Cornell University
Computer Science 211

Second Preliminary Examination 18 April 2006

Solutions

1. True/false [24 pts] (parts a–h)

   Each question is worth 3 points.

   (a) The worst-case running time of quicksort is worse than that of mergesort.

   True

   (b) The job of a specification is to explain how code is implemented.

   False

   (c) The lower bound for general sorting algorithms (algorithms that work on any type) is $\Omega(n \log n)$

   True

   (d) Binary search on an array takes $\Omega(n \log n)$ time.

   False

   (e) Open addressing works relatively well with a high load factor.

   False

   (f) Insertion sort is faster than quicksort on a sorted array.

   True

   (g) Stacks are FIFO.

   False

   (h) Set abstractions can be implemented simply using map abstractions.

   True. A lot of people found this question hard. If you have a map abstraction, one easy way to implement a set abstraction is as a map from the elements of the set to some standard value. A set operation like `contains` *can then be implemented using the map's* `get` *operation.*

2. Sorting [25 pts] (parts a–c)

   (a) [10 pts] Suppose we use mergesort to sort an array containing the following sequence of elements: 1, 5, 6, 3, 2, 4, 9, 0. For this example input, illustrate how merge sort works by drawing the array states that result after each merge.

   Answer:

```
initial: 1, 5, 6, 3, 2, 4, 9, 0
      -> 1 5,  3 6,  2 4,  0 9
      -> 1 3 5 6,    0 2 4 9
      -> 0 1 2 3 4 5 6 9
```

   (b) [7 pts] Suppose we use quicksort but as a pivot we use the mean (average) of the elements. For simplicity, let us assume that the mean of the elements in the array (or any subarray) always partitions the elements into two equal-sized sets. Explain briefly why quicksort will still take $O(n \lg n)$ time in that case.

   Answer:
   To use the mean, we need to compute it (many people overlooked this). Computing the mean takes $O(n)$ time, because it is necessary to sum all the elements. But partitioning an array of size n also takes $O(n)$ time already. So computing the mean does not make quicksort asympotically slower.

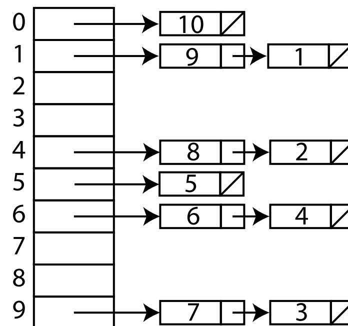
(c) [8 pts] Explain briefly why the worst-case time of quicksort with a mean pivot is still $O(n^2)$. (Hint: consider an array containing elements $i!$).

Answer:
An array of elements $i!$ for i ranging from 0 to n will have the property that its mean is at least $(n-1)!$, so partitioning will merely split off the element $n!$ from the rest. This will happen at every recursive call, so there will be n recursive calls each taking $O(n)$ time, for a total of $O(n^2)$ time. Many people did not do a convincing explanation of the $O(n^2)$ bound.

3. Hash tables [26 pts] (parts a–c)

(a) [10 pts] Suppose we have a hash table with an table size of 10, into which we are inserting the integers from 1 to 10. Our hash function $h(x)$ is the square of the element, modulo the table size (i.e., the remainder when the square is divided by the table size. For example, $h(9) = 1$.) Draw the hash table that results, assuming it uses chaining.

Answer:



(b) [6 pts] When might this hash function $h(x) = x^2 \bmod m$ be a poor choice, assuming m is the length of the hash table? Discuss briefly with an example.

Answer:
This would be a bad hash function if the keys x tend not to be relatively prime in m. Keys with a common factor f will be hashed to at most one in f buckets.

(c) [10 pts] You are given an array of numbers of length n and a sum s. We can create an $O(n)$ algorithm that determines whether two numbers in the array add to the value s. For example, for the array 1, 3, 2, 5 and the sum $s = 8$, the result is `true`, but for $s = 2$ and $s = 10$, it would be `false`. You should clearly describe an algorithm that accomplishes this in $O(n)$ time, and justify why it is $O(n)$. You may write code or pseudocode, but this is not required. Solutions that are not $O(n)$ will be accepted, with some penalty.

Answer:
The key is to have a data structure on the side to keep track of which elements are in the array. In accordance with the title of the problem, it should be a hash table. The algorithm scans through the array once looking at each element x. Then it checks whether $s - x$ is in the hash table. If so, return true. Otherwise, add x to the hash table and continue. Since $O(1)$ work is done for each element, the whole loop is $O(n)$.

4. Implementing a collection [25 pts] (parts a–d)

Consider the following implementation of a set abstraction as a circularly linked list. This implementation uses an extra list node object to represent the end of the list, rather than `null`. This is known as a **sentinel** object; this sentinel also serves as the header object for the whole list, avoiding having a separate class. The `elem` field in the sentinel is unused.

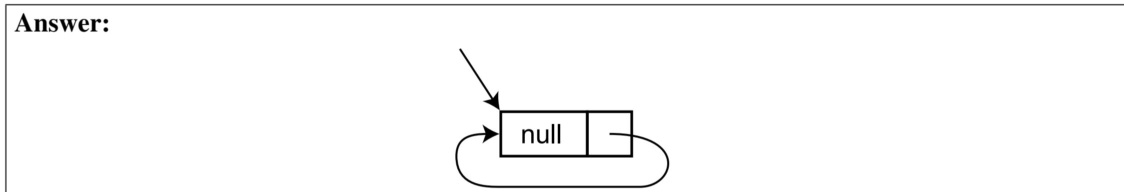
```
class List<T> implements Collection<T> {
  private T elem;
  private List<T> next;

  /** Create an empty list. */
  public List() {
     next = this;
  }
  public boolean contains(T x) {
    List<T> curr = next;
    while (curr != this) {
      if (curr.elem == x) return true;
      curr = curr.next;
    }
    return false;
  }
  public boolean add(T x) {
    if (contains(x)) return false;
    List<T> nw = new List<T>();
    nw.elem = x;
    nw.next = next;
    next = nw;
    return true;
  }
  public boolean remove(T x) {
    List<T> curr = this;
    while (curr.next != this) {
      if (curr.next.elem == x) {
        curr.next = curr.next.next;
        return true;
      }
      curr = curr.next;
    }
    return false;
  }

  public Iterator<T> iterator() {
    return new ListIterator();
  }
  private class ListIterator implements Iterator<T> {
     List<T> curr = next;
     boolean hasNext() { ...  }
     T next() { ...  }
  }
}
```

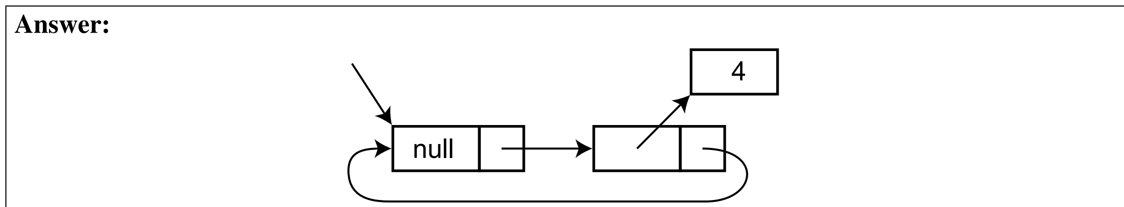
(a) [4 pts] Draw what the data structure created by `List<Integer>()` looks like.

Answer:



(b) [4 pts] Now draw what it looks like after adding the element 4.

Answer:



(c) [4 pts] Identify an error common to the implementations of both the `contains` and `remove` methods and explain how to fix each of them by changing one line of code in each.

Answer:

They both use `==` *where they should use the* `equals` *method.*

(d) [13 pts] Implement the iterator class `ListIterator` with code in which the two `...`'s are filled in. All iterator operations should be $O(1)$. (Hint: the containing `List` object can be accessed in `ListIterator` with the expression `List.this`.) If you can't figure out how to do it by just filling in the `...`'s, you can implement it some other way with only a 3-point deduction (if it works).

Answer:

```
class ListIterator implements Iterator<T> {
   List<T> curr = next;
   boolean hasNext() {
     return curr != List.this;
   }
   T next() {
     if (curr != List.this) {
       T ret = curr.elem;
       curr = curr.next;
       return ret;
     }
     throw new NoSuchElementException();
   }
}
```

Many people forgot to save `curr.elem` *in a temporary variable, skipping the first element in the list.*

Bonus problem

Consider the following implementation of a set. We use a pair of arrays in which the longer array is kept sorted and the shorter one is not. The length of the shorter array is the square root of the length of the longer array, although it may have some empty entries if it is not full yet. To add an element, we insert it in order in the short array, moving other elements as necessary. If the short array fills up, we allocate larger long and short arrays, and merge the two old arrays into the new long array. The new short array is empty. To find an element, we use binary search on both the long and short arrays.

Show that this data structure gives $O(\lg n)$ lookup time, but the amortized time to add an element is $O(\sqrt{n})$.

Answer:

Because both the arrays are sorted, binary search takes logarithmic time in both. Suppose there are m elements in the larger array and at most $\sqrt{m}$ elements in the smaller. Then the lookup time in the larger $O(\log m)$ and in the smaller, $O(log\sqrt{m}) = O(\frac{1}{2}\log m) = O(m)$. So the full lookup is $O(\log m)$. Because m is $O(n)$, lookup is $O(\log n)$.

When we insert an element into the smaller array, we pay $O(\sqrt{m})$ cost moving all the elements to make space for the new one. However, there is also the cost of doing periodic merges. The intuition is that merges take $O(n)$ work, but they only have to be done every $\sqrt{m}$ insertions. So inserting $\sqrt{m}$ elements incurs $O(m)$ work, with an amortized cost of $O(m/\sqrt{m}) = O(\sqrt{n})$.

This simple analysis was enough to get credit, but we can make this analysis more precise by pretending that the computer has a meter inside it that has to be fed money to keep it going. When an operation is called, enough money has to be supplied to finish the operation. If an operation works correctly with $O(\sqrt{n})$ money, then we can say that the operation takes $O(\sqrt{n})$ time.

For a proper amortized analysis, the key idea is charge extra on some operations, and store the extra money to inside the data structure to pay for later operations. In this case, we want to save enough money to pay for the periodic merges, while still paying only $O(\sqrt{n})$ money on any given insertion.

Suppose we want to insert n elements into the data structure, doing some number of merges along the way. Suppose that at the ith merge, the long array has length m_i and the short array, $\sqrt{m_i}$. Then (non-merging) insertion takes time that is $O(\sqrt{m_i})$ and merging takes time $m_i + \sqrt{m_i}$. We can charge $3\sqrt{n}$ money for each insertion, using the first $\sqrt{n}$ to do the insertion (since $n > m_i$), saving the remaining $2\sqrt{n}$ for later. When it comes time to do a merge, we will have saved up $(\sqrt{m_i}) \cdot (2\sqrt{n})$ money, which is at least $2m_i$, greater than the $m_i + \sqrt{m_i}$ needed to pay for merging. Since each insertion costs $O(\sqrt{n})$ money, this is the amortized cost.