100 points total.

1 Problems (56 points)

1.1 Set (1) [8 points]

In listing 3, two entries are locked before key presence is determined. If no entries were locked and it instead returned `true`/`false` based on key existence, would this alternative still be linearizable? If so, explain; if not, give a counterexample.

1.2 Set (2) [8 points]

Will listing 1, by itself, function correctly if we switch the locking order on line 9? What about in context with listings 2 and 3? If it has issues in context, how can that be fixed?

1.3 Set (3) [8 points]

Show that listing 1 only needs to lock `pred`.

1.4 Lock-free Linked List (1) [8 points]

Listings 4 and 5 show the `add()` and `remove()` methods for LockFreeList class. Describe the two execution scenarios where the `compareAndSet()` operation in line `11` of `add()` method fails because of:

1. expected reference (1st argument)
2. marked value (3rd argument)

1.5 Lock-free Linked List (2) [8 points]

Listings 4 and 5 show the `add()` and `remove()` methods for LockFreeList class. Describe the two execution scenarios where the `compareAndSet()` operation in line `11` of `remove()` method fails because of:

1. expected reference (1st argument)
2. marked value (3rd argument)

```java
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) {
          return false;
        } else {
          Node node = new Node(item);
          node.next = curr;
          pred.next = node;
          return true;
        }
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

Listing 1: Add item

```java
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        pred.next = curr.next;
        return true;
      } else {
        return false;
      }
    }
  } finally {
    pred.unlock(); curr.unlock();
  }
}
```

Listing 2: Remove item

```java
public boolean contains(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        return curr.key == key;
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

Listing 3: Containment check

```java
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
      return false;
      } else {
        Node node = new Node(item);
        node.next = new AtomicMarkableReference(curr, false);
        if (pred.next.compareAndSet(curr, node, false, false)) {
          return true;
        }
      }
   }
}
```

Listing 4: LockFreeList add()

```java
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip;
  while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key != key) {
      return false;
    } else {
      Node succ = curr.next.getReference();
      snip = curr.next.compareAndSet(succ, succ,
          false, true);
      if (!snip)
        continue;
      pred.next.compareAndSet(curr, succ, false,
          false);
      return true;
    }
  }
}
```

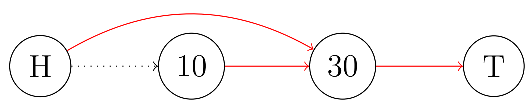
Listing 5: LockFreeList remove()

1.6 Lock-free Linked List (3) [8 points]

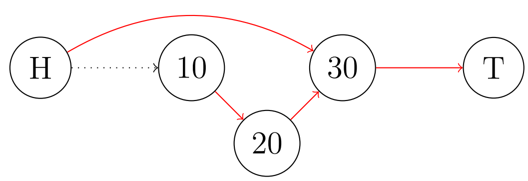
A programmer notices that the `add()` method of the `LockFreeList` (Listing 4) never finds a marked node with the same key. Therefore, to save the need to insert a new node, he/she modifies algorithm to simply insert new added object into the existing marked node with same key if such a node exists in the list. Explain why will this not work.

1.7 Compare-And-Swap [8 points]

Some naïve approaches to lock-free deletion in linked lists use a single CAS, swapping the element to be deleted with its next node as shown below.



Unfortunately, this can lead to the following situation:



Why can this happen? Why is it a problem?

2 Programming (44 points)

In this implementation, you will write a microbenchmark to test each of the linked-list algorithms for sets described in chapter 9 of the textbook; for the locking algorithms, replace usage of explicit locks with **`synchronized`** blocks that provide the same semantics as the explicit lock usage (this will not be possible in all cases). Don't forget to add **`volatile`** to those variables that need it (but keep usage to a minimum as otherwise performance may suffer). Note that there may be errors in the code that you will need to correct as well.

Use sets of integers `Set<Integer>` as the implementation to benchmark. A worker thread does an operation `ITER` times. The operation on the set could be `add`, `remove` or `contains` with a number passed as a parameter.

`ITER = 1000` and operating with numbers (a random choice) from `0` to `100` is a good choice for a worker thread as it ensures some degree of contention.

You will need to vary workload parameters (e.g. ratio of `contains`/`add` /`remove` operations) and find trends in behavior of all algorithms. Vary percentage of `contains` operation from 20% to 80% with the following spacing - 20%, 40%, 60% and 80%. Divide rest of the operations between `add` and `remove` equally wherever possible. Benchmark from 4 threads to 40 threads with reasonable spacing and at least for 8 different thread counts (Use Rlogin).

There are two ways for approaching mixed workloads. Choose whichever you feel comfortable with. `(1)` Have worker threads that only perform one operation. For example, for 20 threads, you can have 12 threads that always invoke `contains`, and 4 threads each that always invoke `add` and `remove`. Approximate wherever needed. `(2)` Each thread chooses between `contains` , `add` and `remove` through a uniformly distributed float random number generator.

Hint: Read about `java.util.concurrent.ThreadLocalRandom`. Use it in the `run()` method of your thread.

Don't forget proper benchmarking practices (accounting for JVM warm-up - the more the better, no points of contention beyond the set object itself, etc.); you do not need to use JMH, but you may find it useful to avoid the pitfalls discussed in `http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html`.

Provide the following plots:

1. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 20; all algorithms.

2. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 40; all algorithms.

3. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 60; all algorithms.

4. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 80; all algorithms.

5. *throughput vs. contains %*; contains = 20%, 40%, 60%, 80% for fixed thread count = 20; all algorithms.

Analyze your findings in a writeup, which should not exceed two pages in length.

Your code and writeup (with filename `hw5_<myPID>.pdf`) should be included in a zip file named `hw5_<myPID>.zip` to be uploaded to Canvas by the announced due date; as with homework 1, your code should be part of a `hw5` package and should be runnable from the command line (document the exact command(s) to use).