CSE 332: Data Abstractions Autumn 2015

Practice Midterm Exam Solutions

Name:	Sample Solutions
ID #:	1234567
TA:	The Best
Section:	A9

INSTRUCTIONS:

- You have **50 minutes** to complete the exam.
- The exam is closed book. You may not use cell phones or calculators.
- All answers you want graded should be written on the exam paper.
- If you need extra space, use the back of a page.
- The problems are of varying difficulty.
- If you get stuck on a problem, move on and come back to it later.
- It is to your advantage to read all the problems before beginning the exam.

Problem	Points	Score	Problem	Points	Score
1	8		6	15	
2	3		7	8	
3	3		8	15	
4	6		9	10	
5	6		10	10	
			Σ	84	

One Liners.

This section has questions that require very short answers. To get full credit, you should answer in no more than one sentence per question.

1. X **Marks The Spot** [8 points]

For each of the following rows, circle each option on the right that is true for the function on the left and X each option that is false for the function on the left.

$\sum_{i=0}^{n} i^2$	~~$\mathcal{O}(n)$~~	~~$\mathcal{O}(n^2)$~~	[✓ $\mathcal{O}(n^3)$]	[✓ $\mathcal{O}(n^4)$]
3^n	[✓ $\mathcal{O}(3^n)$]	~~$\mathcal{O}(3^{n/2})$~~	[✓ $\Omega(3^n)$]	[✓ $\Omega(3^{n/2})$]
$10000n^{25}$	~~$\mathcal{O}(n)$~~	[✓ $\mathcal{O}(n^{26})$]	[✓ $\Omega(n)$]	~~$\Omega(n^{26})$~~
$\log n$	[✓ $\mathcal{O}(\log n)$]	~~$\mathcal{O}(\log\log n)$~~	[✓ $\Omega(\log n)$]	[✓ $\Omega(\log\log n)$]
n^n	~~$\mathcal{O}(20000^n)$~~	[✓ $\Omega(20000^n)$]	~~$\Theta(20000^n)$~~	[✓ $\Theta(n^n + 20000^n)$]

2. `find` The Error [3 points]
Consider a *dictionary* implemented using a sorted array. Consider the following argument:

> We claim that `find()` is amortized $\mathcal{O}(1)$ in this implementation. Consider a sequence of $\lg(n)$ operations. Each operation takes $\lg(n)$ time; so, the amortized cost of each operation is $\frac{\lg(n)}{\lg(n)} = \mathcal{O}(1)$.

This argument is faulty. Explain why it's faulty and give a correct (tight) asymptotic bound for the actual amortized cost.

Solution: This argument is faulty because it is using an incorrect formula to compute the amortized cost. The argumnt is using the formula:

$$\frac{\text{Time per operation}}{\text{Number of operations}} \rightarrow \text{Amortized cost}$$

However, the correct formula is:

$$\frac{\text{Cost of first (usually expensive) operation} + \text{Cost of next X - 1 operations}}{\text{X}} \rightarrow \text{Amortized cost}$$

. . . where X is the number of operations.
Using this formula, and if we let $X = \lg n$ (which, by the way, is a very weird amount of operations to do, we can see that the amortized cost be:

$$\frac{\lg n + (\lg n - 1) \cdot \lg n}{\lg n} = \frac{(\lg n)^2}{\lg n} \rightarrow \mathcal{O}(\lg n)$$

3. It's $\mathcal{O}$lementary, My Dear Watson [3 points]
Let $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$ be increasing functions with $f(n) \neq g(n)$ for any $n \in \mathbb{N}$. Consider the statement:

$$f(n) \in \Omega(g(n)) \text{ and } g(n) \in \Omega(f(n))$$

Is this statement *always*, *sometimes*, or *never* true? Explain your answer in one sentence.

Solution: This statement is sometimes true.
For example, the statement is true if $f(n) = 2n$ and $g(n) = 3n$, but is false if $f(n) = 1$ and $g(n) = n$.

4. The-ta Knows Best! [6 points]

For each of the following, give a $\Theta(-)$ bound, in terms of n, for the *worst case runtime* of the method.

(a) (2 points)

```java
int hello(int n) {
   if (n == 0) {
      return 0;
   }
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n * n; j++) {
         System.out.println("HELLO");
      }
   }
   return hello(n − 1);
}
```

Runtime

$\Theta(n^4)$

(b) (2 points)

```java
void whee(int n) {
   for (int i = 1; i < n; i *= 2) {
      for (int j = 1; j < n; j *= 3) {
         System.out.println("WHEE!");
      }
   }
   for (int k = n/2; k < n; k++) {
         System.out.println("WOAH!");
   }
}
```

Runtime

$\Theta(n)$

(c) (2 points)

```java
void flipflop(int n, int sum) {
   if (n > 10000) {
      for (int i = 0; i < n * n * n; i++) {
         sum++;
      }
   }
   else {
      for (int i = 0; i < n * n * n * n; i++) {
         sum++;
      }
   }
}
```

Runtime

$\Theta(n^3)$

5. Analysis [6 points]
For each of the following, determine the runtime of the algorithm/operation:

Runtime

(a) (2 points) Worst case `find` in a B-Tree

$\mathcal{O}(\lg(L) + \lg(M)\log_M n)$

(b) (4 points) Best case `insert` in a hash table with size n and a current $\lambda = 1$ if the collision resolution strategy is:

Runtime

- Separate Chaining

$\mathcal{O}(1)$

Runtime

- Double Hashing

$\mathcal{O}(n)$

6. Choose The Data Structure [15 points]

For each of the following, decide which data structure is most appropriate to solve the problem:

AVL Tree	B-Tree	Bit Set	FIFO Queue	Hash Table
Linked List	Heap	Stack	MTF List	Vanilla BST

(a) (3 points) An operating system needs to schedule when to run each of the current processes.

Solution: We should use a *heap*. Processes in an operating system have different priorities, and we should service them in order of priority.

(b) (3 points) You want to store 1MB of *non-comparable* data and you expect to run the `find` operation very frequently.

Solution: We should use a *hash table*. We can't use any of the trees, because the data is not comparable. Since `find` is a frequent operation, we want it to be $\mathcal{O}(1)$. We don't know anything about the keys other than that they're comparable; so, we can't necessarily use a trie or a bitset.

(c) (3 points) You want to keep track of ASCII characters that are allowed in a valid password.

Solution: We should use a *bit set*. There are very few ASCII characters, and most of them are likely to be allowed in a password. So, the bitset is going to be dense and the keys are ints.

(d) (3 points) Google (which has petabytes of data) wants to store search queries in such a way that they can easily find the closest alphabetical query to a new user query.

Solution: We should use a *B-Tree*. Since there is a lot of data, we expect disk accesses to matter; so, a B-Tree is the right choice.

(e) (3 points) A grocery store wants to make a database of their products (they only have about 5000 unique products) so that employees can look up the aisle that a particular product is in by name.

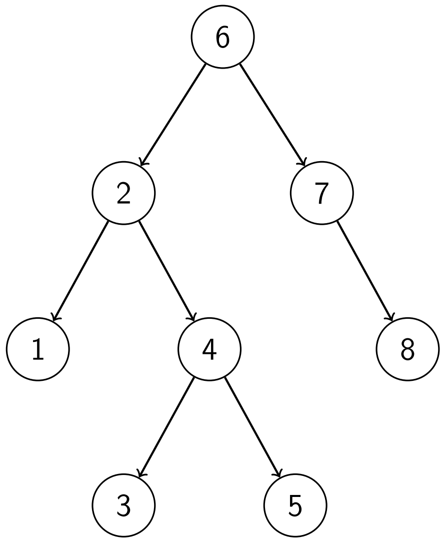
Solution: If we assume that the products are listed by *id number*, then we should use a bit-set (because there are very few products). Otherwise, we should use a hash table, because we want as fast a lookup as possible.

Basic Techniques.

This part will test your ability to apply techniques that have been explicitly identified in lecture and reinforced through sections and homeworks. Remember to show your work and justify your claims.

7. AVL Trees [8 points]
Insert $1, 2, 7, 6, 8, 3, 4, 5$ into an AVL tree *in that order*. You do not have to show your intermediary steps, but no work and a wrong answer will receive no credit.

Solution:



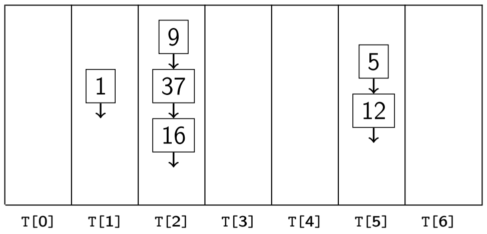
8. Hashing [15 points]
You know in advance that you will never put more than *six* strings into a particular hash table.

(a) (2 points) If you had a choice between 6 and 7 as your table size, which would you choose? Why?

Solution: We should choose 7. Despite knowing that there is a maximum of 6 strings, we don't know precisely what hashing function those strings will end up using. Therefore, to minimize the chances of clustering, we should make the table size a prime number.

(b) (3 points) Insert 1, 5, 9, 12, 37, 16 into your hash table using the hash function $h(x) = x$ and *separate chaining*.

Solution:



(c) (2 points) What is the load factor of your hash table from part (b)?

Solution: The load factor is $\lambda = \frac{\text{num entries}}{\text{num buckets}} = \frac{6}{7}$

(d) (3 points) Insert the same numbers into an *empty version* of your hash table using the hash function $h(x) = x$ and *quadratic probing*.

Solution:

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]
	1	9	37	16	5	12

(e) (3 points) Does your hash table from part (d) have primary clustering? What about secondary clustering?

Solution: It arguably has both primary clustering and secondary clustering. We can see this, because when we insert 16, we go through spots that have several distinct hash codes.

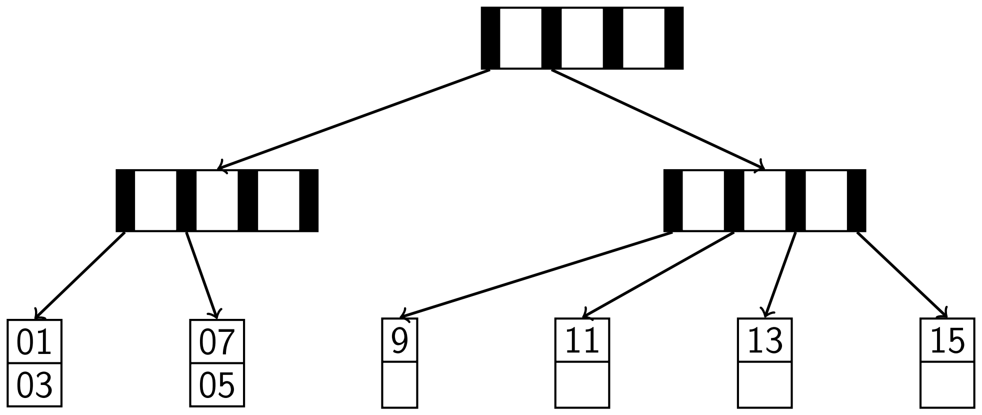
Note that, in general, we do not expect to have primary clustering when we use quadratic probing, but, for example, if the hash table has $\lambda = 1$, there *must* be both primary and secondary clustering.

(f) (2 points) What is the load factor of your hash table from part (d)?

Solution: The load factor is $\lambda = \frac{6}{7}$.

9. **B-Trees** [10 points]

Consider the following "B-Tree":



(a) (2 points) What are $M =$ **4** and $L =$ **2** ?

(b) (5 points) List (and fix, if possible) everything that you can find that is wrong with the above B-Tree.

Solution:

(a) 5 and 7 should be swapped in the second leaf

(b) The leaves containing 9 and 11 should be consolidated; the leaves containing 13 and 15 should be consolidated

(c) The inner nodes are missing the values (left-most inner node should have 5 in the first entry, the right-most inner node should have 13 in the first entry, and the root node should have 9 in the first entry)

(c) (3 points) Suppose that you know the following facts about a computer system:

- 1 page on disk is 1024 bytes
- Keys are 4 bytes
- Pointers are 8 bytes
- Key/Value Pairs are 32 bytes

If you want to use a B-Tree on this system, what should you choose M and L to be?

Solution: Let $p = 1024$ be the page size, $k = 4$ be the key size, $t = 8$ be the pointer size, and $e = 32$ be the leaf entry size.

We also know that $p \geq eL$ and that $p \geq tM + k(M-1)$. We can solve the left-most equation to obtain $L = 32$. We can solve the right-most equation to obtain $\frac{p+k}{k+t} \approx 85.67 \geq M$. We round down to obtain $M = 85$.

A Moment's Thought!

This section tests your ability to think a little bit more insightfully. The approaches necessary to solve these problems may not be immediately obvious. Remember to show your work and justify your claims.

10. Treaps! [10 points]

In this problem, we define and analyze a new data structure called a *Treap*. A *treap* is a combination of a BST (tre-) and a heap (-eap). Operations on treaps are defined as follows:

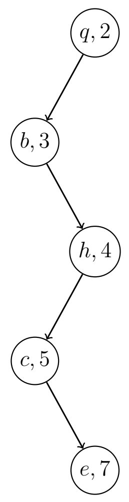
```
void insert(n)
```

- Generate a *random* priority p.
- Do a standard BST insert into the treap of (n, p) completely ignoring the priority.
- Fix the Treap so that the *keys* follow the *BST Property* and the *priorities* follow the *Min-Heap Property*. (Note that the treap *does not* have to obey the *Heap STRUCTURE Property*.)

(a) (6 points) Suppose that we have a random number generator that will generate the following sequence of random numbers: $5, 3, 7, 2, 4, \ldots$

- Insert c, b, e, q, h into a treap in that order.
  Hint: After doing *each* BST insert, use *AVL Rotations* on the inserted element until the Heap Property is satisfied.

  Solution:



- Is your resulting treap an AVL tree? Why or why not?

  Solution: It is not an AVL tree, because it is not balanced.

(b) (4 points) Suppose we implement `deleteMin` on a treap as follows (where `delete` is defined with lazy deletion):

```
int deleteMin() {
   min = findMinKey()
   return delete(min);
}
```

- What is the best case runtime of this operation?

  Solution: The best case runtime is $\mathcal{O}(1)$. Consider the case where the minimum key also has the minimum priority. There would be no nodes to the left of the root, so we would know to return the root immediately.

- What is the worst case runtime of this operation?

  Solution: The worst case runtime is $\mathcal{O}(n)$. Consider the case where the minimum key also has the largest priority, and the tree is pathological (is essentially a linked list). We might potentially need to traverse through the entire treap to find the min key.