Data types and data structures

A data type (also called abstract data type or ADT) defines the operations and behavior supported by an object.

A data type is a concept, similar to mathematical concepts such as function, set, or sequence.

Examples of data types are Stack, Queue, Set, Dictionary.

A data structure is an implementation of a data type: An object that provides all the operations defined by the data type, with the correct behavior.

We often have multiple, different implementations for the same data type: Stacks can be implemented with arrays or with linked lists, sets can be implemented with search trees or with hash tables.

Traits as arguments

A trait can be used as a type for a variable:

```
def reverse(stack: Stack[Char], s: String) {
    for (ch <- s)
      stack.push(ch)
    while (!stack.isEmpty)
      print(stack.pop())
  }
```

The function `reverse` can be called with any object that implements the `Stack[Char]` trait.

```
reverse(new FixedStack[Char], "Hello")
reverse(new LinkedStack[Char], "Hello")
reverse(new scala.collection.mutable.Stack[Char]
        with Stack[Char], "Hello")
```


Traits

In Scala, we can represent a data type using a trait. ("Trait" means "personality".)

```
trait Stack[T] {
  def push(el: T):
    Stack[T]
  def top: T
  def pop(): T
  def isEmpty: Boolean
}
```

```
trait Queue[T] {
  def clear(): Unit
  def isEmpty: Boolean
  def head: T
  def dequeue(): T
  def enqueue(els: T*):
     Unit
}
```

A trait defines the methods that a data type provides, together with the types of the arguments and the result type. A trait can have type parameters (like `T` here).

Traits are called interface or abstract class in other languages.

Implementing a trait

To implement a trait, we create a `class` that provides all the methods of the trait, with the correct types.

Normally, the class would indicate that it implements the trait using the `extends` keyword:

```
class FixedStack[T : ClassTag] extends Stack[T] {
```

`extends` means that where ever a `Stack` trait object is needed, a `FixedStack` object can be used:

```
reverse(new FixedStack[Char], "CS206")
```

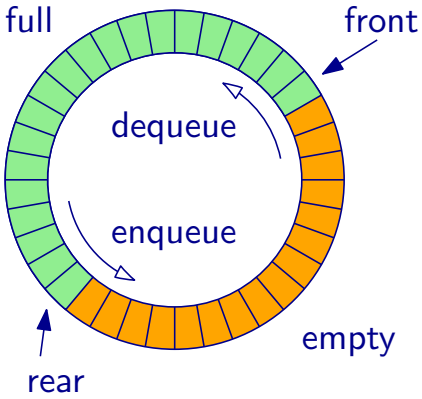
```
val s: Stack[Int] = new FixedStack[Int]
```

Scala allows to use an object as an implementation even if it does not use `extends`, using the `with` keyword. This is not possible in Java or C++.

Circular buffers

Implementing a stack using an array is easy—just keep a counter to remember the top of the stack inside the array.

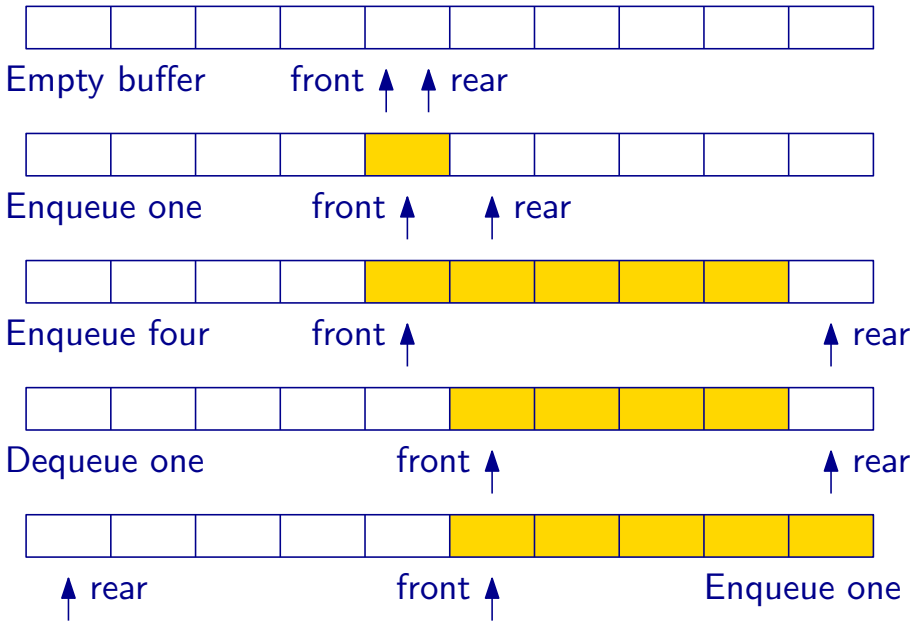
Implementing a queue is harder, because we insert and remove elements at two different places. The normal way to do this is using a circular buffer.



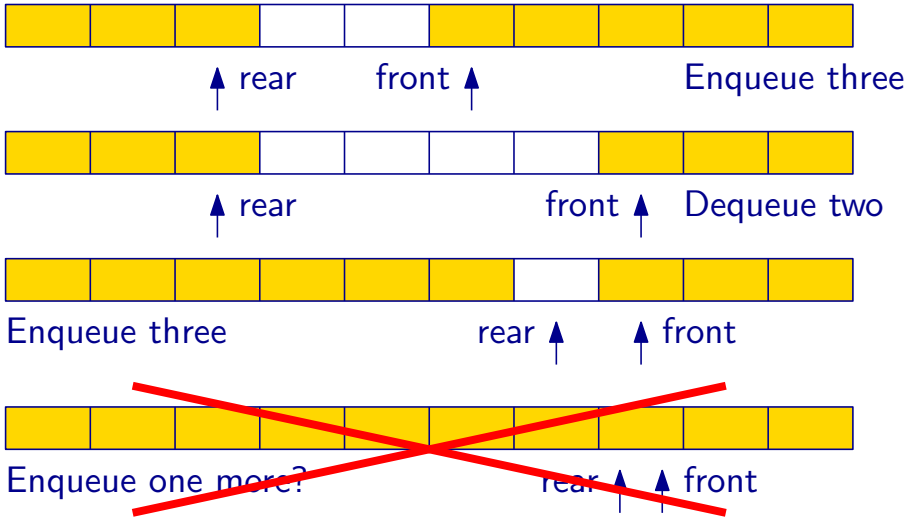
Since we do not have circular memory, we have to simulate that using a linear array, and two index pointers.

When the index reaches the end, it "wraps around" to the beginning.

Circular buffers



Circular buffers



What does `front == rear` mean? Full buffer or empty buffer?

Typical solution: Forbid filling buffer completely, always keep one slot free. (See Circular Buffers on Wikipedia for other solutions.)

A stack without arrays

Using an array to store the stack elements means we either need to know the maximum stack size in advance, or we sometimes need to (slowly) copy elements into a new array.

We can avoid this and achieve `push` and `pop` operations that are always fast, if we use a small `Node` object to hold each stack element.

The `LinkedStack` holds a reference to the `Node` at the top of the stack.
Each `Node` holds a reference to the one directly below in the stack.