CS 62 Midterm 2 Practice

2017-4-7

Please put your name on the back of the last page of the test.

Problem 1: Multiple Choice [8 points]

[2 points each] For each question, circle the correct answer. If you're unsure of the correct answer, you may mark a second guess with an asterisk.

1. What is the ordering constraint on values in a binary search tree?

   A. **Values to the left are always less than or equal to values to the right.**
   B. Values deeper in the tree are always less than values near the root.
   C. Each left child must be smaller than its parent, and right children must be larger.
   D. All of the above.

2. Why doesn't most parallel code run any faster than sequential code in big-O terms?

   A. That's not true: most parallel code *does* run faster in big-O terms.
   B. **With a finite number of processors, speedup is a constant factor, which gets dropped in big-O notation.**
   C. The overhead of starting threads means that the code will actually be slower most of the time.
   D. In big-O analysis, smaller terms get dropped, so when speedup subtracts some time, like O(n - n) the quantity reduces to O(n).

3. What does the `synchronized` keyword do when attached to a method?

   A. It causes that method to run in a new thread each time it's invoked.
   B. **It prevents threads from entering that method unless they have a corresopnding lock (the object the mehod is attached to).**
   C. It prevents the operating system from swapping to a different thread while inside of the method.
   D. It marks that method as one which uses shared ("synchronized") data.

4. What is the `hashCode` method supposed to do?

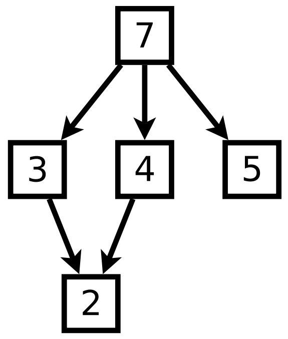
   A. Look up an object in a hash table.
   B. Return a random number that's hard to guess.
   C. **Return a semi-unique number based on an object's contents.**
   D. Store source code in a hash table.

Problem 2: Short Answer [8 points]

1. [4 points] **Splay Trees:** Explain why splay trees use three different operations to move items up to the root instead of just using a single "swap-with-parent" operation.

   If splay trees just swapped each node with its parent every time, then for a perfectly-unbalanced tree (a stick) the splaying operation would not alter the structure of the tree at any point. Because of the special zig-zig operation for rotating the parent before the grandparent, splay trees are guaranteed to eventually shorten stick structures.

2. [4 points] **Parallelism:** Given the following program graph, calculate T_1 and T_∞. What is the theoretical maximum speedup for this graph when executed in parallel?

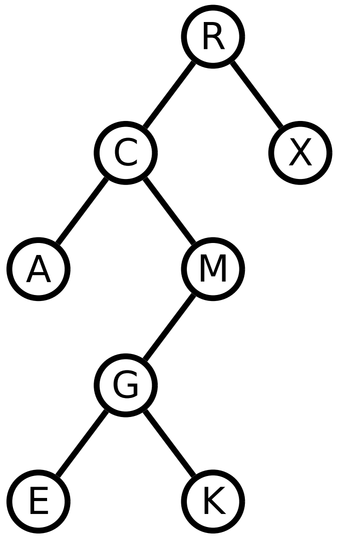


   T_1 is $7 + 3 + 4 + 5 + 2 = 21$

   T_∞ is the largest value between $7 + 3 + 2 = 12$, $7 + 4 + 2 = 13$, and $7 + 5 = 12$, which is 13. The theoretical maximum speedup for this code with perfect parallelization is therefore 21/13, or a bit less than 2.

Problem 3: Binary Search Trees [10 points]

1. [2 points] Given the following binary search tree (with letters kept in alphabetical order), where would the letter 'P' be inserted (draw on the digram below)?



The 'P' would be inserted as the right-hand child of the 'M,' because it's before 'R,' after 'C,' and after 'M.'

2. [4 points] When adding an item to a binary search tree which is a duplicate of an item already in the tree, where will that item wind up? Explain the role of the `predecessor` function in this process and what that function does. Indicate on the diagram above where a second 'M' would be inserted.

It will end up as the right-hand child of the right-most descendent of the left child of the node it's a copy of. The predecessor function finds this "rightmost descendent of the left child of a node" so that the copy can be inserted there. A second 'M' in this tree would accordingly be inserted as the right-hand child of the 'K.'

3. [4 points] What is the big-O run time of looking up a value in a binary search tree? Include best-case and worst-case times, and express each value twice: once in terms of h, the height of the tree, and again in terms of n, the total number of nodes in the tree.

The big-O run time of lookup in a binary search tree is best-case (perfectly balanced tree) O(h) which is O(log n), and worst-case (unbalanced tree/stick) still O(h) which is now O(n).

Problem 4: Parallelism [12 points]

Consider the following code:

```
class ParallelMin extends RecursiveTask<Double> {
    protected double[] data;
    protected int lo, hi;
    private static final int CUTOFF = 1000;

    public ParallelMin(double[] data, int lo, int hi) {
        this.data = data;   this.lo = lo;   this.hi = hi;
    }
    public Double compute() {

        if (______this.hi - this.lo <= CUTOFF____________) {
            double result = this.data[this.lo];
            for (int i = this.lo; i < this.hi; ++i) {
                if (this.data[i] < result) { result = this.data[i]; }
            }
            return result;
        } else {
            ParallelMin firstHalf = new ParallelMin(
                this.data,

                __this.lo___________________,

                __this.lo + (this.hi - this.lo)/2__
            );
            ParallelMin lastHalf = new ParallelMin(
                this.data,

                __this.lo + (this.hi - this.lo)/2__,

                __this.hi___________________
            );
            firstHalf.fork();
            result = firstHalf.join();
            if (lastHalf.compute() < result) { result = lastHalf.compute(); }
            return result;
        }
    }
}
```

(problem continues on next page)

1. [2 points] Fill in the five blanks above with lower and upper bounds so that the code divides the work in half with each recursive call until the `CUTOFF` value is reached.

2. [5 points] This code has two problems: first, it doesn't do any work in parallel, and second, it duplicates some work. Explain why these problems occur and what could be done to fix them.

The code fails to do work in parallel because `join` is called *before* `compute`. This means that the new thread created by `fork` is forced to finish before the current thread continues to do its half of the work. To fix this, just call compute before the call to join. That would also require storing the result of `compute` in a variable, which would fix the other problem: here `compute` is potentially being called twice: once to check its result against the result from the first half, and if that check succeeds, again to change the result value. `compute` should only be called once, and the result should be stored in a variable if it needs to be reused.

3. [5 points] Suppose that each iteration of the `for` loop in the `ParallelMin` code above takes 1 microsecond. Ignoring the time taken by other code (including time to set up threads and combine results), about how long would it take to find the minimum of 1000 elements on a machine that can run 4 threads at once? What about finding the minimum of 8000 elements?

Because 1000 is equal to our CUTOFF value, it should take about 1000 microseconds to find the minimum of 1000 elements, no matter how many cores we have. With 8000 elements, the work will be split into two batches of 4000, then 2000, then 1000, with a total of 8 threads. However, because we can only run 4 threads at once, our 8 threads will have to execute in the best case in two batches of four threads each, giving us a total time of 2000 microseconds.

Problem 5: Concurrency [8 points]

The code below shows two classes: an outer class `Parent`, and an inner class `Child`, where the `Parent` object always contains a `Child` and the `Child` object knows what its parent is. Each can get its own name, or a description that includes its own name and the name of its parent/child. Note which methods are and are not `synchronized`.

```java
class Parent {
    protected Child child;
    protected String name;

    public Parent(String name, String childName) {
        this.name = name;
        this.child = new Child(this, childName);
    }

    public synchronized String getName() { return this.name; }

    public synchronized String description() {
        return this.getName() + ", the parent of " + this.child.getName();
    }

    public String childDescription() {
        return this.child.description();
    }

    protected static class Child {
        protected Parent parent;
        protected String name;

        public Child(Parent parent, String name) {
            this.parent = parent;
            this.name = name;
        }

        public synchronized String getName() { return this.name; }

        public synchronized String description() {
            return this.getName() + ", the child of " + this.parent.getName();
        }
    }
}
```

(problem continues on next page)

1. [3 points] Assume that a single `Parent` object named ‘A’ with a single `Child` named ‘B’ has been constructed. When the `getName` method of the `Parent` object ‘A’ is called, what object serves as the lock for synchronization? Likewise, when the `getName` method of the `Child` object ‘B’ is called, what object serves as the lock?

For the call to ‘`A.getName`’, the object ‘A’ will serve as the lock, and a call to ‘`B.getName`’ will use object ‘B’ as the lock.

2. [5 points] Your friend is using this `Parent` and `Child` code is a multi-threaded project where multiple threads use the `description` and `childDescription` methods of a shared `Parent` object ‘A’ at once. They notice that sometimes, their program seems to freeze, but they’re confused because their program doesn’t have any loops that could be infinite. Explain why their code freezes, and given an example of a sequence of events that causes it to freeze.

Their code freezes because of deadlock. Because the `childDescription` method is not synchronized, it will call `description` on the child, which will acquire the child’s lock, and then attempt to call `getName` on the parent, which will in turn require the parent’s lock. At the same time, code calling `description` on the parent will first acquire the parent’s lock and then attempt to get the child’s lock when it calls `getName` on the child. If these calls happen in the wrong order, deadlock will result as one thread holds the parent lock and is waiting for the child lock, while another thread holds the child lock and is waiting for the parent lock. This could occur if the operating system switches threads during the `description` method of the child before the call to `this.parent.getName`, and while that thread is still paused, another thread enters the `description` method of the parent, and gets to the `this.child.getName` part and begins waiting for the child lock (which is held by the other thread). When the other thread resumes, it will immediately attempt to call `getName` on the parent, and enter a deadlocked state.

Name: ______________________________