Instructions.

1. Solution may **not** be submitted by students in pairs.

2. You may submit a pdf of the homework, either printed or hand-written and scanned, as long as it is **easily** readable.

3. If your solution is illegible and not clearly written, it might not be graded.

4. Unless otherwise stated, you should prove the correctness of your answer. A correct answer without justification may be worth less.

5. If you have discussed any problems with other students, mention their names clearly on the homework. These discussions are not forbidden and are actually **encouraged**. However, you must write your whole solution yourself.

6. In this homework, please solve the questions **yourself** (without consulting other students). In the previous homeworks, I was trying to encourage brainstorming and to promote fair team work. I hope that you have found it useful. But of course then, under this model of collaboration, your homework grade reflects what you know, but is also influenced by what your teammates know.

7. Unless otherwise specified, all questions have the same weight.

8. You may refer to data structures or their properties studied in class without having to repeat details, and may reference theorems we have studied without proof. If your answer requires only modifications to one of the algorithms, it is enough to mention the required modifications, and the effect (if any) on the running time and on other operations that the algorithm performs.

9. In general, a complete solution should contain the following parts:
   
   (a) A high level description of the data structures (if needed). *E.g. We use a binary balanced search tree. Each node contains, a key and pointers to its children. We augment the tree so each node also contains a field...*
   
   (b) A clear description of the main ideas of the algorithm. You may include pseudocode in your solution, but this may not be necessary if your description is clear.
   
   (c) Proof of correctness (e.g. show that your algorithm always terminates with the desired output).
   
   (d) A claim about the running time, and a proof showing this claim.
Questions.

1. Consider a SkipList containing a set $S$ of $n$ keys. We say that the rank of a key $x \in S$ is $k$ if exactly $k - 1$ other keys in $S$ are smaller than $k$. So the rank of the smallest key is 1 and the rank of the largest is $n$. The median has rank $\lceil n/2 \rceil$.

You need to modify We modify the basic structure of the SkipList, so that each one of the following 3 operations could be performed in expected time $O(\log n)$.

(a) Insert($x$) – Insert the new key $x$ into the SkipList.
(b) Delete($x$), – Delete the key $x$.
(c) Find($x$) – Report whether $x$ is in the SkipList.
(d) Query($k$). Given a query integer value $k$, find a key whose rank is $k$.

Explain how each of these operations could be performed in an augmented SkipList. An Augmented SkipList is a structure that is very similar to the SkipList discussed in class, but in addition, each cell $v$ also stores a field $v\rightarrow size$. This field maintains the number of keys of $S$ which $\geq v\rightarrow key$ and also $< v\rightarrow next\rightarrow key$.

![Figure 1: An example of a SkipList with augmented fields. With each cell $v$ only the value of $v\rightarrow size$ is shown. The keys are shown only in some of them. The keys of the other cells are not shown.](image)

2. We need to maintain a data structure for a set $S$ of $n$ items. Each item is given with a unique id. We need to enable the following operations, each in expected time $O(\log n)$. In all the operations below, $x, x_1$, and $x_2$ are all ids of items.

(a) Insert ($x, S$) The item whose id is $x$ has arrived, and is inserted into the data structure.
(b) Delete ($x, S$) The item whose id is $x$ is removed.
(c) Count ($x_1, x_2$) Report how many items in the data structure have ids between $x_1$ and $x_2$, (including $x_1$ and $x_2$).

**Answer:** Use the previous question. Count ($x_1, x_2$) = rank($x_2$) - rank($x_1$).

3. Suggest a data structure for a set $S$ of English words. The structure supports each one of the operations.

(a) Insert($w, S$) – insert the new word $w$ into $S$.
(b) Delete($w, S$).
(c) Find($w, S$) – reports whether $w$ is in $S$. Returns 'YES' if the word $w$ is in $S$.

Assume that $w$ is any English word with up to 50 characters.

Each operation is expected to take $O(1)$ time.

We do not distinguish between small and capital letters. So if we perform insert("Hello"), and then perform find("hello"), the answer should be 'YES'.
Answer: We will convert every word to its canonical form. For example, we could decide that canonical form here means that the word is using small letters. This means that both during ‘insert’, or during ‘find’, we will use this form.

Next we need to decide what is the hash function. Obviously we use dot product. We need to guesstimate $m$ (table size). Then we pick constants $a_0 \ldots a_r$, all integers between 0 and $m-1$. Then every word $w$ is expressed in binary, chopped into keys $k_0, \ldots k_r$, each of length $\log_2 m$ bits. For example, of $m = 2^{16}$ then each $k_i$ contains 16 bits - 2 ascii characters. Finally we compute $h(w) = \sum (a_i \cdot k_i) \mod m$.

4. The question refers to a $k \times k$ tic-tac-toe game. It is similar to the $3 \times 3$ game, but the grid is of $k$ rows and $k$ columns (rather than $3 \times 3$). You are given a huge database $S$ of states from the game. Each state is specified by the positions of the 'O' and the 'X' on the grid.

In practical applications, each such configuration is stored with, say, the chances to reach a winning configuration from this configuration. This is not needed for this question.

![Figure 2: Examples of a $3 \times 3$ tic-tac-toe game (Credit:Wiki)](image)

(a) Suggest a data structure that given a new state, this data structure enables you to efficiently find whether this state appears in $S$. Your method should be as fast as possible. Express all parameters of the method you use.

The rules of the games are not important - you just need to decide whether this state exactly have seen before.¹

(b) Next we define two state to be identical if one is obtained from the other by rotation clockwise or ccw by 90°, 180°, and/or mirror reflection. (hint - use Question 3).

Suggest a method that enables you not to store any two identical states, and yet for a new state, find if an identical state is stored already. You could not perform 8 queries per state.

(c) Extend your method so you could store configuration of a game where different pieces have different rules. (e.g. Chess).

5. You are given a preference list of $n$ males and $n$ females. Is it possible that a person $b$ is the pessimal person for two people ?

Use the definition of “Pessimal” as defined in the slides. (Note that this might be different than an intuitive guess of the meaning.)

6. Suggest an algorithm that checks if your 10TB disk hard drive contains two identical files. You are not allowed to use values provided by the File System/Operating System (such as MD5). The number $n$ of files is about $10^{10}$. Note that 1GB is roughly $10^9$.

If there are any identical pairs of files, your algorithm should print the names of such a pair, and stop.

¹ From Wiki: Tic-tac-toe (also known as noughts and crosses or Xs and Os) is a paper and pencil game for two players, X and O, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.
Suggest a solution that is efficient both in terms of CPU time and in terms of the number of disk access operations (I/O).

Your algorithm should be practical for your current desktop or PC.

**Answer:** A naive solution is to use a hash table $T[1..m]$, a hash function $h(FILE) \rightarrow [0..m-1]$. For every file $f$ in the system, we compute $h(f)$, store a point to the file at $T[h(f)]$, and if any collision occurs with another file, that is, $h(f) = h(f')$, we compare $f$ and $f'$ explicitly.

If the number of files is not too large, this is a decent method. However, it is limited by the size of $m$ that fits to memory. For a huge number of files, one might prefer to compute first a hash function $h'(FILE) \rightarrow [0..M-1]$, where $M >> 2^{64}$ is a large constant. We refer to $h(FILE)$ as the signature of $FILE$. Another hash function $h(SIGNATURE) \rightarrow [0..m-1]$ is computed, and a table $[0..m-1]$ is used as before. If collision occurs, we first compare the files signatures, and only if they match, we compare the files themselves. Hence the number of false matches is miniscule.

Finally, the dot product method could be used for computing $h'(file)$ efficiently.

7. Repeat the previous question, but now use MD5.

Functions as MD5 have been implemented very efficiently in your OS, and are likely to run faster hash functions you could implement.

8. This question tries to clarify why every hash function could be beaten by bad data.

   (a) Let $U$ be a set of 100 keys, and $h$ be a hash function that maps each key to an integer between 0 and 9. Show that you can find a subset $K \subset U$ of 10 keys in $U$ such that $h(x) = h(y)$ for every pair $x, y \in K$. That is, $h$ maps all keys of $K$ to the same index.

   (b) Generalize the previous question, but now $U$ is a set of $m^2$ keys and $h$ maps each key to an integer between 0 and $m-1$.

   **Answer:** Assume we use a chain hashing with a table $T[0..m-1]$. Let $n_i$ be the number of keys from $U$ mapped to slot $i$ (for every $0 \leq i < m$.) Since $m^2 = \sum_{i=0}^{m} n_i$, at least one $n_i$ must be $\geq m$. Let's assume WLOG that this is $n_0$. This means that $U$ contains $\geq m$ keys that $h$ maps to $T[0]$. We set $K$ to be the set of these keys.

9. Which of the following two open addressing hash functions is likely to give better performances (for the same set of keys, and same table size $m$)? Explain your choice.

   (a) $h_1(k, i) = (h(k) + i) \mod m$

   (b) $h_2(k, i) = (h(k) + i^2) \mod m$