package. The `EliminationBackoffStack` we present here is modular, making use of exchangers, but somewhat inefficient. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit presented a highly effective implementation of an `EliminationArray` [131].

11.6 Exercises

Exercise 11.1. Design an unbounded lock-based `Stack<T>` implementation based on a linked list.

Exercise 11.2. Design a bounded lock-based `Stack<T>` using an array.

1. Use a single lock and a bounded array.
2. Try to make your algorithm lock-free. Where do you run into difficulty?

Exercise 11.3. Modify the unbounded lock-free stack of Section 11.2 to work in the absence of a garbage collector. Create a thread-local pool of preallocated nodes and recycle them. To avoid the ABA problem, consider using the `AtomicStampedReference<T>` class from java.util.concurrent.atomic (see Pragma 10.6.1), which encapsulates both a reference and an integer *stamp*.

Exercise 11.4. Discuss the back-off policies used in our implementation. Does it make sense to use the same shared `Backoff` object for both pushes and pops in our `LockFreeStack<T>` object? How else could we structure the back-off in space and time in the `EliminationBackoffStack<T>`?

Exercise 11.5. Implement a stack algorithm assuming there is a known bound on the difference between the total number of successful pushes and pops to the stack in any state of the execution.

Exercise 11.6. Consider the problem of implementing a bounded stack using an array indexed by a `top` counter, initially zero. In the absence of concurrency, these methods are almost trivial. To push an item, increment `top` to reserve an array entry, and then store the item at that index. To pop an item, decrement `top`, and return the item at the previous `top` index.

Clearly, this strategy does not work for concurrent implementations, because one cannot make atomic changes to multiple memory locations. A single synchronization operation can either increment or decrement the `top` counter, but not both, and there is no way atomically to increment the counter and store a value.

Nevertheless, Bob D. Hacker decides to solve this problem. He decides to adapt the dual data structure approach of Chapter 10 to implement a *dual* stack. His `DualStack<T>` class splits `push()` and `pop()` methods into *reservation* and *fulfillment* steps. Bob's implementation appears in Fig. 11.10.

The stack's top is indexed by the `top` field, an `AtomicInteger` manipulated only by `getAndIncrement()` and `getAndDecrement()` calls. Bob's `push()` method's reservation step reserves a slot by applying `getAndIncrement()` to `top`. Suppose the call returns

```
public class DualStack<T> {
  private class Slot {
    boolean full = false;
    volatile T value = null;
  }
  Slot[] stack;
  int capacity;
  private AtomicInteger top = new AtomicInteger(0); // array index
  public DualStack(int myCapacity) {
    capacity = myCapacity;
    stack = (Slot[]) new Object[capacity];
    for (int i = 0; i < capacity; i++) {
      stack[i] = new Slot();
    }
  }
  public void push(T value) throws FullException {
    while (true) {
      int i = top.getAndIncrement();
      if (i > capacity - 1) { // is stack full?
        top.getAndDecrement(); // restore index
        throw new FullException();
      } else if (i >= 0) { // i in range, slot reserved
        stack[i].value = value;
        stack[i].full = true; // push fulfilled
        return;
      }
    }
  }
  public T pop() throws EmptyException {
    while (true) {
      int i = top.getAndDecrement();
      if (i < 0) { // is stack empty?
        top.getAndDecrement() // restore index
        throw new EmptyException();
      } else if (i <= capacity - 1) {
        while (!stack[i].full){};
        T value = stack[i].value;
        stack[i].full = false;
        return value; // pop fulfilled
      }
    }
  }
}
```

FIGURE 11.10

Bob's problematic dual stack.

index i. If i is in the range $0\ldots$`capacity` $-$ 1, the reservation is complete. In the fulfillment phase, `push`(x) stores x at index i in the array, and raises the `full` flag to indicate that the value is ready to be read. The `value` field must be **`volatile`** to guarantee that once `flag` is raised, the value has already been written to index i of the array.

If the index returned from `push`()'s `getAndIncrement`() is less than 0, the `push`() method repeatedly retries `getAndIncrement`() until it returns an index greater than or equal to 0. The index could be less than 0 due to `getAndDecrement`() calls of failed `pop`() calls to an empty stack. Each such failed `getAndDecrement`() decrements the `top` by one more past the 0 array bound. If the index returned is greater than `capacity`$-$1, `push`() throws an exception because the stack is full.

The situation is symmetric for `pop`(). It checks that the index is within the bounds and removes an item by applying `getAndDecrement`() to `top`, returning index i. If i is in the range $0\ldots$`capacity` $-$ 1, the reservation is complete. For the fulfillment phase, `pop`() spins on the `full` flag of array slot i, until it detects that the flag is true, indicating that the `push`() call is successful.

What is wrong with Bob's algorithm? Is this problem inherent or can you think of a way to fix it?

Exercise 11.7. Exercise 8.7 asks you to implement the `Rooms` interface, reproduced in Fig. 11.11. The `Rooms` class manages a collection of *rooms*, indexed from 0 to m (where m is a known constant). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. The last thread to leave a room triggers an `onEmpty`() handler, which runs while all rooms are empty.

Fig. 11.12 shows an incorrect concurrent stack implementation.

1. Explain why this stack implementation does not work.
2. Fix it by adding calls to a two-room `Rooms` class: one room for pushing and one for popping.

Exercise 11.8. This exercise is a follow-on to Exercise 11.7. Instead of having the `push`() method throw `FullException`, exploit the push room's exit handler to resize the

```java
public interface Rooms {
  public interface Handler {
    void onEmpty();
  }
  void enter(int i);
  boolean exit();
  public void setExitHandler(int i, Rooms.Handler h) ;
}
```

FIGURE 11.11

The `Rooms` interface.

```
public class Stack<T> {
  private AtomicInteger top;
  private T[] items;
  public Stack(int capacity) {
    top = new AtomicInteger();
    items = (T[]) new Object[capacity];
  }
  public void push(T x) throws FullException {
    int i = top.getAndIncrement();
    if (i >= items.length) { // stack is full
      top.getAndDecrement(); // restore state
      throw new FullException();
    }
    items[i] = x;
  }
  public T pop() throws EmptyException {
    int i = top.getAndDecrement() - 1;
    if (i < 0) {           // stack is empty
      top.getAndIncrement(); // restore state
      throw new EmptyException();
    }
    return items[i];
  }
}
```

FIGURE 11.12

Unsynchronized concurrent stack.

array. Remember that no thread can be in any room when an exit handler is running, so (of course) only one exit handler can run at a time.