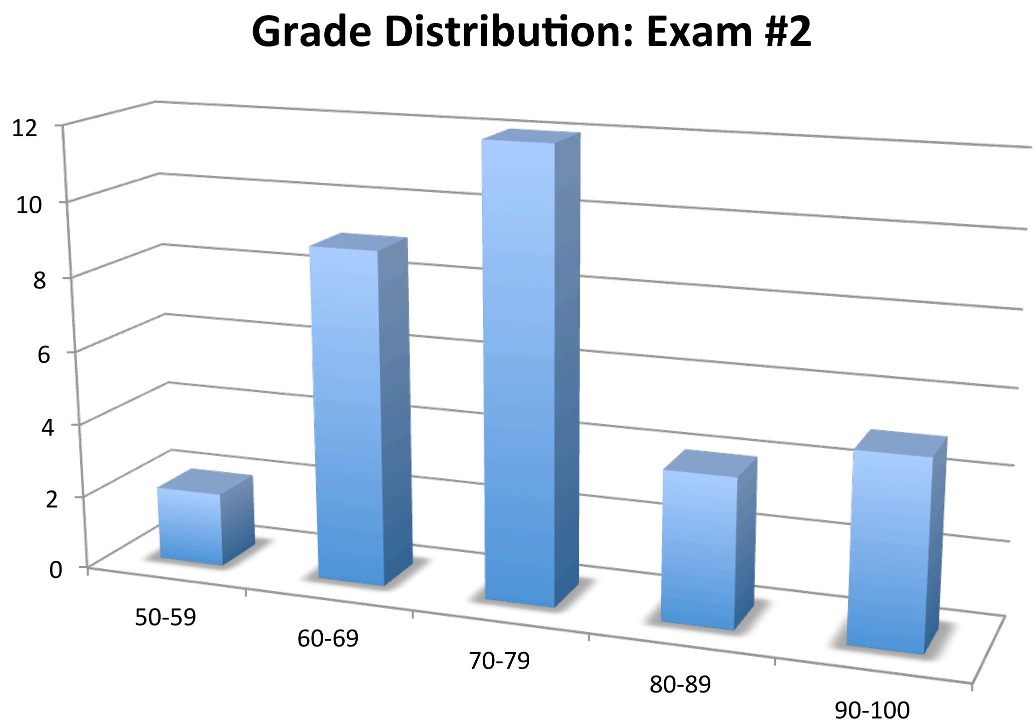
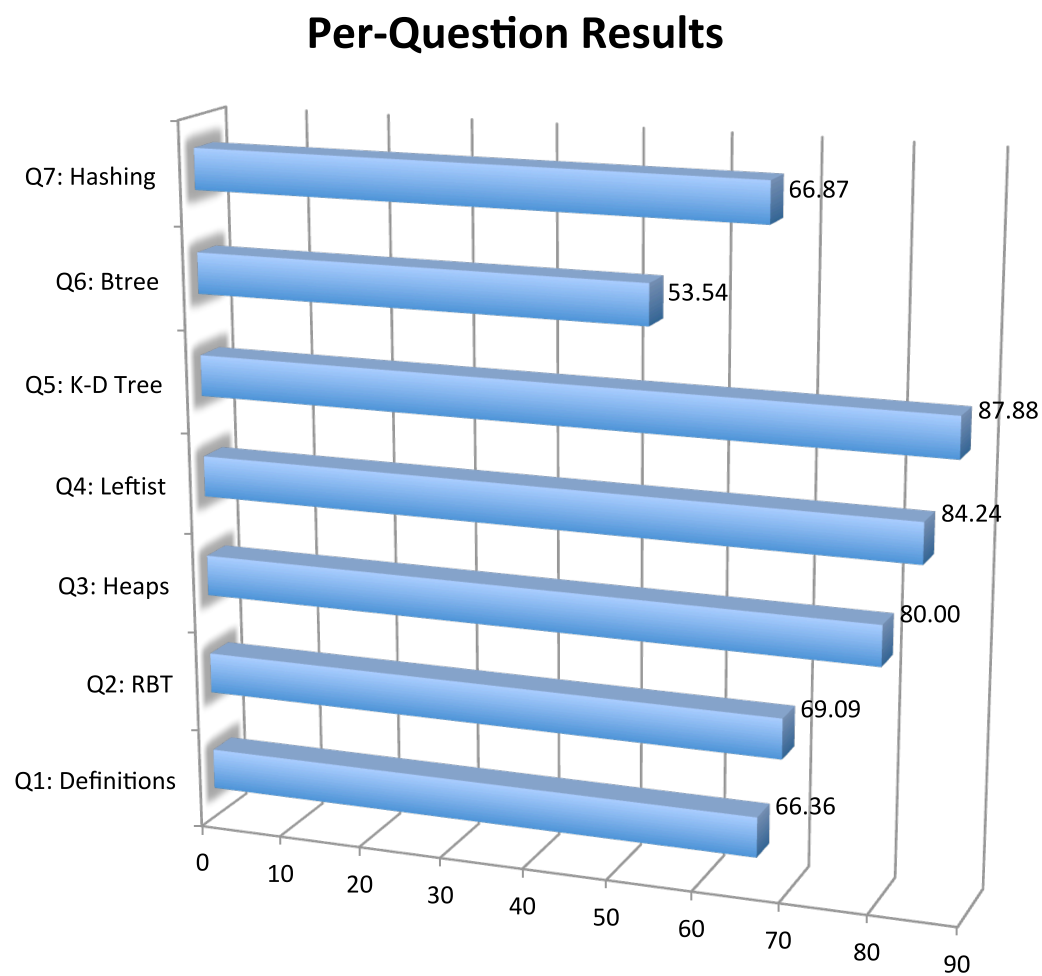
13	14 Exam 2	15 PROJ: Project 4 Ou	16 Threads	17	18	19
20	21 Threads	22	23 Disjoint Sets HW: HW 6 Out	24	25	26
27	28 Graphs	29 PROJ: Project 4 Du PROJ: Project 5 Ou	30 HW: Hw 6 Due	May 1	2	3
4	5 Graphs	6	7	8	9	10
11	12 TBD	13 PROJ: Project 5 Du	14 EMAIL GRACE DAYS EMAIL GRACE DAYS NO CLASS	15	16	17
18	19	20	21 10:30am CMSC 341,	22	23	24

Exam #2



Mean = 72.88%
Max = 94%

Per-Question Distribution



Question 1: General Questions (10 points)

1. What does it mean to say a B-Tree is order M?

2. When describing a B-Tree, what does L represent?

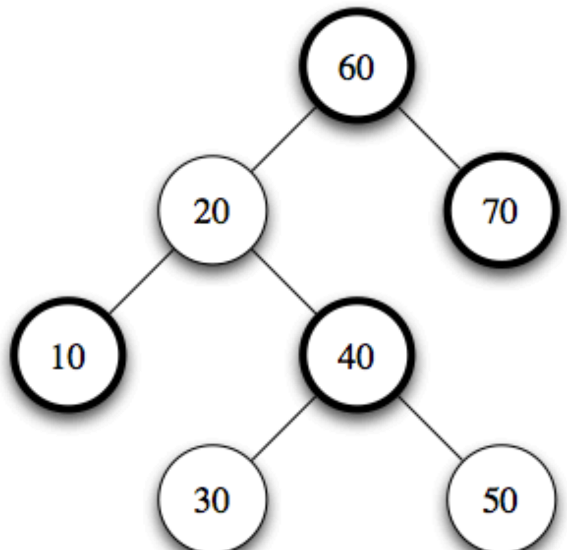
3. When describing RBTs, what is the *black height* of a node, x?

4. When describing Leftist trees, what is the *null path length*?

5. What is the minimum and maximum number of leaves in a B-Tree of height h = 2 when M = 3?

Question 2: Red-Black Trees (15 points)

Show the result of inserting the value 25 into the Red-Black Tree below. Nodes with thick outlines are black, all other nodes are red. For maximum partial credit show intermediate steps.



Question 3: Priority Queues (15 points)

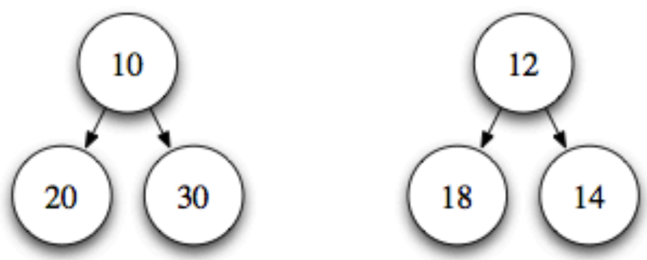
Priority Queues: Suppose a min heap represented as an array contains the following values (starting at array index one): 4, 6, 5, 7, 10, 6. Show the heap, either as an array or tree, after performing each of the following operations:

- **`insert(3)`**
- **`insert(5)`**
- **`deleteMin()`**

The operations are cumulative. That is, the second **`insert`** operates on the heap that results from the first, and the **`deleteMin`** operates on the heap that results from the second **`insert`**.

Question 4: Leftist Trees (15 points)

Leftist Heaps: Show the result of merging the two leftist heaps below. For maximum partial credit, show intermediate results.



Question 5: K-D Trees (15 points)

K-D Trees: Show the tree that results from inserting the following triples into an initially empty 3-D tree: (24, 22, 21), (20, 7, 5), (16, 17, 16), (23, 3, 28), (5, 10, 27), (10, 17, 5), (4, 19, 6), (18, 28, 23), (7, 35, 11), (4, 26, 23).

Question 6 - BTrees: (15 points)

Start with an initially empty BTree for which $L = 3$ and $M = 3$, show the BTree that results from inserting the following integers: 3, 28, 5, 10, 27, 11, 17, 6, 4, 19.

Question 7: **Hashing** (15 points)

Suppose the following numbers are inserted into a hash table of size 10 in the given order: 71, 23, 73, 99, 44, 79, 89, 31, 43. The hash function used is $h(k) = k \bmod 10$.

(A; 5 points) Show the resulting separate chaining hash table.

(B; 7 points) Show the resulting open addressing hash table using quadratic probing, $f(I) = I^2$.

(C; 3 points) What's the *load factor* of the hash table in problem **B**?

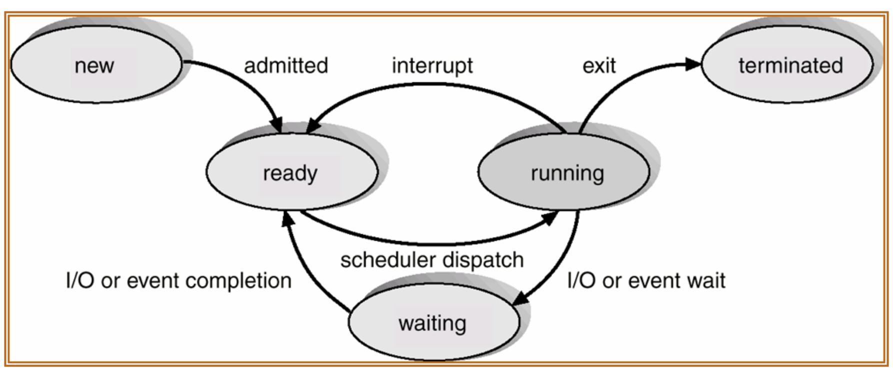
Parallel Programming

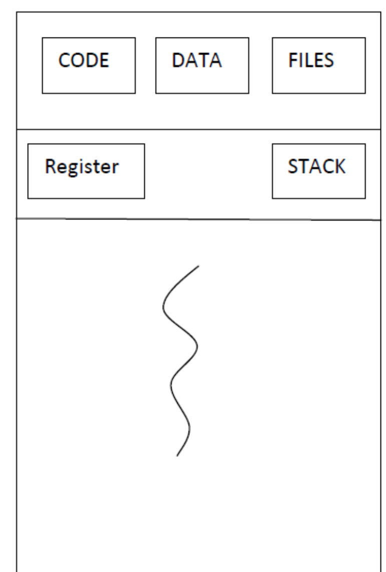
A brief intro to:

Parallelism, Threads, and Concurrency

(Multi)Process vs (Multi)Thread

- Assume a computer has one CPU
- Can only execute one statement at a time
  - Thus one program at a time
- Process: an operating-system level "unit of execution"





Single threaded process

Task

- A **task** is an abstraction of a series of steps
  - Might be done in a separate thread
- In Java, there are a number of classes / interfaces that basically correspond to this
  - Example: Runnable
    - work done by method run()

Java’s Thread Class

- To use Thread class directly (not recommended now):
  1. define a subclass of Thread and override run() – not recommended!
  2. Create a task as a Runnable, link it with a Thread, and then call start() on the Thread.
     - The Thread will run the Runnable’s run() method.

Creating a Task and Thread

- Get a thread object, then call start() on that object
  - Makes it available to be run
  - When it's time to run it, Thread's run() is called
- So, create a thread using inheritance
  - Write class that extends Thread, e.g. MyThread
  - Define your own run()
  - Create a MyThread object and call start() on it
- We won't do this! Not good design!

Java's Thread Class

- Class Thread: it's method run() does its business when that thread is run
- But you never call run(). Instead, you call start() which lets Java start it and call run()

```
class PrimeThread extends Thread {
        long minPrime;
        PrimeThread(long minPrime) {
            this.minPrime = minPrime;
        }

        public void run() {
            // compute primes larger than minPrime
             . . .
        }
    }
```

```
PrimeThread p = new PrimeThread(143);
     p.start();
```

Runnables and Thread

- Use the “task abstraction” and create a class that implements Runnable interface
  - Define the run() method to do the work you want
- Now, two ways to make your task run in a separate thread
  - First way:
    - Create a Thread object and pass a Runnable to the constructor
    - As before, call start() on the Thread object

  - Second way: hand your Runnable to a “thread manager” object
    - Several options here!

Runnables and Thread

- Use the “task abstraction” and create a class that implements Runnable interface
  - Define the run() method to do the work you want

```
class PrimeRun implements Runnable {
         long minPrime;
         PrimeRun(long minPrime) {
             this.minPrime = minPrime;
         }

         public void run() {
             // compute primes larger than minPrime
              . . .
         }
     }
```

```
PrimeRun p = new PrimeRun(143);
     new Thread(p).start();
```

Join (not the most descriptive word)

- The **`Thread`** class defines various primitive methods you could not implement on your own
  - For example: **`start`**, which calls **`run`** in a new thread

- The **`join()`** method is one such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its **`run`** returns)
  - E.g. in method foo() running in “main” thread, we call:
    myThread.start(); myThread.join();
  - Then this code waits (“blocks”) until myThread’s run() completes

- This style of parallel programming is often called “fork/join”
  - Warning: we’ll soon see a library called “fork/join” which simplifies things. In that, you never call join()

Synchronization

- Threads communicate primarily by sharing access to fields
- This form of communication is extremely efficient, but makes two kinds of errors possible
  1. *thread interference*
  2. *memory consistency errors*

Thread Interference

```
class Counter {
    private int c = 0;

    public void increment()
{
        c++;
    }
    public void decrement()
{
        c--;
    }
    public int value() {
        return c;
    }
}
```

If a *Counter* object is referenced from multiple threads, interference between threads may prevent simple operations from happening as expected.

Thread A: Retrieve c.
Thread B: Retrieve c.
Thread A: Increment retrieved value; result is 1.
Thread B: Decrement retrieved value; result is -1.
Thread A: Store result in c; c is now 1.
Thread B: Store result in c; c is now -1.

Memory Consistency Errors

- When different threads have inconsistent views of what should be the same data

How to avoid Inference & Inconsistency?

- Using Synchronization
    1. Synchronized methods
    2. Synchronized statements

(1) Synchronized Methods

```java
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

- It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

- When a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Locks

- Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*
- Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns.

(2) Synchronized Statements

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Examples

Atomic Access

- An *atomic* action is one that effectively happens all at once
  - An atomic action cannot stop in the middle.

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double)

More about Locks

- Synchronized code relies on a simple kind of reentrant lock
- Lock objects work very much like the implicit locks used by synchronized code
- As with implicit locks, only one thread can own a Lock object at a time
- The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock

Locks

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) {
            this.name = name;
        }

        public boolean impendingBow(Friend bower) {
            Boolean myLock = false;
            Boolean yourLock = false;
            try {
                myLock = lock.tryLock();
                yourLock = bower.lock.tryLock();
            } finally {
                if (! (myLock && yourLock)) {
                    if (myLock) {
                        lock.unlock();
                    }
                    if (yourLock) {
                        bower.lock.unlock();
                    }
                }
            }
            return myLock && yourLock;
        }
```

The tryLock method backs out if the lock is not available immediately or before a timeout expires

Issues with threads

- ***Deadlock*** describes a situation where two or more threads are blocked forever, waiting for each other
- ***Starvation*** describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress
- **Livelock**: A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result.

Executors

- In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object.

- This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application.

Executor Interfaces

- Executor, a simple interface that supports launching new tasks.
  - *ExecutorService*, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.

  - *ScheduledExecutorService*, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Fork/Join

- The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors.
- Basic use

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

New Java ForkJoin Framework

- Designed to support a common need
  - Recursive divide and conquer code
  - Look for small problems, solve without parallelism
  - For larger problems
    - Define a task for each subproblem
    - Library provides
      - a Thread manager, called a ForkJoinPool
      - Methods to send your subtask objects to the pool to be run, and your call waits until their done
      - The pool handles the multithreading well

Overview of How To

- Create a ForkJoinPool "thread-manager" object

- Create a task object that extends RecursiveTask
  - We'll ignore use of generics with this (see docs)
  - Create a task-object for entire problem and call invoke(task) on your ForkJoinPool

- Your task class' compute() is like Thread.run()
  - It has the code to do the divide and conquer
  - First, it must check if small problem – don't use parallelism, solve without it
  - Then, divide and create >1 new task-objects. Run them:
    - Either with invokeAll(task1, task2, …). Waits for all to complete.
    - Or calling fork() on first, then compute() on second, then join()

Mergesort Example

- Top-level call. Create “main” task and submit

```
public static void mergeSortFJRecur(Comparable[] list,
                                    int first,
                                    int last) {

   if (last - first < RECURSE_THRESHOLD) {
       MergeSort.insertionSort(list, first, last);
      return;
   }
   Comparable[] tmpList = new Comparable[list.length];
   threadPool.invoke(new SortTask(list, tmpList, first, last));

}
```

Mergesort's Task-Object Nested Class

```
static class SortTask extends RecursiveAction {
    Comparable[] list;
    Comparable[] tmpList;
    int first, last;

    public SortTask(Comparable[] a, Comparable[] tmp,
            int lo, int hi) {
        this.list = a;
        this.tmpList = tmp;
        this.first = lo;
        this.last = hi;
    }
// continued next slide
```

compute() Does Task Recursion

```
protected void compute() {

    if (last - first < RECURSE_THRESHOLD)
        MergeSort.insertionSort(list, first, last);

    else {
        int mid = (first + last) / 2;

        // the two recursive calls are replaced by a call to invokeAll
        SortTask task1 = new SortTask(list, tmpList, first, mid);
        SortTask task2 = new SortTask(list, tmpList, mid+1, last);
        invokeAll(task1, task2);
        MergeSort.merge(list, first, mid, last);
    }
}
```