CMPSCI 187: Programming With Data Structures

Review for Final Exam
David Mix Barrington
10 December 2012

Exam Overview

- Thursday 13 December, 1:30-3:30 p.m., Marcus 131
- Format is the same as the Fall 2011 finals and similar to our midterms.
- Question types: Vocabulary (10 x 2 = 20), Software Engineering (3 x 5 = 15), Tracing Code (3 x 5 = 15), Finding Errors (3 x 5 =15), Timing Analysis (3 x 5 = 15), Short Coding (2 x 10 = 20), Long Coding (20), total 120 points.
- Scale: Optimistic is A = 105, C = 70, F = 35, might move down if needed.
- Exam is cumulative, with 50% of coverage on the last third of the course and 25% each for the first two thirds of the course.
- Exam counts for either 25% or 50% of course grade, whichever is better for you.

Content Overview

- **First Third**: Basic Concepts (Data Structures, Software Engineering, Timing), Array-Based and Linked Implementations, Stacks. **Projects**: StringBag, Sudoku.
- **Middle Third**: Recursion in General, Queues, Lists. **Projects**: MapGrids (testing for existence of paths in a grid).
- **Final Third**: Binary Search Trees, Heaps and Priority Queues, Sorting, Hashing. **Project**: Frequency of Words (with priority queues).
- **Throughout**: Java Literacy, Programming Practice, Abstraction of Data Structures.

First Third of the Course

- We outlined the main themes of the course. **Data structures** are both abstract design ideas and concrete programming constructs. We define a data structure by what data it keeps and what methods it will support. Separating the interface of a data structure from its implementation aids our **software engineering** practice. Careful analysis of the **running time** of the methods of our data structures helps us choose which ones to use.

- Our implementations of data structures were generally either **array-based** or **link-based**. In general array-based structures are easier to search and run faster, other things being equal, due to memory locality. Link-based structures allow for easier insertion and deletion of elements in general places.

- The **StringLog** was our toy example of a general data structure. We then looked at the **Stack**, which allows removal only of the last element added. We implemented stacks with arrays and links, and solved Sudoku with a stack.

Middle Third of the Course

- We studied **recursion** as a general programming technique. Recursion allows for code that is simpler and easier to verify from a definition. It may run more slowly than equivalent iterative code. Bad recursions will repeatedly do the same work, which can be very slow. Designing a recursion requires a general problem statement whose solution can be defined in terms of itself. Verifying the correctness of recursive code means identifying the base case, showing it correct, showing that the general case must make progress toward the base case, and showing the general case correct *if* the recursive calls work.

- We studied the **Queue** and the **List** in the same way we studied the Stack, with both array and linked implementations. In a queue we can only remove the element that has been there the longest. In a list we can add, remove, and find arbitrary elements, with running times depending on the implementation (which includes whether the list is **ordered** or **indexed**). Our MapGrid projects used queues to find shortest paths on a grid.

Final Third: Binary Search Trees

- A **binary search tree** is a link-based structure, where elements are in **nodes** and each node may have a left and/or a right **child**. If a node has element x, the elements in its left child and anything under its left child are ≤ x, and the values in the corresponding right subtree are ≥ x. (So x must be from a class with a `compareTo` method.) This is the **BST rule** -- it implies that the **inorder traversal** of the tree gives the elements in order.

- We can look for an element by following a **path** from the root to either where the element is or where it has to be. This is the place to insert a new element. Removing an element is a bit more complicated, especially if its node has two children. The time to find is at worst the **depth** of the tree. This is O(log n) for an n-node tree if the tree is **balanced**. But the shape of the tree depends on the order in which the elements were added.

- DJW used a binary tree to count **frequencies of words** in a text, with the inorder traversal of the tree keeping alphabetical order on words.

Final Third: Heaps and Priority Queues

- A **priority queue** also has comparable elements -- we can insert new elements and remove the largest element. There are many ways to implement a priority queue, with different running times for the operations.
- A **heap** is a tree-based structure, usually using **implicit pointers** in an array. In a heap, the only rule is that the element at a node is ≥ the elements at its children, so that the top element is always at the root. A heap with n nodes has a fixed shape, with the tree balanced except for a left-justified bottom level.
- We can add a new element by making a hole in the new location and **reheaping up** until the new element fits in the hole. We remove the top element by creating a hold and **reheaping down** until the element in the last location fits. Each of these operations takes O(log n) time in the worst case.
- In Project #5 you used Java's heap-based priority queue class to count word frequencies.

Final Third: Sorting, Searching, and Hashing

- We can **sort** any collection of comparable elements by putting the elements in order. We measure the number of comparisons needed to sort n elements using various methods. We showed that every sort needs at least O(n log n).
- Simple sorts like **insertion**, **selection**, and **bubble** sorts take O(n^2) comparisons but may be useful if access to the collection prevents other methods or if (in case of insertion) the collection is already nearly sorted.
- We looked at three O(n log n) sorts: **Merge Sort**, **Quick Sort**, and **Heap Sort**. Quick Sort's performance depends on the input and only averages O(log n). Merge Sort needs more temporary storage but can be used on files. Heap Sort is similar to using a heap-based priority queue, but is slightly faster.
- We compared the time to search for an element in the various collections. With hashing, we can simulate a dedicated-slot array and get (in practice) O(1) average time to insert, remove, or find elements.

Fall 2011 Final Exam -- Vocabulary

pop and peek in a (java.util) stack

postfix expression and prefix expression

ordinary array and circular array

queue and deque

binary search and linear search

hasNext( ) and next( ) in an iterator

direct and indirect recursion

HashSet and HashMap (we only briefly mentioned these in fall 2012)

chaining and open addressing (linear probing) in a hash table

child (in a tree) and descendant (in a tree)

Fall 2011 Final Exam: Software Engineering

- These questions depended on Fall 2011 projects, of course. Your questions will depend on the Fall 2012 projects or on case studies in DJW. But the types of questions are typical:
- Change the specification in a project and ask how the code changes.
- Use a known data structure to implement some job that you have seen in a project or case study, implemented there in another way.
- Indicate how you would attack a new job in the context of one of the projects or case studies.

Fall 2011 Final Exam -- Tracing Code

- Dogs removed in order are Cardie, Duncan, Cardie, and Biscuit.

```
// This is an L&C priority queue -- "next" is the element
// in the queue whose priority is the smallest integer.  It breaks
// ties by earlier creation, i.e., it is FIFO on elements with the
// same priority.  Assume standard dogs have been created.
  PriorityQueue<Dog> pq = new PriorityQueue<Dog> ( );
  pq.addElement (cardie, cardie.getAge( )); // 3
  pq.addElement (king, king.getAge( )); // 73
  Dog z = pq.removeNext( );
  pq.addElement (duncan, duncan.setAge( )); // 1
  z.setAge(1);
  pq.addElement (z, z.getAge( ));
  z = pq.removeNext( );
  pq.addElement (biscuit, biscuit.getAge( )); // 1
  pq.addElement (buck, buck.getAge( )); // 108
  pq.removeNext( );
  System.out.println (pq.removeNext( ).getName( ));
```

Fall 2011 Final Exam -- Tracing Code

- This uses a class definition from discussions, repeated on the exam.
- The trick here is that the team contains two pointers to the same dog, so that when we change the name of one the other's name changes as well.

```
// Again, assume that the dogs from above have been created.
// This code is not in the DogTeam class.
DogTeam t = new DogTeam( );
t.addToLead (king);
t.addToLead (balto);
t.addtoLead (king);
t.switchLastTwo( );
SledDog s = t.removeLead( );
s.setName ("Yukon King");
SledDog z = t.removeLead( );
System.out.println (z.getName( ));
```

Fall 2011 Final Exam -- Tracing Code

- In a BST of Integer objects, we successively add 3, 1, 4, 15, 9, 2, 6,and 5. Then we successively remove 4, 9, and 3 in that order.
- The adds make a tree which has 3 at root, 1 as left child with right child 2, and 4 as left child with right child 15 and a left-chain with 9, 6, 5 below 15.
- Since the 4 and 9 are unary nodes we just splice them out. The root’s right child is now 15 with left-chain 6 and 5 below it.
- To remove 3, which has two children, DJW replaces it with its inorder predecessor and removes that node recursively. (The solutions refer to L&C’s different definition.) Here we move the 2 to the root and remove the leaf with the 2.

Fall 2011 Final Exam -- Finding Errors

- Again, don't try to assume what the code is trying to do if it doesn't tell you in a comment or with the title.
- This code will compile and run, but if n is odd it will get stuck at count = n - 1, never leaving the loop because it no longer changes the value of count.

```
public int summation (int n) {
      int count = 0, sum = 0;
      while (count < n) {
         sum += count;
         if (count < n) sum += (count + 1);
         if (count < n - 1) count += 2;}
      return sum;}
```

Fall 2011 Final Exam -- Finding Errors

- Again, this is rather sensible code that would actually convert the dog team to an array as it says, except that the array is created improperly -- it is not given a size.

```
// method to be added to DogTeam class
public SledDog[ ] toArray ( ) {
   SledDog [ ] result = new SledDog[ ];
   LinearNode<SledDog> cur = leadNode;
   int index = 0;
   while (cur != null) {
      result [index] = cur.getElement( );
      cur = cur.getNext( ); index++;}
   return result;}
```

Fall 2011 Final Exam -- Finding Errors

- This is attempt to reverse the calling DogTeam, as a side effect since it is void. It saves the lead dog, moves the others into a temporary team, reverses that team, puts the lead dog back, and moves the other dogs back. This makes progress toward the base case and would be right if the recursive reversal were right. But there is no code for the base case -- the way it fails depends on what happens when you remove the lead dog of an empty team.

```
// method to be added to DogTeam class
public void reverse ( ) {
   SledDog lead = this.removeLead( );
   DogTeam temp = new DogTeam( );
   while (this.getSize( ) > 0)
      temp.addToLead (this.removeLead( ));
   temp.reverse( );
   this.addToLead (lead);
   while (temp.getSize( ) > 0)
      this.addToLead (temp.removeLead( ));}
```

Fall 2011 Final Exam -- Timing Analysis

- The first spends O(1) time to divide n by 2 and recurse -- total time O(log n). The second takes O(n + (n-1) + (n-2) + ... + 1) = O(n^2).

```
public static String binary (int n) {
      if (n == 0) return "0";
      if (n == 1) return "1";
      if (n % 2 == 0) return binary (n/2) + "0";
      return binary (n/2) + "1";}

// n is size of input stack s
   public static void reverse (StackADT<Dog> s) {
      if (size <= 1) return;
      StackADT<Dog> temp = new ArrayStack<Dog> ( );
      while (s.size( ) > 1)
         temp.push (s.pop( ));
      reverse (temp);
      Dog last = s.pop( );
      while (!temp.isEmpty( ))
         s.push (temp.pop( ));
      s.push (last);}
```

Fall 2011 Final Exam -- Timing Analysis

- This is a reminder that big-O running time is a function of n as n gets large. Here as n gets large, the code does nothing and thus takes O(1) time. The remaining code would be O(n) if not for the cutout.

```
// uses L&C code base; n is size of input queue
   public static void reorder (QueueADT<Dog> q) {
      int n = q.size( );
      if (n >= 11) return;
      Dog d = q.first( );
      QueueADT<Dog> temp = new ArrayQueue<Dog> ( );
      for (int i = 0; i < 5; i++)
          if (!q.isEmpty( )) temp.enqueue (q.dequeue( ));
      while (!temp.isEmpty( ))
          q.enqueue (temp.dequeue( ));
      while (d != q.first( ))
          q.enqueue (q.dequeue( ));}
```

Fall 2011 Final Exam -- Coding Questions

- The first short problem dealt pretty closely with the prefix tree of that term's Project #7, which we didn't do. It was to write new code to remove a leaf node from that tree, which had consequences further up the tree.

- The second short problem was to implement a hash table with chaining, where each entry of the table was a list, in this case an ArrayList. All you needed to do was find the right table entry and use list operations. A complication was that your input was an Object that you had to cast.

- The long problem involved Sudoku, but rather than find a solution you just had to allow the user to give you moves and undo the moves they had made. The undo command forced you to keep the moves that had been made on a stack. You needed to create what was more or less the method called isBad in our Project #2, though it only needed to work on a completed board.