

# Algorithms and Analysis

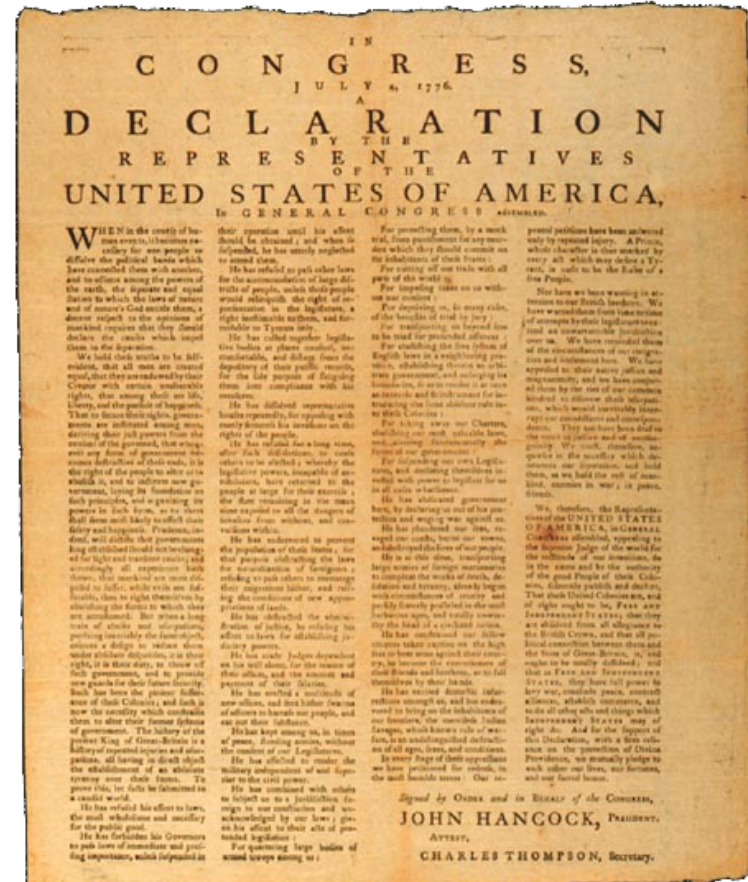
## Lesson 3: *Declare your intentions (not your actions)*



*ADTs, stacks, queues, priority queues, sets, maps*

# Outline

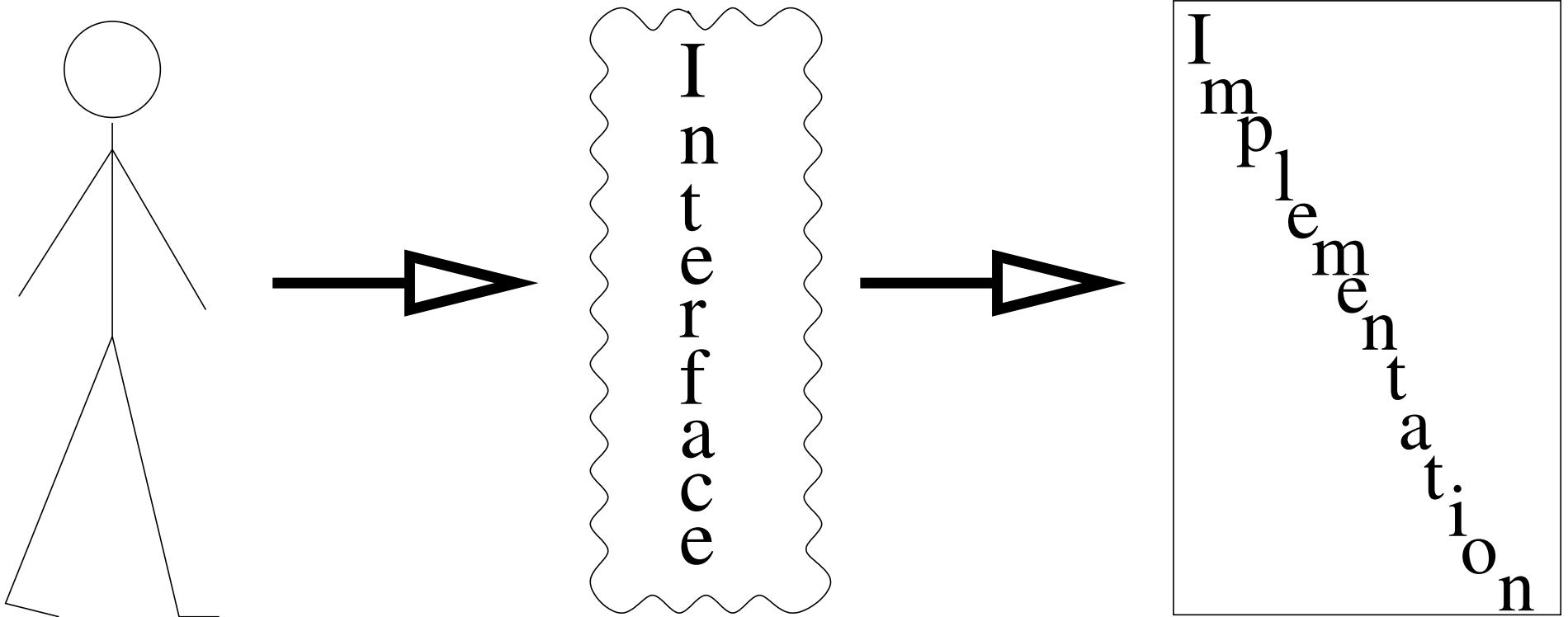
1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



# Object Oriented Programming

- OO-programming allows you to build large systems reliably■
- In the OO-methodology you separate the interface from the implementation■
- The **interface** is the public methods (functions) of a class■
- The implementation is hidden (**encapsulated**) and may be changed without affecting how the class is used■
- There exist other ways of programming, but C++ is designed to support the OO-methodology■—for building systems it is brilliant■

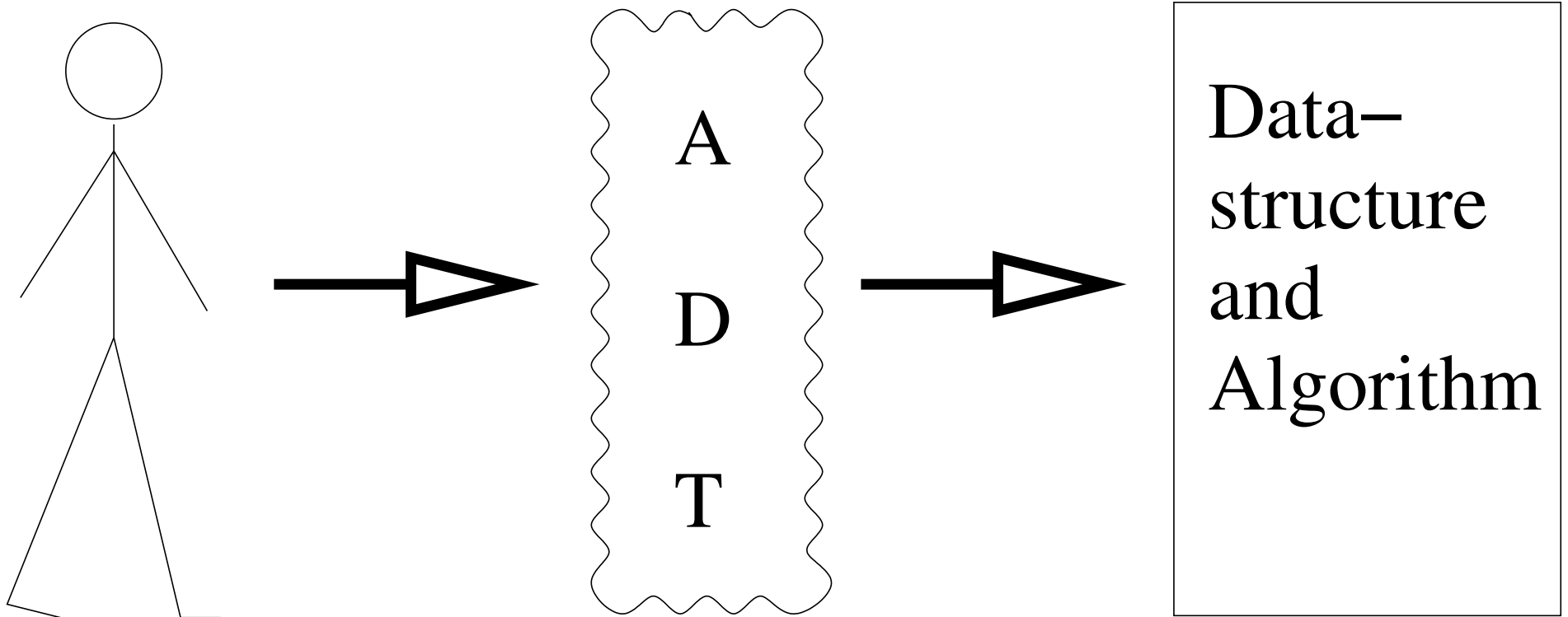
# Object-Oriented Classes



# Abstract Data Types

- With data structures there are some traditional interfaces called **Abstract Data Types** or ADTs■
- These are implementation free data structures■
- They are mathematical abstractions of the data structure■
- Their purpose is to allow you to declare you intentions■
- You are entering into an agreement that you only intend to use the underlying data structure in the way specified by the interface■

# ADTs



# Say it with an ADT

- Common ADTs include stacks, queues, priority queues, sets, multisets and maps■
- There are many possible implementations of these ADTs■(some far from obvious)■
- Each ADT has a limited set of methods associated with it■
- They are an abstraction away from the implementation■
- By declaring your intentions you are making your code easier to understand and maintain■

# Outline

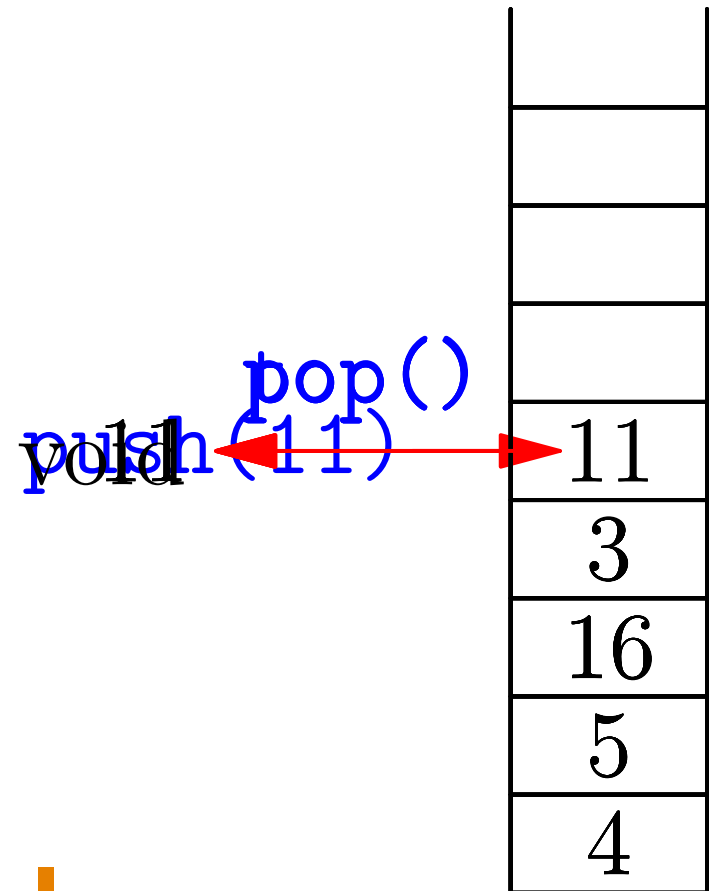
1. Abstract Data Types (ADTs)
2. **Stacks**
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together





# Stacks

- Last In First Out (LIFO) memory■
- Standard functions■
  - ★ `push(item)`■
  - ★ `T top()`■
  - ★ `T pop()`■ except in C++ `pop()` doesn't return the top of the stack■
  - ★ `boolean empty()`■
- Implemented using an array■ (or a linked-list)■



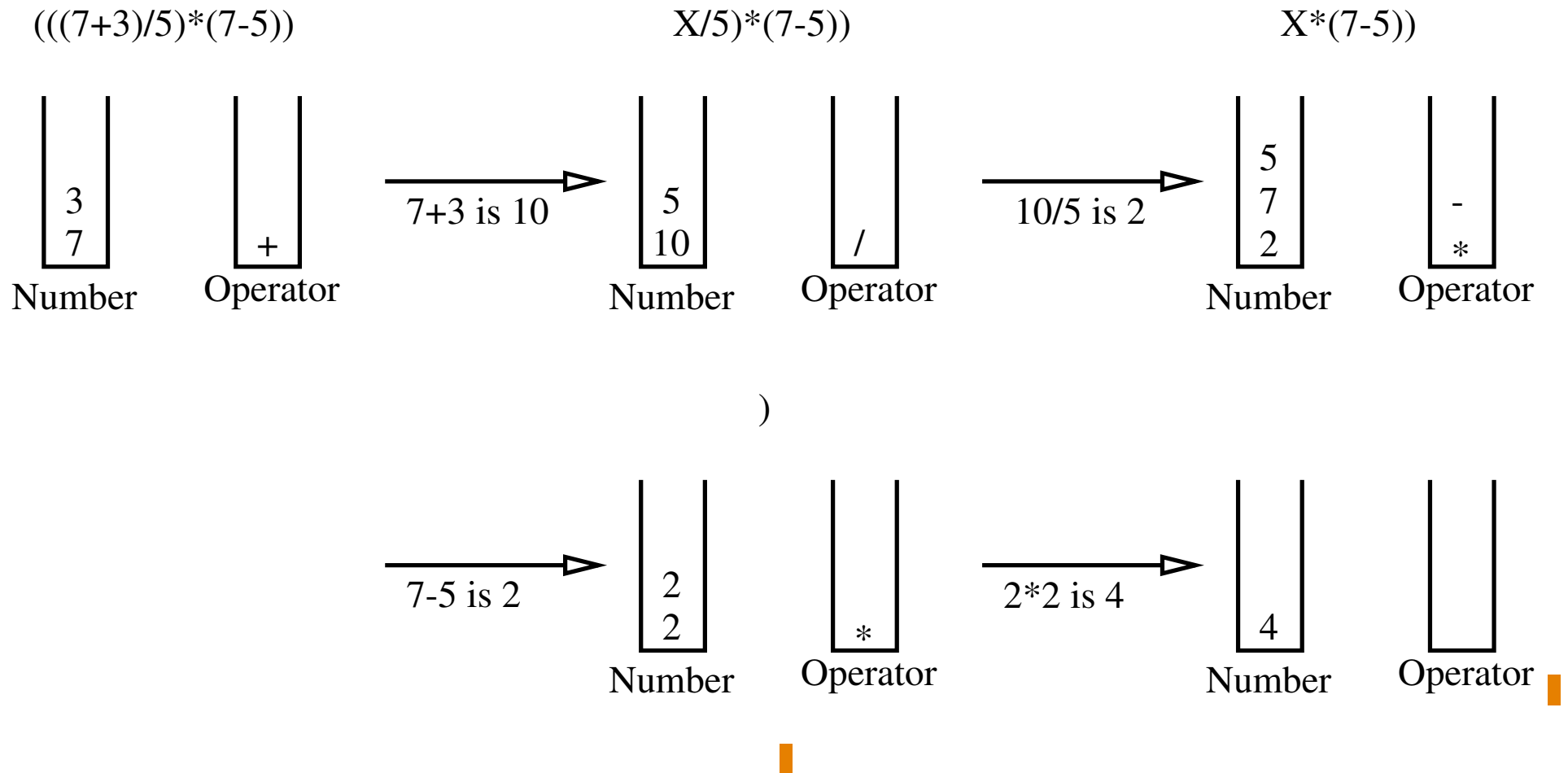
# Why Use a Stack?

- Stacks reduces the access to memory—no longer random access■
- Seems counter intuitive to reduce what you can do■
- Gives you a very simple interface■
- Prevents another programmer from using memory in a way that will break existing code■
- Sufficient for large number of algorithms■

# Uses of Stacks

- Reversing an array■
- Parsing expression for compilers■
  - ★ balancing parentheses■
  - ★ matching XML tags■
  - ★ evaluating arithmetic expression■
- Clustering algorithm■

# Evaluating Arithmetic Expressions



# Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



# Queues

- First-in-first-out (FIFO) memory model ■
- `enqueue(elem)` ■
- `peek()` ■
- `dequeue()` ■
- C++ has a double ended queue (`deque`) with `push_front()`, `push_back()`, etc. ■

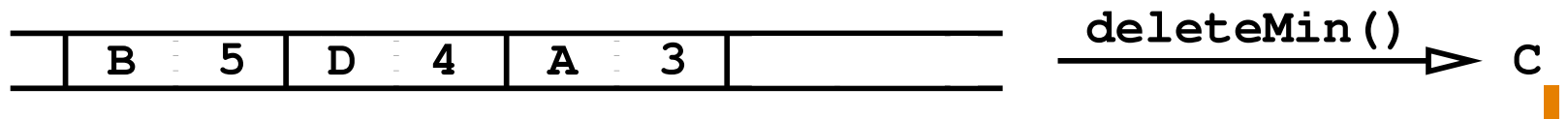


# Uses of Queues

- Queues are heavily used in multi-threaded applications (e.g. operating systems)■
- Multi-threaded applications need to minimise waiting and ensure the integrity of the data structure (for instance when an exception is thrown)■
- Because of this they are more complicated than most data structures■
- They can be implemented using linked-lists or circular arrays■

# Priority Queues

- Queue with priorities■
- `insert(elem, priority)` (in C++ `push()`)■
- `findMin()` (in C++ `top()`)■
- `deleteMin()` (in C++ `pop()`)■





# Uses of Priority Queues

- Queues with priorities (e.g. which threads should run)■
- Real time simulation■
- Often used in “greedy algorithms”
  - ★ Huffman encoding
  - ★ Prim’s minimum spanning tree algorithm■

# Implementation of Priority Queue

- Could be implemented using a binary tree or linked list■
- Most efficient implementation uses a heap■
- A heap is a binary tree implemented using an array■

# Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together



# Lists

- In C++ the standard list is known as `vector<T>`■
- That is, it is a collection where the order in which you put items into the list counts■
- You can have repetitions of elements■
- It has random access, e.g. `v[i]`■
- You can `push_back(i)`, `insert`, `erase`, etc.■
- C++ has a linked list class `List<T>`■

# Sets

- Models mathematical sets
- Container with no ordering or repetitions
- Methods include `insert`, `find`, `size`, `erase`
- Provides fast search (`find`)
- This is the class to use when you have to rapidly find whether an object is in the set or not—don't use a list like `vector<T>!`

# Iterators

- Wish to act on all members of the set■
- Performed using an `iterator`■
- Iterators are used by many collections■
- In C++ iterators follow the pointer convention■

```
set<string> words;
```

```
words.insert("hello");
```

```
words.insert("world");
```

```
for(auto iter = words.begin(); iter != words.end(); ++iter) {  
    cout << *iter << endl;  
}
```



# Implementation of Sets

- Sets are very important and there are many implementations depending on their usage■
- Two common implementations of sets are
  - ★ hash tables: `unordered_set<T>`
  - ★ binary trees: `set<T>` ■
- Which is most efficient depends on the application■
- Binary trees allow you to iterate in order■(iterating over a hash table will give you outputs in random order)■
- `multiset<T>` are sets with repetition■

# Maps

- A map provides a content addressable memory for pairs *keyword: data*■
- It provides fast access to the data through the keyword■
- Implement as tree or hash table■
- Multimaps allows different data to be stored with the same keyword■

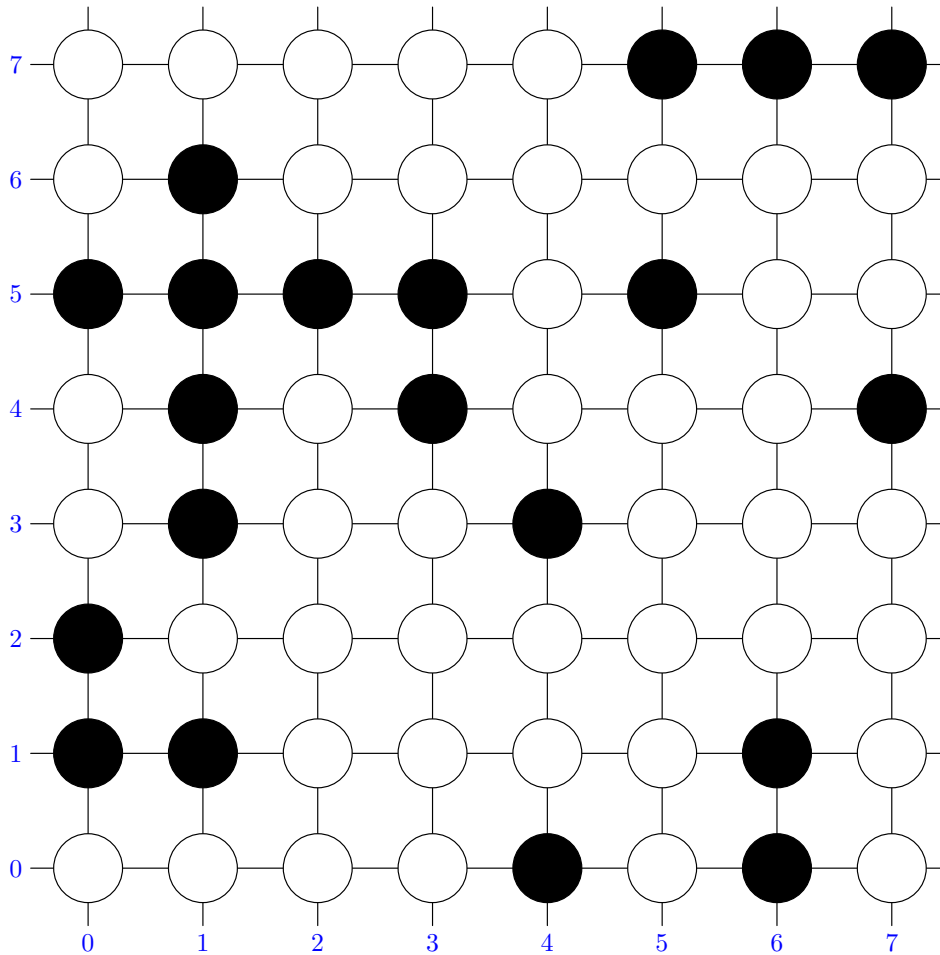


# Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together

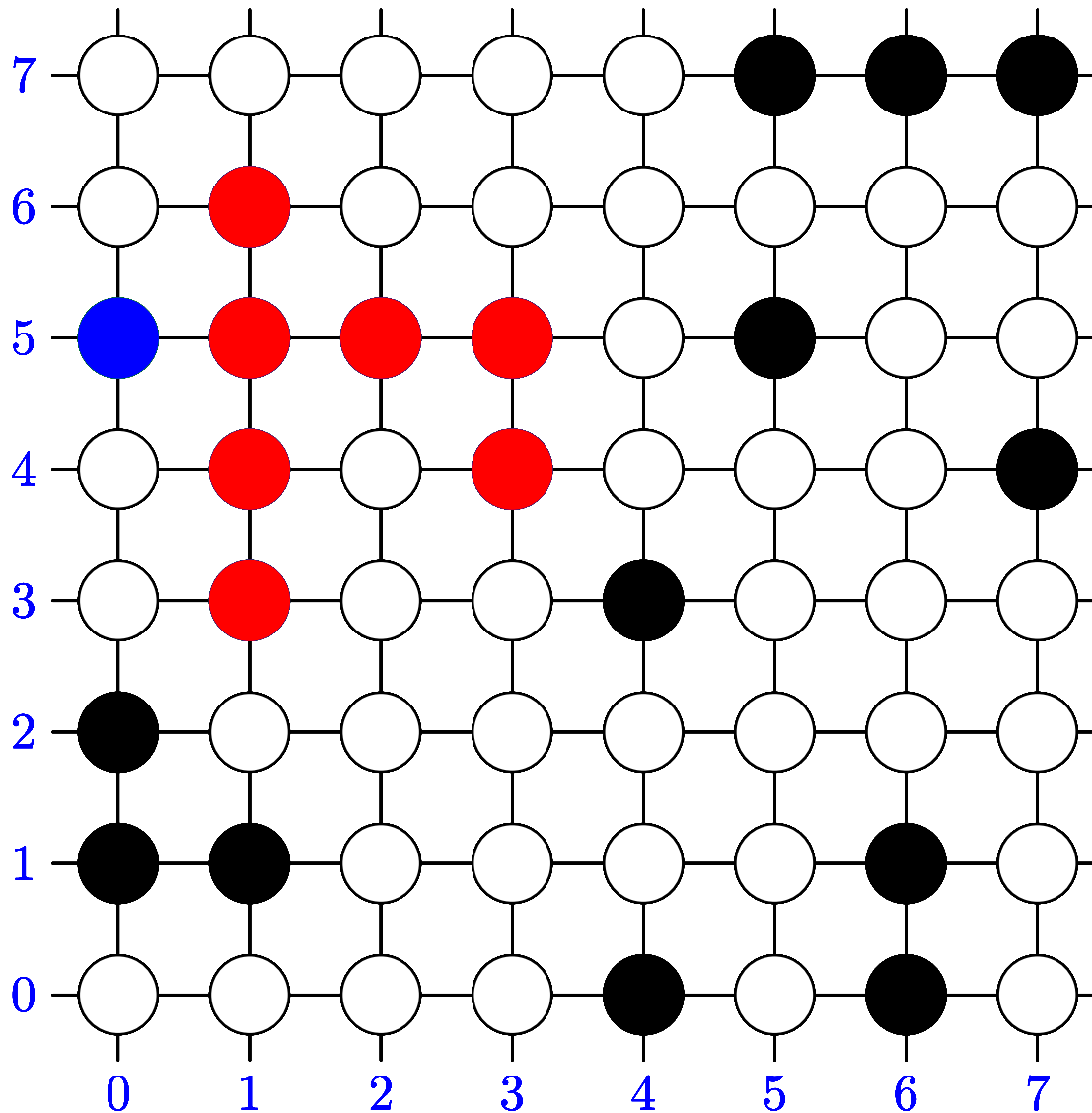


# Connected Nodes



- A frequent problem is to find clusters of connected cells
- Applications in computer vision, computer go, graph connectedness, . . .

# Connected Nodes



`startNode = (0, 4) (2, 5)`

`uncheckedNodes =`

`(1, 6)`  
`(3, 4)`  
`(0, 5)`

`clusterNodes =`  
`{(2, 5), (1, 5), (3, 5),`  
`(3, 4), (0, 5), (1, 4),`  
`(1, 6), (1, 3) }`

# Connected Node Algorithm

```
set<Node> findCluster(Node startNode, Graph graph)
{
    stack<Node> uncheckedNodes = new Stack<Node>();
    set<Node> clusterNodes = new HashSet<Node>();

    uncheckedNodes.push(startNode);
    clusterNodes.add(startNode);

    while (!uncheckedNodes.empty()) {
        Node next = uncheckedNodes.top();    uncheckedNodes.pop();
        vector<Node> neighbours = graph.getNeighbours(next);
        for (Node neigh: neighbours) {
            if (graph.isOccupied(neigh) && !clusterNodes.contains(neigh) ) {
                uncheckedNodes.push(neigh);
                clusterNodes.insert(neigh);
            }
        }
    }

    return clusterNodes;
}
```

# Lessons

- Abstract Data Types (ADT) are interfaces to data■
- Their purpose is to allow the programmer to declare their intentions■
- They often have different implementations with different properties■
- The most efficient implementation is not always obvious■—we will see many of these implementations as we go through this course■
- You need to know the common ADTs (e.g. Stack, Queue, List, Set, Map) and how and when to use them■