

Homework 2

October 4, 2005

Due Date: October 11

1. (10 pts) For both parts, the hash table size is 8. Let

	22	30	45	47	51	73
hash	0	5	5	4	0	0
stepHash	1	3	1	7	5	5

- (a) Show the result of inserting items with keys of 73, 22, 51, 30, 45, and 47 when using chaining to resolve collisions.
- (b) Show the result of inserting items with keys of 47, 22, 51, 30 (in that order) using open addressing with double hashing to resolve collisions.
2. (15 pts) This problem is designed to be sure you understand the definition of expected value and can use it to determine the expected number of times a portion of code is executed. There are practice problems on the webpage like this so you should look at that if you feel like you need more guidance before starting. Here is code for binary search which assumes that A is sorted such that $A[i] \leq A[i + 1]$. The initial call made is `binarySearch(x, A, 0, 14)`.

```
boolean binarySearch(x, A, p, r)
1  if (p == r)
2    return (x == A[p])
3  q = [(p + r)/2]
4  if (x == A[q])
5    return true
6  if x < A[q]
7    binarySearch(x, A, p, q - 1)
8  else
9    binarySearch(x, A, q + 1, r)
```

You are to exactly compute the expected number of times that x is compared to an element of A (in line 2,4 or 6) when searching in array $A[0..14]$ for an element x where for $0 \leq i \leq 14$, x is $A[i]$ with a probability of $1/30$ and with probability $1/2$, x is not in the array. Using the formal definition of expected value, and compute the exact value.

3. (20 points) In order to have an *extensible* array (such as Java's `ArrayList` that has no upper limit on the size), the most efficient implementation is to double the array size copying over the elements in use when the array is full. We study an extensible array here in the context of hash tables but the same basic analysis would hold for Java's `ArrayList` implementation.

Suppose that you have a hashtable with collisions resolved using chaining that is initially set to a size of 2. Whenever the load reaches 2, the hashtable size must be doubled and the table must be rebuilt since the hash values have changed¹. Assume that there are $2^k + 1$ elements inserted

¹For OpenAddressing with a desired load of $1/2$, when the current load is $3/4$, the hashtable should be resized to bring the load back to $1/2$.

(and none deleted) for some integer $k \geq 0$. Compute the *amortized cost* of the doubling operation which is defined as the time complexity of doubling and rebuilding the hash table divided by the number of elements inserted since the last doubling operation.

More generally, amortized complexity is used when an expensive operation is periodically used (typically to perform some reorganization of the data structure). The cost of each such operation is amortized over the public methods called to require the expensive operation to be needed. While we won't have any further discussion of amortized complexity in this course, you can read more about it and the techniques used in Chapter 17 of the text book.

4. (35 points) For the following problem you are to select the data structure(s) to use. The following components should be clearly given in each of your solutions.

- You should very clearly describe your data structure choice, including all decisions about how the data structure is to be applied (e.g. what is used as the key, what is the associated data, what is the table size and collision resolution choice for each hashtable,...).
- You should clearly describe how each of the provided operations will be implemented AND analyze the efficiency for each of the operations. You only need to describe any new methods or variations of methods covered in class that you need. For the methods covered in class (e.g. insertion into a hashtable) you don't need to describe how it or derive the expected time complexity. You can just specify what standard method you are using (e.g. insertion) along with its parameters (e.g. what is given as the key, and what is given as the associated data).
- Briefly discuss your choice of data structures included the hashtable size and the method for collision resolution.

You have been hired as a consultant by University X. They have an enrollment of s students where s is approximately 10,000. Each student has an id number which is a 6 digit number that is unique for that student. In addition, for each student there is a StudentRecord that includes a name, an email address, a mailing address, current courses (initially empty) and a transcript. Each course also has unique id (e.g. E81241FL05). For each course there is a CourseRecord that includes a professor, classroom and class list (initially empty). Each semester, University X teaches approximately t courses where $t = 1000$. The size of course i is denoted by n_i where $5 \leq n_i \leq 300$. Finally, you can assume that each student takes no more than 10 courses in a semester.

You are to design a data structure to implement the following operations in the specified *expected* time. Any operation that must run in expected constant time cannot depend on the size of s , t , or n_i for any i . From the names of the methods it should be clear what is expected of them. For `printSortedRoster` the roster must be sorted alphabetically by the students' names (and also include their id and email address). If you are uncertain about the semantics of any other methods please ask. You should assume that both the `insertStudent`, `removeStudent`, `registerStudent`, and `withdrawStudent` methods are called frequently, whereas the `insertCourse` method is not called very frequently. Also, notice that courses are never removed.

Operation	Expected Time Requirement
<code>insertStudent(int studentID, StudentRecord data)</code>	$O(1)$
<code>removeStudent(int studentID)</code>	$O(1)$
<code>insertCourse(int courseID, CourseRecord data)</code>	$O(1)$
<code>registerStudent(int studentID, int courseID)</code>	$O(1)$
<code>withdrawStudent(int studentID, int courseID)</code>	$O(1)$
<code>printSortedRoster(int courseID)</code>	$O(n_i \log n_i)$ for course i

5. (20 pts) Consider the following problem. You are given arrays A and B which each have n elements. The elements in the arrays are **not** necessarily in sorted order, and each array can contain repeated elements. We define A and B to be *disjoint* if they contain no elements in common. For example, the arrays $[1, 7, 2, 5, 2]$ and $[6, 7, 8, 9, 6]$ are not disjoint because they both contain the number 7. However, the arrays $[2, 1, 7, 2, 5]$ and $[8, 8, 9, 6, 9]$ are disjoint.

In the *array disjointness problem*, the problem is to determine whether arrays A and B are *disjoint*. If the arrays are not disjoint the algorithm should return a pair (i, j) such that $A[i] = B[j]$. Otherwise, the algorithm should return `null`.

Describe an algorithm that solves the array disjointness in $O(n)$ expected time. The algorithm should be described in sufficient detail that, the TA reading the solution could implement it. However, your algorithm description should be as short as you can while satisfying the above. We do not want detailed code! Give a very brief time complexity analysis of your algorithm to convince us that it runs in $O(n)$ expected time.

Challenge Problems:

Only work on these after you have completed the required problems. Note that you can obtain 100% without doing either of these.

5* (2 points)

The following program determines the maximum value in an unordered array $A[0..n-1]$.

```

1 max =  $-\infty$ 
2 (int i = 0; i < n; i++)
3     if  $A[i] > \textit{max}$ 
4         then  $\textit{max} = A[i]$ 
```

What is the expected number of times the assignment in line 4 is executed? (Assume that all numbers in A are drawn randomly from the interval $[0,1]$.) You may find it useful to let s_1, s_2, \dots, s_n be n random variables, where s_i represents the number of times (0 or 1) that line 4 is executed during the i th iteration of the **for** loop. Since $s = s_1 + s_2 + \dots + s_n$, by linearity of expectations $E(s) = E(s_1) + \dots + E(s_n)$.

- 6* (3 points) Generalize your work from Problem 2 to compute the expected number of elements examined when the array has $n = 2^r - 1$ elements (for r an integer) and with probability $1/2$, x is not in the array, and with probability $1/(2n)$, $x = A[i]$ for $i = 0, 1, \dots, n - 1$.