CS 61B: Practice for the Final Exam

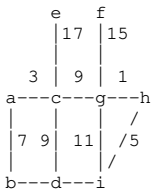
We will cover some of these questions for review, but please attempt these questions before that comes to pass. Starred problems are particularly difficult.

Warning: Midterm 1 topics are absent here, but will reappear on the final.

[1] Given an array containing the digits 71808294, show how the order of the digits changes during each step of [a] insertion sort, [b] selection sort, [c] mergesort, [d] quicksort (using the array-based quicksort of Lecture 31, and always choosing the last element of any subarray to be the pivot), and [e] heapsort (using the backward min-heap version discussed in Lecture 30). Show the array after each swap, except in insertion sort. For insertion sort, show the array after each insertion.

[2] Some sorting methods, like heapsort and array-based quicksort, are not naturally stable. Suggest a way to make _any_ sorting algorithm stable by extending the keys (making them longer and adding extra information).

[3] Consider the graph at right.



[a] In what order are the vertices visited using DFS starting from vertex a? Where a choice exists, use alphabetical order. What if you use BFS?

[b] A vertex x is "finished" when the recursive call DFS(x) terminates. In what order are the vertices finished? (This is different from the order in which they are visited, when DFS(x) is called.)

[c] In what order are edges added to the minimum spanning tree by Kruskal's algorithm? List the edges by giving their endpoints.

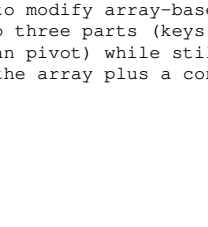
[4] [a] How long does it take to determine if an undirected graph contains a vertex that is connected to no other vertices [i] if you use an adjacency matrix; [ii] if you use adjacency lists.

[b] Suppose we use DFS on a binary search tree, starting from the root. The edge to a left child is always traversed before an edge to the right child. In what order are the nodes visited? Finished?

[c] An undirected graph contains a "cycle" (i.e., loop) if there are two different simple paths by which we can get from one vertex to another. Using breadth-first search (not DFS), how can we tell if an undirected graph contains a cycle?

[d] Recall that an undirected graph is "connected" if there is a path from any vertex to any other vertex. If an undirected graph is not connected, it has multiple connected components. A "connected component" consists of all the vertices reachable from a given vertex, and the edges incident on those vertices. Suggest an algorithm based on DFS (possibly multiple invocations of DFS) that counts the number of connected components in a graph.

[5] What does the splay tree at right look like after:



[a] maxElement()

[b] insertItem(4.5)

[c] findElement(10)

[d] remove(9)

Start from the _original_ tree, not the tree resulting from the previous operation.

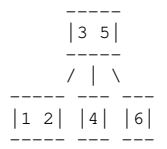
[6] Consider the quick-union algorithm for disjoint sets. We know that a sequence of n operations (unions and finds) can take asymptotically slightly more than linear time in the worst case.

[a] Explain why if all the finds are done before all the unions, a sequence of n operations is guaranteed to take O(n) time.

[b] * Explain why if all the unions are done before all the finds, a sequence of n operations is guaranteed to take O(n) time.

[7] [a] Suggest a sequence of insertion operations that would create the binary tree at right.

[b] Suggest a sequence of operations that would create the 2-3-4 tree at right. You are allowed to use removal as well as insertion.



[8] Suppose an application uses only three operations: insertItem(), findElement(), and remove().

[a] Under what circumstances would you use a splay tree instead of a hash table?

[b] Under what circumstances would you use a 2-3-4 tree instead of a splay tree?

[c] Under what circumstances would you use an unordered array instead of a 2-3-4 tree?

[d] Under what circumstances would you use a binary heap instead of an unordered array?

[9] [a] * Suppose we are implementing a binary heap, based on reference-based binary trees (_not_ arrays). We want to implement a deleteRef() operation which, given a _reference_ to a node in the tree, can delete that node (and the item it contains) from the heap while maintaining the heap-order property--even if the node isn't the root and its item isn't the minimum. deleteRef() should run in O(log n) time. How do we do it?

[b] Building on your answer to the previous question, explain how to combine a min-heap and max-heap (both using reference-based binary trees) to yield a data structure that implements insertItem(), deleteMin(), and deleteMax() in O(log n) time. Hint: You will need inter-heap pointers. Think of how you deleted edges in Project 3, for example.

[c] How can we accomplish the same thing if we use array-based heaps? Hint: Add an extra field to the items stored in each array.

[10] * Recall that the linked-list version of quicksort() puts all items whose keys are equal to the pivot's key into a third queue, which doesn't need to be sorted. This can save much time if there are many repeated keys. The array-based version of quicksort() does not treat items with equal keys specially, so those items are sorted in the recursive calls.

Is it possible to modify array-based quicksort() so that the array is partitioned into three parts (keys less than pivot, keys equal to pivot, keys greater than pivot) while still being in-place? (The only memory you may use is the array plus a constant amount of additional memory.) Why or why not?

[11] Suppose we wish to create a binary heap containing the keys D A T A S T R U C T U R E. (All comparisons use alphabetical order.)

[a] Show the resulting min-heap if we build it using successive insertItem() operations (starting from D).

[b] Show the resulting min-heap if we build it using bottomUpHeap().

[12] Suppose we modify the array-based quicksort() implementation in the Lecture 31 notes to yield an array-based quickselect() algorithm, as described in Lecture 35. Show the steps it would use to find the median letter in D A T A S T R U C T U R E. (The median in this case is the 7th letter, which would appear at array index 6 if we sorted the letters.) As in Question [1], choose the last element of any subarray to be the pivot, and show the array after each swap.

[13] C++ objects can be declared on the stack, on the heap, or in static storage, just like structures. Consider the following C++ code.

```
class ListNode {
public:
  int number;
  ListNode *next;
};

class List {
public:
  ListNode *head;
}
```

```
void dumbRoutine(int i, int j) {

  List myList;
  ListNode node2;

  myList.head = new ListNode();
  myList.head->number = i;
  myList.head->next = &node2;
  myList.head->next->number = j;
}
```

[a] If a program calls dumbRoutine() frequently, is the program certain to suffer from a memory leak?

[b] If a program calls dumbRoutine() frequently, is the program thereby susceptible to having some of its data inadvertently corrupted?

[c] If the declaration "List myList;" is moved outside of dumbRoutine(), so that myList is a global variable, the answers to both [a] and [b] change. Explain why.

[14] What's wrong with this heap allocation strategy? We keep all allocated chunks on the heap contiguous. Whenever a free() or delete operation occurs, we find the position of the newly freed chunk, then we move all the chunks to the right of it leftward to fill the gap. That way, no fragmentation occurs at all! Allocation requests are satisfied using the first free memory, to the right of all the allocated chunks.

```
 ---------------------------------      ---------------------------------
 |&&***&&*******&&**&&****&&......  =>  |&&***&&**&&****&&................
 ---------------------------------      ---------------------------------
         ^free()
```

[15] What are the values of the four elements of the array x after the following C code is executed?

```
int x[4];
int *p;
p = &x[1];
p[2] = 20;
p = &p[1];
*p = 8;
*x = 33;
```

[16] Suppose our radix-sort algorithm takes exactly n+r seconds per pass, where n is the number of keys to sort, and r is the radix (number of queues). To sort 493 keys, what radix r will give us the best running time? With this radix, how many passes will it take to sort 420-bit keys? To answer this question, you'll need to use calculus (and a calculator), and you'll need to remember that log2 r = (ln r) / (ln 2).

[17] Suppose that while your computer is sorting an array of objects, its memory is struck by a cosmic ray that changes exactly one of the keys to something completely different. For each of the following sorting algorithms, what is the _worst-case_ possibility? For each, answer [x] the final array won't even be close to sorted, [y] the final array will have just one or two keys out of place, or [z] the final array will consist of two separate sorted subsets, one following the other, plus perhaps one or two additional keys out of place.

[a] Insertion sort
[b] Selection sort
[c] Mergesort
[d] Radix sort

Programming questions.

[18] Rewrite the quicksort() method from the Lecture 31 notes in C++. The signature should appear as follows.

```
void quicksort(Comparable *a, int length) {
```

Note one big change from the Java version: instead of passing an array, a low index, and a high index, we pass only an array and its length (the number of items to be sorted). The call above should sort the elements a[0] through a[length - 1]. The reason only two parameters are needed for the recursion is because the Java call quicksort(a, x, y) can be replaced by the C++ call quicksort(&a[x], y-x+1). Be careful to make sure that your array indices are correct in your C++ version.

[19] Implement tree sort (as described in the sorting video) in Java. Assume your treeSort() method's only input parameters are the number of items and a complete (perfectly balanced) BinaryTree of depth d in which each leaf has an item; hence, there are 2^d items to sort. All internal nodes begin with their item field set to null. Use the data structures below (in which each node knows its left and right child), not a general tree.

Your algorithm should never change a node reference; only the items move. The centerpiece of your algorithm will be a method that fills an empty node by (i) recursively filling its left and right children if they're empty, and (ii) choosing the smaller of its children's items, which is moved up into the empty node. treeSort() will repeatedly: (i) apply this method to the root node to find the smallest item remaining in the tree, (ii) pluck that item out of the root node, leaving the root empty again, and (iii) put the item into an array. Your treeSort() should allocate and return that array.

```
class BinaryTreeNode {
  Comparable item;
  BinaryNode leftChild;
  BinaryNode rightChild;
}
```

```
class BinaryTree {
  BinaryTreeNode root;
}
```