CS 61B: Data Structures (Spring 2014)

Final Exam

Solutions

Problem 1. (12 points) A miscellany.

b. The object referenced by `y` (and now by `x`), if not `null`, has its counter incremented. The object formerly referenced by `x`, if not `null`, has its counter decremented.

c. Reference counting permits you to free the memory occupied by a zero-count object immediately, whereas mark-and-sweep garbage collection can only collect garbage after marking all live objects by performing a complete sweep phase, during which the program is frozen and may pause for a noticeable period of time.

d. $\Theta(1)$ time, because Java implements references as handles (pointers to pointers). Therefore, only one pointer must change when an object is moved.

e. To compress the paths so that no treenode has a grandchild:

```
find(9);
find(5);
```

(It doesn't matter what order these two operations are in.)

f. Here are two different acceptable answers.

Strings (unlike game boards) come in different lengths. Because we represent the letter `a` with 0, the hash code does not distinguish between Strings like `moral` and `amoral`, or `sexual` and `asexual`; these pairs collide.

Our second answer is trickier, because it requires some understanding of bit representations of `int`s. If there are more than 32 letters in the String, the leftmost letters have no effect on the hash code, because they are multiplied by 26^{32} or a higher power of 26, whose last 32 bits are all zero. (Note that it is not a problem that the hash code overflows—that works perfectly fine for base-3 numbers, like when we hash a checkerboard—but it is a problem that the low-order bits that survive the overflow are zero.)

g.

```
Array contains 0
Array contains 0
Except my apologies
Finally it's over
java.lang.Exception: I won't do it again
```

(Note: we don't penalize students who don't remember the "java.lang.Exception:" part.)

h. $f(x, y) = x^3$. (There are many other possibilities, of course.)

i. $\Theta(\log \log n)$.

Problem 2. (11 points) Trees.

a.

b. The minimum depth at which the fourth-smallest key can reside is 1. The maximum depth at which the fourth-smallest key can reside is 3.

c. Every edge connects one child to its parent. Every vertex, except the root, adjoins only one edge that connects it to its parent. Therefore, there is a one-to-one correspondence between edges and children (vertices that aren't the root). So an n-node tree must have n - 1 edges.

d. Here are two acceptable answers, both using hash tables.

The easy answer: Use a hash table to map each key to its corresponding value. The binary heap stores only the keys—not the values. That way, `replace(k, v)` needs to change only an entry in the hash table; it does not change the heap. `insert(k, v)` adds `k` to the heap and `(k, v)` to the hash table. `removeMin()` removes the minimum key from the heap and its entry from the hash table.

The hard answer: Use a hash table to map each key to its index in an array-based binary heap. The binary heap stores (key, value) pairs. Whenever a key in the heap is moved (e.g. swapped with another key), we must also update its hash table entry to store its new index. `replace(k, v)` uses the hash table to find the key's index in the binary heap, then replaces its value with `v`.

e. Add a field `biggest` that stores the maximum key currently in the binary heap. Whenever a new key is inserted, check whether it is greater than `biggest`; if so, update `biggest`. The `max()` operation simply returns the value of `biggest`.

Most operations do not reduce the value of `biggest`, with one exception: if `removeMin()` completely empties the heap, then `biggest` is set to $-\infty$.

e. A postorder traversal.

Problem 3. (11 points) Sorting.

a.
3 7 9 5 1 0 4 8 2 6
6 7 9 5 1 0 4 8 2 3
2 7 9 5 1 0 4 8 6 3
2 0 9 5 1 7 4 8 6 3
2 0 1 5 9 7 4 8 6 3
2 0 1 3 9 7 4 8 6 5
0 2 1 3 9 7 4 8 6 5
0 1 2 3 9 7 4 8 6 5
0 1 2 3 5 7 4 8 6 9
0 1 2 3 6 7 4 8 5 9
0 1 2 3 4 7 6 8 5 9
0 1 2 3 4 5 6 8 7 9
0 1 2 3 4 5 6 7 8 9

b. Use binary search to find the location to insert each key.

c. We accepted two answers. Some of you may be aware of a trick for merging k

queues of total length n in O(n log k) time by using a binary heap of queues; most of you are not.

If you don't know the trick, the top-level merging step takes $\Theta(nk)$ worst-case time, so an entire run takes $\Theta(nk \log_k n) = \Theta(nk \log n / \log k)$ worst-case time.

If you know the trick, the top-level merging step takes $\Theta(n \log k)$ worst-case time, so an entire run takes $\Theta(n \log k \log_k n) = \Theta(n \log n)$ worst-case time.

As an aside, here's an interesting observation that gives some insight why k-way mergesort with the merging trick runs in O(n log n) time regardless of the value of k. If we choose k = 2, we have pure mergesort. If we use the merging trick and choose k = n, we have pure heapsort; we're using a heap to merge n queues of length 1. But intermediate values of k yield a mergesort-heapsort hybrid algorithm.

d.

```
1110101  1000101  0110001  0000101  1110001  0000001  1000101  1000000
1000000  0110001  1110001  0000001  1110101  1000101  0000101  1000101
1000000  0000001  1000101  0000101  1000101  0110001  1110001  1110101
0000001  0000101  0110001  1000000  1000101  1000101  1110001  1110101
```

Halfway through the key-copying loop of the first pass of counting sort, the `counts` array looks like this.

	000 (0)	001 (1)	010 (2)	011 (3)	100 (4)	101 (5)	110 (6)	111 (7)
counts	0	2	4	4	4	7	8	8

Problem 4. (4 points) Graphs.

a. Depth-first search visits the vertices in the order 0, 6, 2, 4, 5, 3, 1.

b. $\Theta(v + e)$. The term e follows because the sum of all the vertex degrees is 2e. The term v follows because we have to check each vertex for an adjacency list. (If there are many more vertices than edges, most vertices have degree zero, but we still have to spend time checking whether they have degree zero.) Observe that this reasoning is largely the same as the reasoning we gave you for why depth-first search runs in $\Theta(v + e)$ time.

c. During the preparation stage, when we have as much time as we want, we prepare the following data structures.

- A v-by-v adjacency matrix with no edges (yet).
- An array-based disjoint sets data structure representing v items (each in its own set).
- A v-vertex adjacency list, in the form of an array of linked lists, with no edges (yet).
- An array of v `visited` flags, all set to `false`.

During the contest stage, we start reading edges from L from left to right. We use the adjacency matrix to screen out duplicate edges: when we read an edge, if the edge is already entered in the adjancency matrix, we ignore it and go directly to the next edge. But if the edge is new, we enter it in the adjancency matrix. Then we check whether its endpoints are in the same disjoint set. If not, we unite

(union operation) the sets in which its endpoints lie, and go on to the next edge. But if its endpoints are in the same disjoint set, we have discovered that there is a cycle! (Note that this works only because we discarded duplicate edges.) However, we do not add this new edge to the graph data structure.

Once we have discovered a cycle, we must identify which vertices are in the cycle. We can't simply print the whole disjoint set; there may be edges adjoining the cycle that are not part of the cycle. Therefore, we will use depth-first search (or breadth-first search) to identify the cycle. DFS is not fast enough if we use the adjacency matrix; therefore, we build an adjacency list as we read the edges. When we discover that there is a cycle, we run the search to find a path from one endpoint to the other. This search uses the array of `visited` flags.

It takes only constant time per edge to check and update the adjacency matrix and the adjacency list, for a total of $O(k)$ time. Each edge may entail two find operations and one union operation, so the total time for all the disjoint sets operations for all the edges is in $O(k\ \alpha(3k, k))$. The depth-first search runs in $O(k)$ time, because there are k edges in the adjacency list and the array of `visited` flags is already initialized. The total running time is in $O(k\ \alpha(3k, k))$, which is less than $O(k\ \sqrt{\log k})$.

Problem 5. (7 points) Disjoint Sets.

a.

```
public void union(SibTreeNode other) {
  if (size < other.size) {                          // other has larger tree
    other.size += size;
    parent = other;
    nextSibling = other.firstChild;
    other.firstChild = this;
  } else {                                  // "this" has equal or larger tree
    size += other.size;
    other.parent = this;
    other.nextSibling = firstChild;
    firstChild = other;
  }
}
```

b. Consider two roots that each have $\Theta(n)$ children. Take the union of those two roots; now one of the roots has $\Theta(n)$ grandchildren. Suppose we perform one `find` operation on each of those grandchildren, in order from right to left. To perform path compression on a grandchild, we must remove it from the singly-linked list of its parent's children. Because it is the last child in the list, we must walk through the entire list to change the `nextSibling` reference of the previous child. Therefore, it takes $\Theta(n)$ time on average for each `find`.

c. Give every node a `prevSibling` reference. Siblings are now doubly-linked, so a node can be removed from the list of its parent's children in constant time.

Problem 6. (5 points) Quickselect.

```
public static Comparable quickselect(Comparable[] a, int k, int low, int high) {
  if (low < high) {
    // Generate a random number in the range low...high.
    int pivotIndex = low + (new Random()).nextInt(high - low + 1);
```

```
      Comparable pivot = a[pivotIndex];
      a[pivotIndex] = a[high];                    // Swap pivot with last item
      a[high] = pivot;

      int i = low - 1;
      int j = high;
      do {                                        // Partition the (sub)array
        do { i++; } while (a[i].compareTo(pivot) < 0);
        do { j--; } while ((a[j].compareTo(pivot) > 0) && (j > low));
        if (i < j) {
          Comparable c = a[i];  a[i] = a[j];  a[j] = c;   // Swap a[i] and a[j]
        }
      } while (i < j);

      a[high] = a[i];
      a[i] = pivot;                 // Put pivot in the middle where it belongs
// !!! Changes start here !!! (No changes were needed above this line) !!!
      if (k < i) {
        return quickselect(a, k, low, i - 1);         // Find item in left list
      } else if (k == i) {
        return pivot;
      } else {
        return quickselect(a, k, i + 1, high);       // Find item in right list
      }
    } else {
      return a[low];
    }
  }
```

Mail inquiries to cs61b@cory.eecs