Notes for TA: We'd recommend going through the problems in order. There are many more problems than time, so don't worry about going through all of them.

Problem 1-1. (Implementing a dictionary)

Notes for TA:

This is a question aimed at reviewing some of the key concepts from the lecture on hash tables. Some of the key terms that will hopefully be defined over the course of discussing this problem are:

- the **dictionary data structure**
- **direct addressing**
- **hash table**
- **simple uniform hashing**
- **collision resolution** (specifically **chaining**)
- **load factor**

Drawing a chart may be helpful for comparing the different approaches.

(a) Describe how a direct-address table can be used to implement a dictionary. What are some advantages and disadvantages to this approach?

Solution:

A direct-address table is an array (call it T) that has a slot for each possible key in the universe of keys (U). Searching for an element with key k simply returns the element indexed by k (i.e. $T[k]$). Inserting an object x is simply setting $T[x.key]$ to store x. And then deleting an object x is simply setting the slot, $T[x.key]$, to nil.

Advantages:

- constant-time in the worst case for all operations.

Disadvantages:

- impractical if $|U|$ is very large
- waste of space if number of keys in set is very small relative to $|U|$.

(b) Describe how a hash table (with chaining) can be used to implement a dictionary. What are its advantages and disadvantages?

Solution:

Components of a hash table with chaining:

- hash function mapping from U to one of the m buckets.
- m buckets that each point to a linked list of objects or *nil* if nothing has been hashed to that bucket.

Insert : insert at the head of the list hashed to.
Search : go to list where key of object hashes to, and linearly search linked list for object.
Delete : same as search, but if object found, remove object from linked list

Advantages:

- constant-time insertion in worst-case
- constant-time deletion and search in the average-case (assuming a reasonable hash function and load factor)
- need space proportional to actual number of elements being stored in dictionary and not the set of all possible elements (which is the case for direct-addressing).

Disadvantages:

- Worst-case linear time for deletion and search
- Extra-storage space needed for the linked lists

(c) Describe what open addressing is? What are its advantages and disadvantages?

Solution:

Open addressing can be thought of as an alternative form of collision resolution to chaining. All objects are stored in the array itself. Use a probe sequence to determine where to insert/search for a given key.

Advantages:

- No extra storage needed for chained lists

Disadvantages:

- Error if table overflows i.e. load factor exceeds 1
- Deletion is hard to program
- Performance is generally worse than chaining.

Problem 1-2. (Hashing and Collisions)

(a) Assume we have a hash function h that hashes keys to m buckets and meets the criteria of simple uniform hashing. What is the probability that two hashed keys collide?

Solution:

$\frac{1}{m}$

(b) When using h to hash n keys, how many collisions do we expect to have? Formally, the set of collisions is $\{(x,y) : x \neq y \text{ and } h(x) = h(y)\}$. So alternatively, what is the expected size of this set?

Solution:

Let Y be a random variable for the total number of collisions. Let X_{xy} be the indicator variable $I\{h(x) = h(y)\}$. Y and the indicator variables are related as follows:

$$Y = \sum_{x \neq y} X_{xy}$$

Furthermore, we know that from part (a):

$$P(X_{xy} = 1) = 1/m$$

This allows us to compute the desired expectation:

$$\begin{aligned} E[Y] =& E[\sum_{x \neq y} X_{xy}] \\ =& \sum_{x \neq y} E[X_{xy}] \\ =& \sum_{x \neq y} \frac{1}{m} \\ =& \frac{1}{m} \sum_{x \neq y} 1 \\ =& \frac{1}{m} \binom{n}{2} \end{aligned}$$

(c) How does the quantity above, the expected number of collisions, change as the load factor increases? Decreases?

Solution:

Increasing the load factor has the same effect as increasing n but holding m constant. From this, it is easy to see that as the load factor grows, so does the expected number of collisions (matching our intuition).

Decreasing the load factor has the same effect as increasing m (the number of buckets), while holding n constant. Thus, the expected number of collisions decreases as the load factor decreases (matching intuition again).

(d) What happens to the expected number of collisions if we keep the load factor the same but increase n (i.e. scale n and m at the same rate)?

Notes for TA: This becomes much easier to see if we write the expression from part (a) in terms of the load factor α:

$$\frac{1}{m}\binom{n}{2} = \frac{n(n-1)}{2m} = \frac{\alpha(n-1)}{2}$$

Solution:

The expected number of collisions increases.

Problem 1-3. (Getting familiar with the binary-search-tree property)

(a) Suppose we have numbers between 1 and 1000 in a binary search tree and we want to search for the number 363. Which of the following sequences could **not** be the sequence of the nodes examined?

(i)	$2, 252, 401, 398, 330, 344, 397, 363$
(ii)	$924, 220, 911, 244, 898, 258, 362, 363$
(iii)	$925, 202, 911, 240, 912, 245, 363$
(iv)	$2, 399, 387, 219, 266, 382, 381, 278, 363$
(v)	$935, 278, 347, 621, 299, 392, 358, 363$

Solution: (iii) and (v)

(b) Can you generalize your approach from above into an algorithm that outputs whether or not a sequence of keys is valid when searching for some key x? Write the pseudocode for this algorithm. (You can assume a successful search and thus the last key in the sequence should be the key you are searching for).

Solution:

```
IS_VALID_SEQUENCE(x, seq):
upper_bound = 10001
lower_bound = 0
for key in seq[:-1]
if key >= upper_bound or key <= lower_bound:
return FALSE
else:
if x < key:
upper_bound = key
elif x > key:
lower_bound = key
else:
```

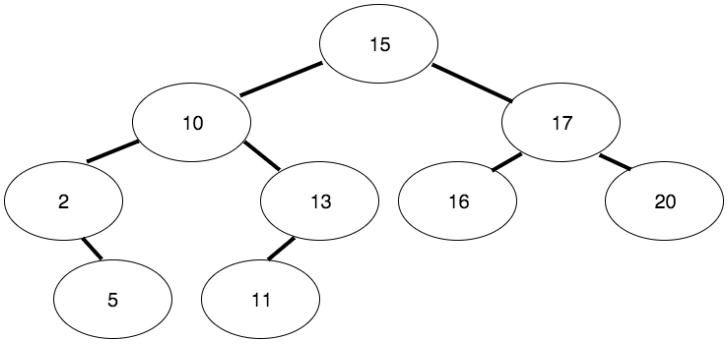
```
    return FALSE
    return x == seq[-1]
```

> **Notes for TA:** If you are feeling daring, you could do live coding of this algorithm in Python in order to demonstrate to students that are unfamiliar with Python that it is quick and easy language to learn

Problem 1-4. (Building a binary search tree)

(a) Construct a binary search tree by inserting the following keys into an empty tree T in the order given: $15, 10, 13, 17, 2, 5, 20, 16, 11$.

Solution:



(b) Are there alternative permuatations of these keys that produce this same tree?

Solution: Yes, we could permute the last two keys in the given order and end up with the same tree.

(c) Write out the order in which the keys would be printed when performing a *postorder tree walk, preorder tree walk, and inorder tree walk.*

Solution:
postorder: 5, 2, 11, 13, 10, 16, 20, 17, 15
preorder: 15, 10, 2, 5, 13, 11, 17, 16, 20
inorder: 2, 5, 10, 11, 13, 15, 16, 17, 20

(d) What would T look like after performing $Insert(T, 12); Delete(T, 10)$? What about if we had performed $Delete(T, 10); Insert(T, 12)$? In general, does the order in which we insert a key and delete a key not matter on the resulting tree?

> **Notes for TA:** It may be helpful to step through how a node gets deleted according to the pseudocode in lecture/the textbook.

Solution:

If we performed the alternate order, the resulting tree would be the same. However this is not always the case. As an example where the orderings do produce different trees, consider performing $Delete(T,17);Insert(T,18)$ vs $Insert(T,18);Delete(T,17)$?