   Helgrind is a piece of software designed to detect synchronization problems in c/c++

Concurrent Objects
   We need to be able to add a thread and keep program robustness and not ruin throughput

   Queue locks - Define a priority and waits until it is up before it requests a lock. This prevents
   multiple threads from busy waiting

   If the API for the data structure be sequential then you should NOT get speed up for applying
   parallelization

Ways to parallelize a Structure:
   • Synchronizing everything
     • Leads to a bottleneck in code
     • Adding more threads makes it difficult to maintain performance because of Amdahls law
       and the proportion of sequential code
   • Smart locking
     • For example, having a lock on each element of a tree and unlocking the left side if
       traversing the right side. This way, you're only locking the minimum possible
     • Instead of using a single lock, we should split object into independently synchronized
       components this way accessing one part of an object will not affect access to the other part.
   • Do work without locking and check the validity (during a interval) to ensure that the result is
     still correct before returning
   • Postpone hard work
     • Set bit (in a tree for example) to indicate work to be done. So if you needed to do a removal,
       you would only do it if you got to that point in the tree. This could optimize out changes that
       would never have been read
   • No Locks
     • Much more complicated example.
     • Locks can be replaced with compareAndSet() atomic functions in order to change data

Each method locks the object
   uses queue locks

Set interface Example:

List of nodes;
   each has type T
   has int key;
   has node next;
The Invariants are the properties that must hold during a given action

Most steps are trivial

The invariants are often only necessary in one particular part of the code. It is ok for the invariants to be false while surrounding this part of the code but the MUST be true for the one critical section

Encapsulation helps with concurrency

Concrete representation:
   Required:
      Sorted?
      Duplicates?

   Suppose:
      Add() can add duplicate copies
      remove() will only remove one element
   If Invariant says not duplicates:
      Add() is wrong
   else:
      remove is wrong

   Course grain locking: easy but creates bottlenecks
   Fine grain locking: Much more difficult but lessens bottlenecks
      Hand over Hand locking: grabbing second lock immediately before releasing previous lock