The homework assignment is available at:
`http://www.jennylam.cc/courses/146-s17/homework09.html`

Exercises (a) (11.2-1) Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}$

Solution. Each slot has on expectation n/m keys, and the number of collisions in each slot is the number of pairs of keys, which is

$$\binom{n/m}{2} = \frac{1}{2}\frac{n}{m}\left(\frac{n}{m} - 1\right).$$

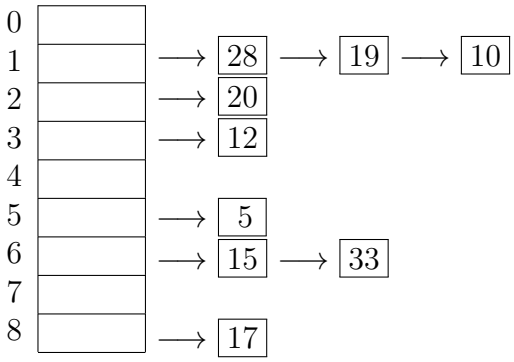
So for a total of m table slots, that's

$$\frac{n}{2}\left(\frac{n}{m} - 1\right)$$

collisions on expectation. •

(b) (based on 11.2-2) Consider a hash table with 9 slots, in which collisions are resolved by chaining. Show the contents of the hash table when the following keys are inserted: 5, 28, 19, 15, 20, 33, 12, 17, 10. Let the hash function be $h(k) = k \mod 9$.

Solution.



•

(c) (11.4-1) Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h?(k) = k$. Illustrate the result of inserting theses keys using linear probing, using the quadratic probing hash function $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ where $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m - 1))$.

Solution.

	linear probing	quadratic probing	double hashing
0	22	22	22
1	88		
2		88	59
3		17	17
4	4	4	4
5	15		15
6	28	28	28
7	17	59	88
8	59	15	
9	31	31	31
10	10	10	10

•

(d) (based on 11.4-2) Consider a hash table in which collisions are resolved by open addressing, and in which the probing function is $h(k, i)$. Write pseudocode for insertion and deletion from the hash table. The deletion method should set the deleted table entry to the special value DELETED, and the insertion method should handle that special value appropriately.

Solution.

```
def insert(T, k, v):
    for i from 0 to m:
        if T[h(k,i)] == null or T[h(k,i)] == DELETED:
            T[h(k,i)] = (k,v)
            return
    error("insert failed due to table full")

def delete(T, k):
    for i from 0 to m:
        if T[h(k,i)] == null:
            error("delete failed due to k not found")
        if T[h(k,i)][0] == k:
            T[h(k,i)] = DELETED
            return
    error("delete failed due to k not found")
```

•

1. (11.2-3) Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

Solution. If the linked lists maintained by hash chaining are not kept sorted, insertion can be done in constant time by appending to the beginning of a list (if using singly linked

lists) or at the end of a list (if using doubly linked lists with pointers to the head and tail of the lists). Searches are done by scanning a list, which has on expectation a length of n/m. Successful searches are on expectation $n/(2m)$ (which is the average of all possible successful search times: 1, 2, ..., n/m), whereas unsuccessful searches always require scanning the entire list, which is on expectation n/m. Once a key is found, deletion is constant time.

On the other hand, if we need to maintain linked lists that are sorted by key, insertion will take more time, as the linked lists need to be traversed until the correct position is found for the key we are trying to insert. Assuming that the keys are uniformly distributed, it is equally likely that the key will be inserted in any position in the linked list. So by the same argument as for successful searches in an unsorted list, an insertion into a sorted linked list takes on expectation $n/(2m)$ steps. Searches and deletions take the same amount of time whether the lists are sorted or not, because the same algorithm is used. In particular, we cannot speed up searches, say using a binary search because we are using a linked list rather than an array.

Overall, maintaining sorted linked lists slows insertions from worst case constant time to $n/(2m)$ time on expectation, while offering no improvement in searches and deletions. •

2. (11.4-3) Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.

Solution. By Theorem 11.6, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, where α is the load factor. That's $1/(1-3/4) = 4$ probes if the load factor is 3/4 and $1/(1-7/8) = 8$ probes if the load factor is 7/8.

By Theorem 11.8, the expected number of probes in a successful search is at most $(1/\alpha)\log(1/(1-\alpha))$. That's $1/(3/4)\ln 4 \simeq 1.38$ probes if the load factor is 3/4 and $1/(7/8)\ln 8 \simeq 2.08$ probes if the load factor is 7/8. •

3. Given a sorted array of distinct integers $A[1,\ldots,n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in $O(\log n)$.

Solution. Since the integers are distinct and sorted, for every i and j such that $i < j$, we have $j - i \leq A[j] - A[i]$, or equivalently

$$A[i] - i \leq A[j] - j.$$

This means that the sequence $A[0] - 0, A[1] - 1, \ldots, A[i] - i, \ldots A[n] - n$ is increasing. Therefore, to determine if there exists an index i such that $A[i] = i$ is equivalent to determining if this sequence of differences contains 0. Since this sequence is sorted, this can be done in $O(\log n)$ with a binary search for 0. •

4.
```
// clear all the cells in the same region as c.
private void sweep(Integer c) {
   Queue<Integer> queue = new LinkedList<>();
   Set<Integer> discovered = new HashSet<>();
   clearedCells[c] = true; // set to true to clear cell c

   if (mineField[c] == 0) {
      queue.add(c);
      discovered.add(c);
   }
   while (!queue.isEmpty()) {
      c = queue.remove();
      for (int n : adjacentCells(c)) {
         clearedCells[n] = true;
         if (mineField[n] == 0 && !discovered.contains(n)) {
            queue.add(n);
            discovered.add(n);
         }
      }
   }
}
```