31 tiles of the form shown?

This is the **mutilated  
checkerboard problem**.

There are only finitely many  
ways we can arrange the 31  
tiles, so there is a brute-force  
(and very inefficient) way of  
solving the problem.



# Transform and Conquer? Use Abstraction?

Can we cover this board with  
31 tiles of the form shown?

Why can we quickly determine  
that the answer is no?

Hint: Using the way the  
squares are coloured helps.

# Algorithms and Data Structures

**Algorithms:** for solving problems, transforming data.

**Data structures:** for storing data; arranging data in a way that suits an algorithm.

- Linear data structures: Stacks and queues
- Trees and graphs
- Dictionaries

Which data structures are you familiar with?

# Exercise: Data Structures

Pick a data structure and describe:

- How to insert an item into the data structure
- How to find an item
- How to handle duplicate items

# Primitive Data Structures: The Array

An array consists of a sequence of consecutive cells in memory.

Depending on programming language:  $A[0]$  up to  $A[n-1]$ , or  $A[1]$  up to  $A[n]$ .

Locating a cell, and storing or retrieving data at that cell is very fast.

The downside of an array is that maintaining a **contiguous** bank of cells with information can be difficult and time-consuming.

# Primitive Data Structures: The Linked List

A collection of objects with links to one another, possibly in different parts of the computer's memory.

Often we use a dummy head node that points to the first object, or to a special `null` object that represents an empty list.

Inserting and deleting elements is very fast: just move a few links around.

Finding the  $i$ th element can be time-consuming.

# Iterative Processing

Walk through the array or linked list. For example, to locate an item.

```
j := 0
while j < last
  if A[j] == x
    return j
  j := j+1
return null
```

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next
return null
```



# Recursive Processing

Solve the problem on a smaller collection and use that solution to solve on the full collection.

```
function find(A,x,lo,hi)
  if lo > hi
    return null
  else if A[lo] == x
    return lo
  else
    return find(A,x,lo+1,hi)
```

Initial call: find(A,x,0,last)

We return to recursion in Week 3.

```
function find(p,x):
  if p == null
    return p
  else if p.val == x
    return p
  else
    return find(p.next,x)
```

Initial call: find(head,x)

# Abstract Data Types

A collection of data items, and a family of operations that operate on that data.

Think of an ADT as a set of promises, or contracts.

We must still **implement** these promises, but it is an advantage to separate the implementation of the ADT from the “concept”.

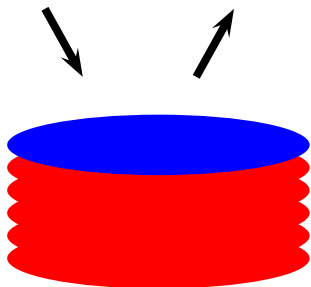
Good programming practice is to support this separation: Nothing outside of the definitions of the ADT should refer to anything inside, except through function calls for the basic operations.

# Fundamental Data Structures: The Stack

Last-in-first-out (LIFO).

Operations:

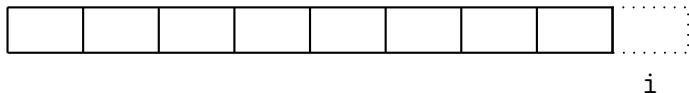
- CreateStack
- Push
- Pop
- Top
- EmptyStack?
- ...



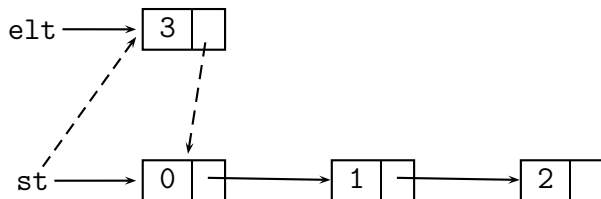
Usually implemented as an ADT.

# Stack Implementation

By array:



By linked list (push):

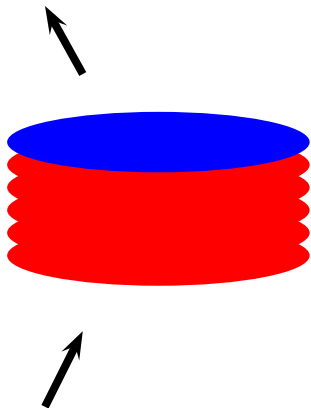


# Fundamental Data Structures: The Queue

First-in-first-out (FIFO).

Operations:

- CreateQueue
- Enqueue
- Dequeue
- Head
- EmptyQueue?
- ...



# Other Data Structures

We shall meet many other (abstract) data structures, such as

- The priority queue
- Various types of “tree”
- Various types of “graph”

If you check out Algorithms in Action, you will meet exotic data structures such as splay trees and skip lists.

# Next Up

Algorithm analysis - how to reason about an algorithm's resource consumption.