CS 240
Midterm Exam
Model Solutions

Date: February, 27, 2007
Time: 4:30pm-6:30pm
Instructor: J.P. Pretti

No calculators or other aids are permitted

Answer each question in the space provided. You have some freedom in how you express your answers to many questions. What matters most is that what you write convinces the marker that you understand a good solution to the problem.

Questions will not be interpreted. Proctors will only confirm or deny errors in the questions. If you consider the wording of a question to be ambiguous, state reasonable assumptions clearly and proceed to answer the question to the best of your ability.

Name (please print): _____

Signature: _____

ID: _____

Question	Out Of	Mark
1	9	
2	5	
3	6	
4	8	
5	8	
6	8	
7	4	
Total	48	

There are 11 pages including this cover page. Extra blank paper is available for rough work upon request.

1. [9 marks] *Short Answer.* Answer each of the following short answer questions using at most one or two sentences. They are not meant to be difficult or tricky.

   (a) The first recursive solution to the maximum subsequence sum problem discussed in class is $O(n \lg n)$ in the worst-case. So, what is wrong with the following analysis?

   $\Theta(n)$ recursive calls are made in every case. Each recursive call is $O(n)$. Hence the worst-case runtime is $\Theta(n^2)$.

   n varies over the recursive calls and so we can only use the above observations (which are both true) to observe that the worst-case runtime is $O(n^2)$. A more careful analysis shows that it is $o(n^2)$, or $\Theta(n \log n)$ to be precise.

   (b) What is a *priority queue*?

   An ADT storing a collection of prioritized items supporting arbitrary insertion but removal only of the item of highest priority.

   (c) Give a significant advantage of using an unordered linked-list over a heap when implementing a priority queue.

   Insertions can be performed in $O(1)$ time in the worst-case.

   (d) Under what circumstances is heap-sort the best choice of sorting algorithm?

   It is ideal when an in-place algorithm with a $O(n \lg n)$ worst-case runtime is required.

(e) When would the leftmost-child right-sibling representation of general trees probably use less space than implementing a node using a fixed-size array of pointers to its children?

When the number of children varies considerably from node to node. Using a fixed-size array of pointers to children means that lots of extra space is wasted for "missing children" in this case. The leftmost-child right-sibling representation only requires a single reference for each child in the general tree it represents.

(f) Why don't hashing techniques easily support efficient implementations of ADT Table operations such as `returnMax`?

Good hash functions map keys to buckets in unpredictable random ways. In particular, they are not placed in buckets "in order".

(g) How do we measure the efficiency of an algorithm in the *external memory model*?

We count the number of times a page/block on disk is accessed.

(h) What is *primary clustering* and why is it a problem for linear probing?

It is the phenomenon where the largest blocks of consecutive non-empty buckets are the most likely to grow yet larger. It slows down the runtime of operations using linear probing because longer and longer portions of probe sequences need to be followed.

(i) If the keys to be inserted into a hash table are strings consisting of animal names, why is it a bad idea to use only the first character of the animal name as the basis for a hash function?

Some letters are much more likely to be the first character than others and so such a hash function would likely not be uniform and spread keys evenly across buckets.

2. [5 marks] *Asymptotic Notation.*

(a) [1 mark] Give a function that is $\Omega(n)$ but not $O(n)$.

n^2

(b) [4 marks] Prove that $n + n \lg n$ is $\Theta(n \lg n)$ from first principles (i.e. using the formal definition of Θ-notation).

$n + n \lg n \geq n \lg n$ for all $n \geq 1$ so $n + n \lg n$ is $\Omega(n \lg n)$. Also, $n + n \lg n \leq n \lg n + n \lg n = 2n \lg n$ for all $n \geq 2$ so $n + n \lg n$ is $O(n \lg n)$. Together, this proves the required.

3. [6 marks] *Algorithm Analysis.* Express the runtime of the following pseudocode fragments using Θ-notation in terms of n. You do not *have to* justify your answers but part marks will only be awarded for incorrect answers if you correctly indicate the runtime of portions of the code fragment.

   (a) [4 marks]

```
int x = 1
while (x < n)
    for (int i=0; i < n; i++)
        output "I Love CS 240"
        int y = 1
        while (y < n)
            output "Go Leafs Go"
            y = y * 2;
    x = x * 2;
for (int i=0; i < n; i++)
    for (int j=0; j < n; j++)
        output "Go Raptors Go"
```

   The control variables of the first three loops are independent and so the product of their runtimes is the runtime of the "top" portion of the fragment. The for loop is clearly $\Theta(n)$ and both while loops iterate as many times as an integer initially 1 can be doubled before reaching or exceeding n. This is $\Theta(\lg n)$ times. Hence the "top" portion is $\Theta(n \lg^2 n)$.

   The "bottom" portion consists of two simple nested linear for loops and so has a runtime of $\Theta(n^2)$.

   Since polynomials always dominate logarithms to any positive power, the "bottom" portion dominates and the overall runtime is $\Theta(n^2)$.

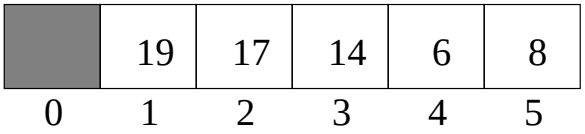
   (b) [2 marks]

```
void silly(int n)
    if (n > 1)
        for (int i=0; i < n; i++)
            output "looping just for fun"
        silly (n/2)
        for (int i=0; i < n; i++)
            output "looping for more fun"
        silly (n/2)
        for (int i=0; i < n; i++)
            output "looping for even more fun"
```

   $\Theta(n)$ work is performed between two recursive calls on half the original input size. This is identical to merge-sort and so the overall runtime is $\Theta(n \lg n)$.

4. [8 marks] *Heaps.* For this question, assume `heap` is a max heap storing n distinct integer keys - it is an array storing the largest integer at `heap[1]`.

   (a) [2 marks] Assume `heap` is empty and then the keys 14, 6, 17, 8, 19 are inserted (in that order) using the heap insertion algorithm. Draw a picture of `heap` (as an array) after the insertions.



   (b) [3 marks] Describe how the following operation can be implemented in $O(\lg n)$ time. Answer using either pseudocode or one or two simple sentences.

```
//pre:  1 <= index <= n
//post: the heap contains newKey and the keys originally at
//      indices 1..index-1 and index+1..n
void changeKeyAtIndex(int index, int newKey)
```

   change the key at position `index` *to* `newKey` *and sink or swim (at most one is necessary) as needed.*

   (c) [2 marks] Give the average height of a subtree rooted at an internal node of a heap using Θ-notation. Informally justify your answer.

   $\Theta(1)$. *Bottom-up heap construction is proportional to the sum of the heights over all subtrees rooted at an internal node which we saw to be* $O(n)$ *in class. Hence the average height of a subtree rooted at an internal node is* $\Theta(1)$.

   (d) [1 mark] If `heap` stores 842 keys, how many positions *might* contain the smallest key? You do not need to justify your answer.

   421. This is precisely the number nodes with 0 internal node children.

5. [8 marks] *Sorting.*

(a) [2 marks] Explain how to sort $n^{0.9}$ comparable keys in $O(n)$ time.

Sort using mergesort or heapsort ($\Theta(n \lg n)$ worst-case runtime). This will take $\Theta(n^{0.9} \lg n^{0.9})$ time which is $\Theta(0.9n^{0.9} \lg n)$ and $\Theta(0.9n^{0.9}n^{0.9})$ or $\Theta(n)$ because polynomials always dominate logarithms to any positive power.

(b) [2 marks] The expected number of key comparisons required by quick-sort assuming pivots are chosen randomly can be described by the following recurrence relation.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= n + 1 + \frac{1}{n}\sum_{k=1}^{n}(T(k-1) + T(n-k)). \end{aligned}$$

Explain the meaning of each of the following:

i. $n + 1$

$n + 1$ is the expected number of comparisons performed by the partition algorithm on input of size n.

ii. $T(k-1)$

$T(k-1)$ is the expected number of comparisons performed by quick-sort when called on the $k-1$ keys to the left of a pivot at position k.

(c) [2 marks] The *mode* of a set is the most frequently occuring element of the set. Give a $O(n+m)$ worst-case algorithm to find the mode of n non-negative integers all less than m. Assume the mode is unique.

```
b = an array of m integers initialized to 0
for (int i=0; i < a.length; i++)
    increment b[a[i]]
mode = 0
for (int i=0; i < b.length; i++)
    if (b[i] > b[mode])
        mode = i
return mode
```

(d) [2 marks] Prove that it is impossible to implement a priority queue in the comparison model such that

- `removeMax` is $O(1)$ in the worst-case, and
- a priority queue containing n items can constructed in $O(n)$ time in the worst-case.

We proceed by way of contradiction. Assume that `removeMax` *has a worst-case runtime of* $O(1)$ *and a priority queue containing* n *items can be constructed in* $O(n)$ *time in the worst-case. One can then sort* n *items in* $O(n)$ *time by constructing a heap from the* n *items and calling* `removeMax` n *times. This contradicts the* $\Omega(n \log n)$ *theoretical lower bound on sorting in the comparison model and so our assumption must be wrong proving the required.*

6. [8 marks] *Hashing.*

   (a) [4 marks] Answer each of the following questions with an exact number. Assume a hash table with 100 buckets stores 50 keys (i.e. $m = 100$ and $n = 50$). No justification is required.

      i. Exactly how many key comparisons are performed *in the worst-case* to *search* for a key if collisions were resolved by *separate-chaining*?

         50 - *the longest a chain can be*

      ii. Approximately how many key comparisons are performed *on average* to *remove* a key if collisions were resolved by *separate-chaining*?

         $1\frac{1}{4}$ - *1 for finding the key to delete plus scanning half an average-sized chain*

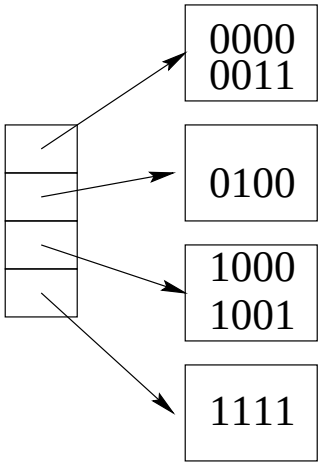
      iii. Exactly how many key comparisons are performed *in the best-case* to perform an *unsuccessful search* if collisions were resolved by *double hashing*?

         0 - *hash to an empty bucket*

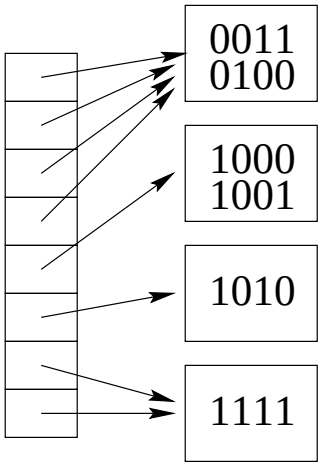
      iv. Exactly how many key comparisons are performed *in the worst-case* to *insert* if collisions were resolved by *double hashing*?

         0 - *no key comparisons are performed although there could be as many as* 50 *probes in an attempt to find an empty bucket*

(b) [2 marks] Show the extendible hash table that results after the hash value 0000 is inserted into the following table. Assume that pages hold up to $m = 2$ values.



(c) [2 marks] Show the extendible hash table that results after 1010 is inserted into the following table. Assume that pages hold up to $m = 2$ values.



7. [4 marks] *Data Structure Design.*

Give a data structure that can be used to implement ADT Midterm which stores integer keys and supports the following operations.

```
//pre:  this does not already contain key x
//post: this contains key x
void insert(int x);

//pre:  this contains key x
//post: this does not contain key x
void remove(int x);

//post: returns true iff this contains key x
boolean search(int x);

//pre:  this is not empty
//post: the largest key is returned and removed
int removeMax();
```

You must ensure that

- `search` is $O(1)$ in the average-case,
- `insert` and `remove` are $O(\lg n)$ in the average-case, and
- `removeMax` is $O(\lg n)$ in the worst-case.

Use pictures and words. You do not need to describe the associated algorithms or justify that your approach can be used to meet the runtime bounds.

Store all keys in both a hash table and a heap. Keys in the heap point to their location in such a way that they can be removed from the hash table in $O(1)$ time given their location. This can be an index if closed hashing is used and a reference to a node in a doubly-linked chain if separate-chaining is used. Keys in the hash table store their location (index) in the heap.