COS 226	Algorithms and Data Structures	Spring 2003

Midterm Solutions

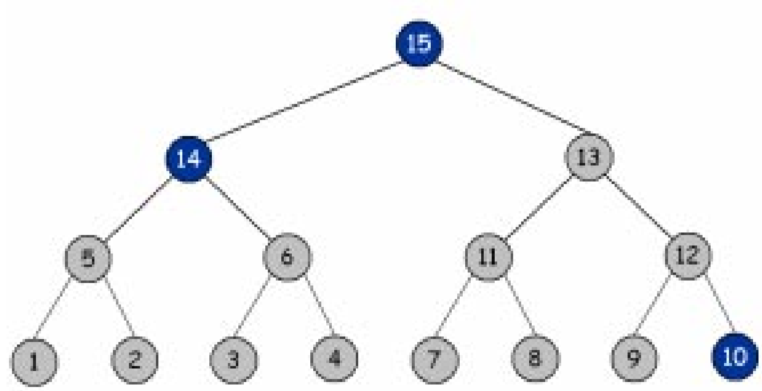
1. **Sorting algorithms.**

   0 4 2 5 7 10 1 3 6 8 9

   - Insertion: the algorithm has sorted the first 12 strings, but hasn't touched the remaining 22 strings.
   - Bubble: the smallest 12 strings are in their final sorted order. `jam` was bubbled down so it's not selection sort.
   - LSD: the strings are sorted on their last character.
   - MSD. The strings are sorted on their first character.
   - Shellsort: The file is 4- and 13-sorted.
   - 3-way radix quicksort: after 3-way partitioning on the `j` in `jam`, all smaller keys are in the top piece, all larger keys are in the bottom piece, and all keys that begin with `j` are in the middle piece.
   - Heapsort: the first phase of heapsort puts the keys in reverse order in the heap.
   - Mergesort: the algorithm has sorted the first 17 strings and the last 17 strings. One final merge will put the strings in sorted order.
   - Quicksort: after partitioning on `jam`, all smaller keys are in the top piece, all smaller keys are in the bottom piece.
   - Selection: the smallest 15 strings are in their final sorted order. `jam` didn't move so it's not bubble sort.

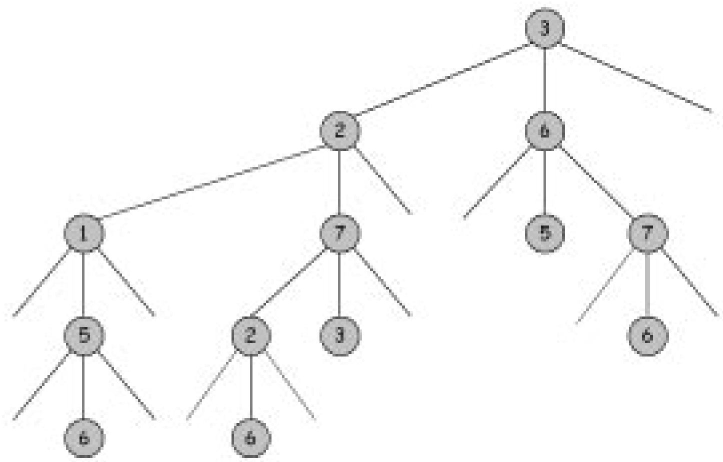
2. **Heaps.**

   2



3. **Tries.**

   156, 273, 365, 376



4. **Choosing the right algorithms and data structures.**

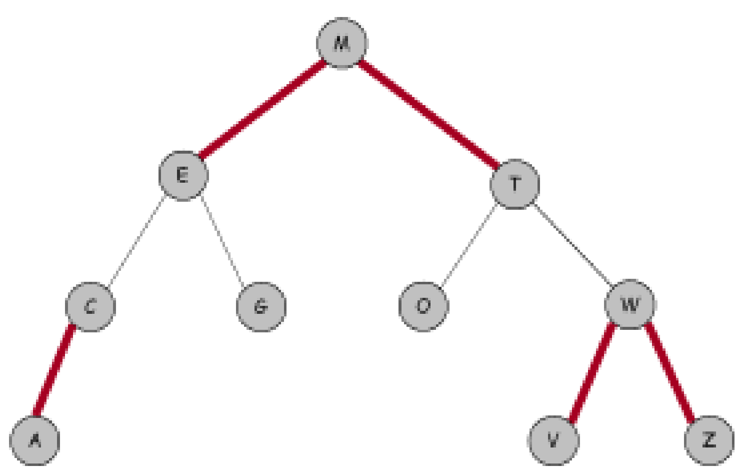
   (a) What is the primary reason to use a binomial queue instead of a binary heap?
   Faster join

   (b) What is the primary reason to use a randomized BST instead of a binary heap?
   Faster search

   (c) What is the primary reason to use double probing instead of linear probing?
   Achieve same search times with less memory

   (d) What is the primary reason to use the Boyer-Moore right-to-left scan algorithm instead of the Knuth-Morris-Pratt algorithm?
   Faster average-case search

5. **Red-black trees.**



6. **Programming assignments.**

   The inner loop gets executed N^3 times. It consists of two additions and one comparison; the innermost for loop also does one increment and one comparison. This is a total of $5N^3$ instructions. The outer and middle loops are inconsequential – $O(N)$ and $O(N^2)$ instructions, respectively.

   (a) Estimate how many seconds it will take (in the worst case) to solve a problem of size N = 1,000?

   5 seconds

   (b) Of size N = 10,000?

   5,000 seconds

7. **Programming assignments.**

   There are many possible solutions.

   Hashing (similar to Assignment 3). Insert all of the integers `a[k]` in a symbol table. Then, enumerate over all pairs i and j to see if `(a[i] + a[j] + a[k] == 0)` for some k. To check this, search for `-(a[i] + a[j])` in the symbol table.

```
for (k = 0; k < N; i++)
   Insert a[k] into a symbol table

for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      Search for -(a[i] + a[j]) in the symbol table
      If found return 1

return 0
```

   For the symbol table, use a linear probing hash table with capacity $2N$. Assuming you have a decent hash function, each search and insert takes $O(1)$ time. The algorithm requires $O(N^2)$ time and $8N$ extra bytes of memory. You could use a BST instead of a hash table; with a splay tree, the running time would be $O(N^2 \log N)$ and it would use $12N$ extra bytes of memory.

Sorting (similar to Assignment 1). First sort the integers `a[k]` in increasing order. Then, enumerate over all pairs i and j to see if `(a[i] + a[j] + a[k] == 0)` for some k. To check this, binary search for `-(a[i] + a[j])` in the sorted array.

```
sort(a, 0, N - 1);

for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      Binary search for -(a[i] + a[j])
      If found, return 1

return 0
```

Sorting takes $O(N \log N)$ time; each search takes $O(\log N)$ time using binary search. The total running time is dominated by the N^2 searches and is $O(N^2 \log N)$. Only a constant amount of extra space is needed, e.g., with heapsort and a non-recursive binary search.

Novel sorting based algorithm. Here's a nice idea to get an algorithm that runs in $O(N^2)$ time while only using $O(1)$ extra space. First sort the integers `a[k]` in increasing order (using heapsort or insertion sort to avoid any extra memory). Then enumerate over all k and try to find i and j such that `a[i] + a[j] + a[k] == 0`. Scan from the left to find `i` and from the right to find `j`. Because of the sorted ordering, you can advance either `i` or `j` according to whether the sum `a[i] + a[j] + a[k]` is positive or negative.

```
sort(a, 0, N - 1);

for (k = 0; k < N; k++)
   i = 0;
   j = N-1;
   while(i <= j)
      sum = a[i] + a[j] + a[k];
      if      (sum <  0) i++;
      else if (sum >  0) j--;
      else    return 1;

return 0
```