# CS 161 (Stanford, Winter 2024) Homework 5

**Style guide and expectations:** Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
**What we expect:** Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

**Exercises.** The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

## 1 Exercise: Universality

Plucky the penguin is hosting a free food event for up to 100 Stanford students! To keep track of RSVP's, Plucky wants to build a hash table with 100 buckets using the attendees' 8-digit student ID as keys.

Plucky still needs to choose a hash family $\mathcal{H} = \{h_m : m \in \{1, \ldots, 1000\}\}$, and being a pedantic penguin, wants it to be universal.

Does each of the following formulations result in a universal hash family?

**[We are expecting:** For each candidate formulation, a proof of universality or a counterexample.**]**

### 1.1 (2 pt.)

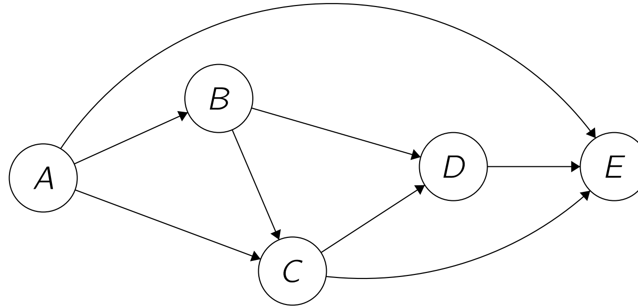
Let $h_m(x)$ be the sum of each digit in input $x$, plus $m$, truncated to the final two digits. For example, $h_{100}(01234567) = 28$ because $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 100 = 128$.

### 1.2 (2 pt.)

Let $h_m(x)$ be the final two digits of $mx$. For example, $h_4(01234567) =$ last two digits of $4938268 = 68$.

## 2 Exercise: BFS and DFS Basics

Consider the following directed acyclic graph (DAG):




## 2.1 (2 pt.)

Run DFS starting at vertex $C$, breaking any ties by alphabetical order. (For example, if DFS has a choice between $B$ or $C$, it will always choose $B$. This includes when DFS is starting a new tree in the DFS forest.) Recall that when you run DFS, if no vertices are reachable from the last node you started from, then it will resume the search at an unvisited vertex.

(a) What do you get when you order the vertices by **ascending** start time?

(b) What do you get when you order the vertices by **descending** finish time?

**[We are expecting:** An ordering of vertices. No justification is required.**]**

## 2.2 (2 pt.)

Run DFS starting at vertex $D$, *this time treating all edges as undirected*. Once again, break any ties by alphabetical order.

(a) What do you get when you order the vertices by **ascending** start time?

(b) What do you get when you order the vertices by **descending** finish time?

**[We are expecting:** A pair of orderings of vertices. No justification is required.**]**

## 2.3 (1 pt.)

Run BFS *(not DFS)* starting at vertex $D$, *treating all edges as undirected*. Break any ties by alphabetical order. What is the order that the nodes are marked by BFS?

**[We are expecting:** An ordering of vertices. No justification is required.**]**

**Problems.** The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

# 3 Perfect hashing

Ollie the overachieving ostrich has just read about hash tables and wants to learn more!

Recall from lecture 8 that a hash table supports the following operations:

- INSERT($k$): Insert key $k$ into the hash table.
- SEARCH($k$): Check if key $k$ is present in the table.
- DELETE($k$): Delete the key $k$ from the table.

For simplicity, Ollie is examining *static hash tables*, a more restricted problem where we know all the keys to be inserted ahead of time. Specifically, a static hash table supports the following operations:

- BUILD($k_1, \ldots, k_n$): Construct a static hash table from a set of $n$ **unique** keys.
- SEARCH($k$): Check if key $k$ is present in the table.

To distinguish static hash tables from the more general hash tables presented in lecture, we will refer to the latter as *dynamic hash tables* for the remainder of this problem.

**Notes from the pedantic penguin:** For this problem you can assume:

- For any integer $b > 0$, you have access to a universal hash family $\mathcal{H}_b$ (each hash function $h$ in $\mathcal{H}_b$ maps a key to an integer in $\{1, 2, \ldots, b\}$) which satisfies

$$\forall k \neq k', \Pr_{h \overset{u.a.r.}{\sim} \mathcal{H}_b}[h(k) = h(k')] = \frac{1}{b}.$$

- INSERT($k$) runs in deterministic $O(1)$ time for dynamic hash tables, as long as the key being inserted is guaranteed to be unique. (If the key is unique, we can just append it to a bucket without having to scan through the bucket.)
- Initializing an empty dynamic table with $n$ buckets takes deterministic $O(n)$ time.

## 3.1 Simple static tables (0 pt.)

Ollie first looks at this simple implementation of static hash tables using dynamic hash tables:

- BUILD($k_1, \ldots, k_n$): Construct a dynamic hash table with $n$ buckets and a hash function $h$ chosen uniformly at random from $\mathcal{H}_n$. Then, run INSERT($k_i$) for each $k_i$ (INSERT($k_i$) simply appends the key $k_i$ to the $h(k_i)$-th bucket).
- SEARCH($k$): Run SEARCH($k$) on the dynamic hash table.

What are the asymptotic runtimes of BUILD and SEARCH for this implementation? What is the asymptotic size of this table?

Solution (provided)

BUILD runs in deterministic $O(n)$ time, since it consists of $n$ unique calls to INSERT, which is a deterministic $O(1)$ operation.
SEARCH($k$) runs in expected $O(1)$ time because expected number of keys in $h(k)$-th bucket is $O(1)$ by our choice of the hash function $h$ (which you have learned in lecture).
Since this hash table uses $O(n)$ buckets, and there are $n$ keys in the table, this table has $O(n)$ size.

## 3.2 Expected vs. deterministic (1 pt.)

Why does an expected $O(1)$ runtime for SEARCH not imply a deterministic worst-case $O(1)$ runtime?

**[We are expecting:** A brief explanation in plain English.**]**

## 3.3 Expected collisions (1 pt.)

Ollie is despondent upon learning of the lack of deterministic search. Being an overachieving ostrich, Ollie searches for a new implementation with better performance.

Ollie now looks at hashing $n$ unique keys $k_1, \ldots, k_n$ into a table with $n^2$ buckets using a hash function chosen uniformly at random from $\mathcal{H}_{n^2}$. What is the expected total number of collisions in such a table? (The **total number of collisions** in a hash table is the total number of pairs $(k_i, k_j)$ with $i < j$ such that $k_i$ and $k_j$ end up in the same bucket.)

**Hint:** You may cite equations from lecture notes.

**[We are expecting:** A mathematical derivation.**]**

## 3.4 Collision probability (1 pt.)

When hashing $n$ unique keys into a table with $n^2$ buckets using a hash function chosen uniformly at random from $\mathcal{H}_{n^2}$, show that there is at least a $1/2$ probability of having no

collisions in the table. (In other words, show that there is at most a 1/2 probability of having any collisions in the table.)

**Hint:** Markov's inequality[1] may be useful. For a random variable $X$ and a constant $a$:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

**[We are expecting:** A mathematical derivation.**]**

## 3.5 Big fast tables (4 pt.)

Using what you have proved in 3.4, help Ollie implement a static hash table with the following properties:

- BUILD($k_1, \ldots, k_n$) runs in *expected* $O(n^2)$ time.
- SEARCH($k$) runs in *deterministic* $O(1)$ time.
- The size of the hash table is $O(n^2)$.

**[We are expecting:** A specification of BUILD and SEARCH in clear English or pseudocode, along with justifications of the required properties.**]**

## 3.6 Small slow tables (4 pt.)

Although we have satisfied Ollie's runtime requirements, Ollie is still worried about the $O(n^2)$ size.

We now turn to more space-efficient tables with deterministic search times. Help Ollie implement a static hash table with the following properties:

- BUILD($k_1, \ldots, k_n$) runs in *expected* $O(n)$ time.
- SEARCH($k$) runs in *deterministic* $O(n)$ time.
- The size of the hash table is $O(n)$.
- The total number of collisions in the hash table is $O(n)$.

**[We are expecting:** A specification of BUILD and SEARCH in clear English or pseudocode, along with justifications of the required properties.**]**

[1] For Markov's inequality to hold, $X$ and $a$ must both be non-negative. This constraint is not relevant here since we can't have negative collisions, but it is important to note in the general case.

### 3.7 Interlude (2 pt.)

For a table with $n$ unique keys, $m$ buckets, and $k$ total collisions, let $s_j$ be the size of bucket $j$, where $j \in [1, \ldots, m]$. Show the following relation:

$$\sum_{j=1}^{m} s_j^2 = 2k + n$$

**[We are expecting:** A mathematical derivation.**]**

### 3.8 Small fast tables (6 pt.)

Now, we are ready to define a data structure that fulfills Ollie's runtime requirements *and* space requirements.

Help Ollie implement a data structure with the following properties:

- BUILD($k_1, \ldots, k_n$) runs in *expected* $O(n)$ time.
- SEARCH($k$) runs in *deterministic* $O(1)$ time.
- The size of the data structure is $O(n)$ buckets.

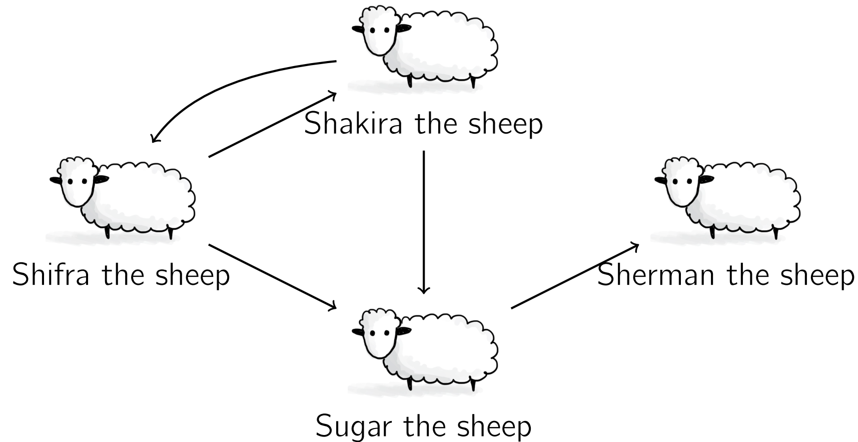
**Hint:** Can we combine static hash tables with different tradeoffs (which you have constructed in 3.5 and 3.6) to get the best of both worlds?

**[We are expecting:** A specification of BUILD and SEARCH in clear English or pseudocode, along with justifications of the required properties. Feel free to reference previous portions of the problem.**]**

## 4 Wake up, Sheeple!

You arrive on an island with $n$ sheep. The sheep have developed a pretty sophisticated society, and have a social media platform called Baaaahtter (it's like Twitter but for sheep[2]). Some sheep follow other sheep on this platform. Being sheep, they believe and repeat anything that they hear. That is, they will re-post anything that any sheep they are following said. We can represent this by a graph, where $(a) \rightarrow (b)$ means that $(b)$ will re-post anything that $(a)$ posted. For example, if the social dynamics on the island were:

[2] Also my new start-up idea




then Sherman the Sheep follows Sugar the Sheep, and Sugar follows both Shakira and Shifra, and so on. This means that Sherman will re-post anything that Sugar posts, Sugar will re-post anything by Shifra and Shikira, and so on. (If there is a cycle then each sheep will only re-post a post once).

For the parts below, let $G$ denote this directed, unweighted graph on the $n$ sheep. Let $m$ denote the number of edges in $G$.

## 4.1 The influencer circle (5 pt.)

Call a sheep an **influencer** if anything that they post eventually gets re-posted by every other sheep on the island. In the example above, both Shifra and Shakira are influencers.

Prove that, if there is at least one influencer, then all influencers are in the same strongly connected component of $G$, and every sheep in that component is an influencer.

**[We are expecting:** A short but rigorous proof.**]**

## 4.2 Who is the influencer? (5 pt.)

Suppose that there is at least one influencer. Give an algorithm that runs in time $O(n + m)$ and finds an influencer. You may use any algorithm we have seen in class as a subroutine.

**[We are expecting:** Pseudocode or a very clear English description of your algorithm, an informal justification that your algorithm is correct , an informal justification that the running time is $O(n + m)$.**]**

## 4.3 Is there an influencer? (2 pt.)

Suppose that you don't know whether or not there is an influencer. Give an algorithm that runs in time $O(n+m)$ and either returns an influencer or returns the text `"no influencer"`. You may use any algorithm we have seen from class as a subroutine, and you may also use your algorithm from previous part as a subroutine.

**[We are expecting:** Pseudocode or a very clear English description of your algorithm, an informal justification that your algorithm is correct, an informal justification that the running time is $O(n+m)$**]**

# 5 Finding shortest paths for special graphs

## 5.1 Graphs with few negative edges (4 pt.)

Let $G = (V, E)$ be a directed weighted graph, and $s \in V$ a vertex. Suppose there are no negative-weight cycles, and only the edges going out of $s$ may have negative weights.

There are algorithms that can solve the shortest path problem on any graph (without negative cycles) including the graph above, such as Bellman-Ford. However, they have worse asymptotic runtimes. Since the given graph is special and only a specific set of edges may be negative, modify the graph suitably to use a faster algorithm from lecture to solve the shortest paths from $s$ on this graph?

**[We are expecting:** Short English description stating the steps to modify the graph, algorithm name (from class) to solve shortest path problem on the modified graph, brief justification for correctness, and time complexity of the proposed algorithm.**]**

## 5.2 Graphs with small integer weights (4 pt.)

Let $G = (V, E)$ be a directed weighted graph, where all the edge weights are from $\{1, 2, 3\}$. Design an algorithm that modifies the graph suitably and solves the shortest path problem for the given graph asymptotically faster than the algorithm you used in the previous part. Mention the time complexity of both the algorithms.

**[We are expecting:** Short English description stating the steps to modify the graph, algorithm name (from class) to solve shortest path problem on the modified graph, and time complexity of the proposed algorithm along with a short justification. (No pseudocode necessary)**]**