COS 226	Algorithms and Data Structures	Fall 2012

Midterm Solutions

1. **Union find.**

   A B C

   A. The `id[]` array contains a cycle: $8 \to 2 \to 4 \to 0 \to 8$.

   B. The height of the forest is $4 > \lg(10)$.

   C. The size of tree rooted at the parent of 3 is less than twice the size of tree rooted at 3.

   D. The following sequence of union operations would create the given `id[]` array:

   ```
   2-0 1-8 7-9 0-9 8-5 4-1 1-9 3-8 5-6
   ```

2. **Eight sorting algorithms.**

   0 2 9 7 6 8 5 4 3 1

3. **Analysis of algorithms.**

   (a) $\sim N^2$

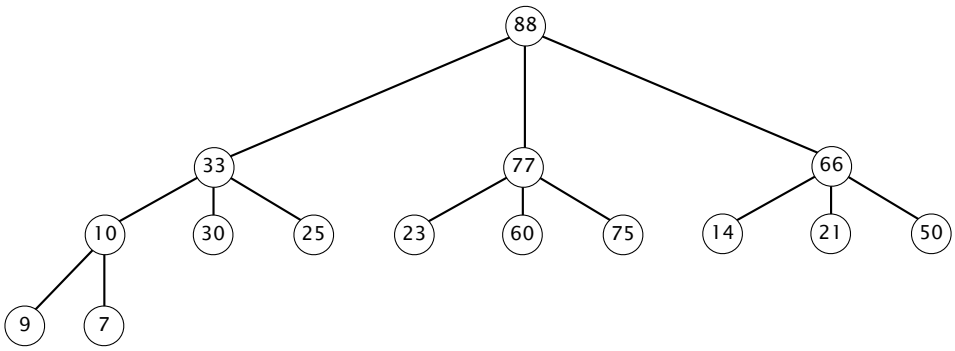
   Each of the N `B`s must be compared and exchanged with each of the N `A`s.

   (b) $\sim 3N$

   The first partition uses $\sim 2N$ compares. After the first partition, all of the keys equal to the partitioning item (either all of the `A`s or all of the `B`s) will be finished. The second (and final) partition will complete the sort using $\sim N$ compares.

4. **3-heaps.**

   Here is a drawing of the original 3-heap:



(a) To *delete-the-maximum*, we exchange 88 with 7 and sink 7 down. To sink 7 down, we exchange 7 with its largest child (77) and then again exchange 7 with its largest child (75).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
--	77	33	75	66	10	30	25	23	60	7	14	21	50	9	--

(b) $3k - 1,\ 3k,\ 3k + 1$

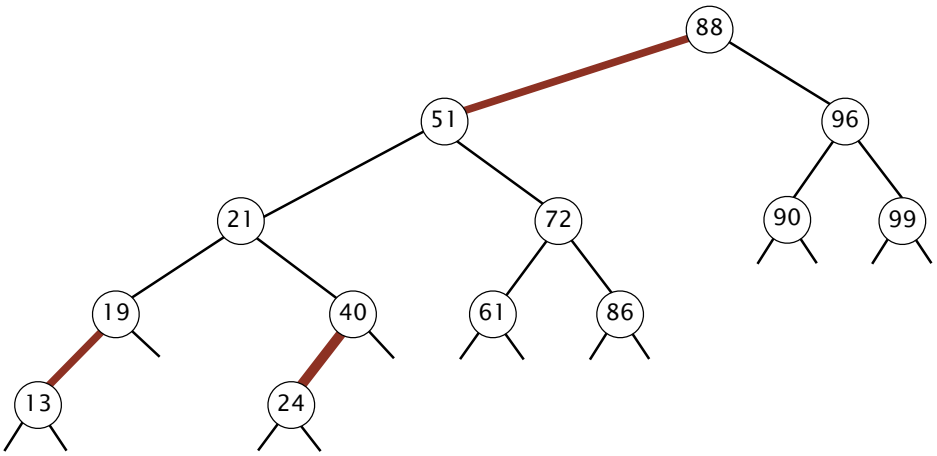
(c) $\sim 3 \log_3 N$

To sink a key, we repeatedly exchange it with its largest child (if the is less than the largest child). Identifying whether a key is less than its largest child requires 3 compares in the worst case. Since the height of a 3-heap is $\sim \log_3 N$, the total number of compares in the worst case is $\sim 3 \log_3 N$.

5. **Red-black BSTs.**

88	51	96	21	72	90	99	19	40	61	86	13	24

To insert 99, we do two color flips, followed by one left rotation. Here is a drawing of the resulting red-black BST.



6. **Problem identification.**

O. Given any array of N distinct integers, determine whether there are three integers that sum to exactly zero in time proportional to $N^{1.5}$.

Conjectured that no such algorithm exists.

P. Given any array of N distinct integers, determine whether there are three integers that sum to exactly zero in time proportional to N^2.

Discussed in precept.

P. Implement a FIFO queue with a constant amount of memory plus two LIFO stacks, so that each queue operation uses a constant amortized number of stack operations.

Discussed in lecture.

P. Given any left-leaning red-black BST containing N keys, find the largest key less than or equal to a given key in logarithmic time.

This is the floor function.

I. Design a priority queue implementation that performs *insert*, *max*, and *delete-max* in $\sim \frac{1}{3} \lg N$ compares per operation, where N is the number of comparable keys in the data structure.

This would violate the $\sim N \lg N$ lower bound for sorting because you can sort an array by inserting N keys into a maximum-oriented priority queue and deleting the maximum N times.

P. Given any array of N keys containing three distinct values, sort it in time proportional to N and using only a constant amount of extra space.

Discussed in lecture as the Dutch National Flag problem. One solution is Dijkstra's 3-way partitioning algorithm.

O. Design a practical, in-place, stable, sorting algorithm that guarantees to sort any array of N comparable keys in at most $\sim N \lg N$ compares.

Described in lecture as the holy sorting grail.

I. Impossible

P. Possible

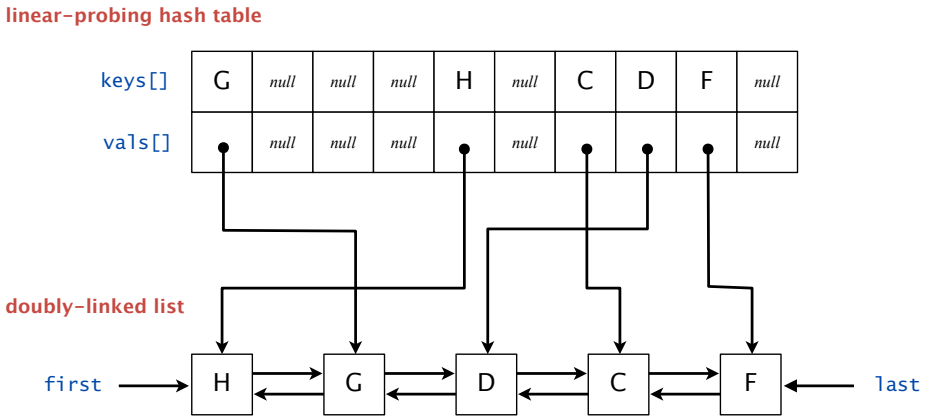
O. Open

7. **LRU cache.**

(a) We use two data structures:

- A doubly-linked list containing the N keys in the cache, with the most-recently cached key at the front and the least-recently cached key at the end.
- A symbol table containing the N keys in the cache, where the value is a reference to the node in the doubly-linked list containing that key.

To achieve the performance requirement, we use a linear probing hash table of capacity $2N$ for the symbol table. (A separate-chaining hash table would also work.)



(b) `inCache(Key key):`

- Use the symbol table to determine whether the key is in the cache.

(c) `cache(Key key):`

- If the key is already in the cache, move the corresponding linked list node from its current position to the front of the list.
- If the key is not already in the cache
  - remove the last node from the linked list (and remove the corresponding key from the symbol table)
  - add a new node to the front of the linked list with the given key (and add a corresponding entry to the symbol table)

8. **Detecting a duplicate.**

(a) The main idea is to consider the N keys in ascending order, so that duplicate keys are adjacent. This is similar to the multiway merging algorithm on pp. 321–322 of the textbook.

- Scan through adjacent entries in each of the k sorted array to check for any duplicate key within one of the original sorted arrays. If a duplicate is detected, stop.
- Initialize a red-black BST with k key-value pairs, where the key is the first (smallest) key in the ith sorted array and the value is the index i of the array.
- Repeat until the BST is empty or a duplicate is detected:
  - Delete the minimum key from the BST and let i be the index of the array associated with the deleted key.
  - If the next remaining key from array i is not already in the BST, add the key and associate it with the value i.
  - Otherwise, stop (duplicate detected).

(b) The amount of space is proportional to k because the BST has at most k keys and we need to maintain one index into each of the k sorted arrays.

The total running time is proportional to $N \log k$. The bottleneck operations are *insert* and *delete-min* in a red-black BST. The cost per operation is $\log k$ because the BST has at most k keys.

An alternate solution is to use a binary heap instead of a red-black BST. Some care is needed with this approach to ensure that you know from which sorted array the smallest key came.