CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 9: Dictionaries and Graphs

Harvard SEAS - Fall 2023 *Oct 3, 2023*

1 Announcements

- Midterm in class 10-12. Arrive by 9:45 am. Midterm grades will be given by percentage points, rather than by N/L/R grading.
- Midterm scope includes material on randomized algorithms (last time) and randomized data structures (today) — the main takeaways (e.g. Monte Carlo vs. Las Vegas, when to use each), not proofs.
- This week's sections will be review for midterm!

2 Randomized Data Structures

Recommended Reading:

- CLRS 11.0–11.4
- Roughgarden II 12.0–12.4

We can allow data structures to be randomized, by allowing the algorithms Preprocess, EvalQ, and EvalU to be randomized algorithms, and again the data structures can either be Las Vegas (never make an error, but have random runtimes) or Monte Carlo (have fixed runtime, but can err with small probability).

A canonical data structure problem where randomization is useful is the *dictionary* problem. These are data structures for storing sets of key-value pairs (like we've been studying) but where we are *not* interested in the ordering of the keys (so min/max/next-smaller/selection aren't relevant).

Updates :	Insert or delete a key-value pair (K, V) with $K \in [U]$ into the multiset
Queries :	Given a key K, return a matching key-value pair (K, V) from the multiset (if one exists)

Data-Structure Problem(Dynamic) Dictionaries

Of course the Dynamic Dictionary Problem is easier than the Predecessor Problem we have already studied, so we can use Balanced BSTs to perform all operations in time $O(\log n)$. So our goal here will be to do even better — to get time $O(1)$.

Theorem 2.1. *Dynamic Dictionaries has an implementation in which Preprocess, Insert, Delete, and Search are all O(1).*

We'll work our way up to a proof of this theorem by a series of ideas which get progressively closer to a correct implementation. We've actually already seen that we can get $O(1)$ time updates and queries as follows:

A deterministic data structure:

- Preprocess(U): Initialize an array A of size U.
- Insert(K, V): Place (K, V) into a linked list at slot $A[K]$.
- Delete(K): Remove the head of the linked list at slot $A[K]$.
- Search(K): Return the head of the linked list at $A[K]$.

Q: Where have you seen this data structure before?
A: Counting Sort!

A problem with this approach: Preprocess takes time (and space) $\Theta(U)$, and U can be very large. If we want to use this with, say, keys that are 64-bit words (common in practice), we would need an array of size 2^{64}, which is completely infeasible.

Idea 1: Smaller array. Use an array A of size $m \ll U$. Put a key K spot $A[K\%m]$. where % represents the mod function. For instance, we might choose $m = \Theta(n)$, where $n \ll U$.

Problem: Some sequences of operations will *always* fail, violating our correctness requirements, since different keys collide under the mod function. For instance, if $m = 128$, then after the operations Insert($(0, A)$) and Insert($(128, B)$), the operation Search(0) will return $(128, B)$, which fails. This problem can occur with real-life data, for example if $m = 1000$ and the keys represent annual salaries, many different salaries will map to $A[0]$

Note that the same problem comes up if, instead of putting K at $A[K\%m]$, we choose some other function $h : [U] \to [m]$ and put K at $A[h(K)]$. In particular, if $m < U$, there will be at least two keys that collide (by the Pigeonhole Principle), and if $m \ll U$, there will be many collisions. In the worst case, our dataset will consist entirely of colliding keys, leading to many incorrect search results.

Also note that if we look through the whole linked list at $A[0]$ instead of just the head, that fixes correctness at the cost of making queries take $\omega(1)$ time: for instance, if $m = 128$, then after the operations Insert($(0, A)$) and n copies of the operation Insert($(128, B)$), the operation Search(0) will have to look through a list of length n to find and return $(0, A)$, which takes $\Theta(n)$ time. We'll come back to that idea later.

Idea 2: Random function. Choose a *random* function $h : [U] \to [m]$, and put K at $A[h(K)]$. That is, in our preprocessing step, for each element $K \in [U]$, we set $h(K) = \texttt{random}(m)$, independently for each key K, and store h. Then we can bound the probability of an error even for worst-case data: Suppose we have inserted data with keys $K_0, K_1, \ldots, K_{n-1}$ and now perform the query Search(K). In order for us to return the wrong value, it must be that one of the keys K_i

hashes to the same value as $h(K)$ yet $K_i \neq K$. Thus:

$$\begin{aligned}\Pr[\text{Search}(K) \text{ returns incorrect value}] &\leq \Pr[\exists i\ h(K_i) = h(K), K_i \neq K_i \\ &\leq \sum_{i:K_i \neq K} \Pr[h(K) = h(K_i)] \\ &\leq n \cdot \frac{1}{m}\end{aligned}$$

where in the final line we used that h was a random mapping from $[U]$ to $[m]$. So if $m \geq 100n$, then we have an error probability of at most $1/100$, so this is a correct Monte Carlo data structure.

Problem: We again need $\theta(U)$ space, since we have to define and store this function by choosing a random output in $[m]$ for every single element in $[U]$, and preprocessing again takes $\theta(U)$ time—both as bad as the original deterministic data structure!

Idea 3: Low-complexity random-ish function, AKA "hash function". In the calculation above, the only thing we needed out of the fact that h was a random function was that $\Pr[h(K) = h(K_i)] = O(1/m)$. That's a much weaker requirement on h than being a fully random function, and it turns out that we can get enough randomness to satisfy that requirement by generating only $O(1)$ random numbers, rather than the U random values that define a random function.

Constructing such functions is outside the scope of CS 120[1], but we'll assume that we can choose a *random hash function* $h : [U] \to [m]$, which behaves like a completely random function, except that:

1. Generating a random hash function h takes time $O(1)$.
2. Storing h takes space $O(1)$.
3. For all $x \in U$, evaluating $h(x)$ takes time $O(1)$.

(The above bounds assume that elements of U fit in a single word. For large universes where elements of U take k words, then the above bounds increase to $O(k)$.) The primary difference between a random hash function and a random function is that we only need to quickly evaluate a (short) function $h(x)$ on every input x in the random hash function, while we needed to store the output to every possible input to define a random function.

Using the random hash function h inside the previous deterministic data structure, we get the following Monte Carlo data structure.

A Monte Carlo data structure:

- Preprocess(U, m): Initialize an array A of size m. In addition, choose a random hash function $h : [U] \to [m]$ from the universe $[U]$ to $[m]$.
- Insert(K, V): Place (K, V) into the linked list at slot $A[h(K)]$.

[1]There is some optional reading on it in Section 2.1 below in case you are curious.

- Delete(K): Remove an element from the linked list at slot $A[h(K)]$.

- Search(K): Return the head of the linked list at $A[h(K)]$.

Unfortunately, this is a Monte Carlo data structure.

A Las Vegas data structure ("Hash Table"):
The same data structure as above, except we modify Search(K) to walk through the linked list $A[h(K)]$ and return the first element of the list that has the correct key K (and if none do, return $\perp$).

Here, we bound the *expected runtime* via the same analysis as before, because if no elements collide (the event we bound above) the additional linked list checking will only be an $O(1)$ slowdown. Quantitatively, we can show an expected runtime of $O(1 + n/m)$. We call $\alpha = n/m$ the *load* of the table, so the runtime is $O(1 + \alpha)$. Notice that here we get $O(1)$ expected time even if $n > m$, provided that $m = \Omega(n)$.

To maintain both time and space efficiency, we need to tailor the size m of the table to the size n of the dataset, which we may not know advance. This can be solved by dynamically resizing the dataset as the hash table gets too full. For example, when we reach load $\alpha = 2/3$, we can double the table size to bring us back to $\alpha = 1/3$.

2.1 Hash functions

This section is optional reading on how hash functions are constructed, in case you are interested. More aspects of this problem are discussed in the CLRS and Roughgarden texsts, and courses like CS 124, CS 127, CS 222, CS 223, and CS 225.

We want a family $\mathcal{H}$ of hash functions h, smaller than the family of all random functions, that allows us to (a) store h compactly, (b) evaluate h efficiently, and (c) still prove that the worst-case expected time for operations on the hash table is $O(1 + \alpha)$. An example: pick a prime number $p > U$. Then for $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ we define the hash function

$$h_{a,b}(K) = ((aK + b) \mod p) \mod m.$$

This takes 2 words to store, can be evaluated in $O(1)$ time, and maintains the same pairwise collision property: for every $K \neq K' \in [U]$, we have

$$\Pr_{a,b}[h_{a,b}(K) = h_{a,b}(K')] \leq \frac{1}{m} \tag{1}$$

(For a proof, see CLRS. This requires a little bit of number theory and is beyond the scope of this course.) A hash family satisfying (1) is known as an *universal hash family*, and this property suffices to prove our expected runtime bounds of $O(1 + \alpha)$.

We could also use a "cryptographic" hash function like SHA-3, which involves no randomness but it conjectured to be "hard to distinguish" from a truly random function. (Formalizing this conjecture is covered in CS 127.) This has the advantages that the hash function is deterministic and that we do not need to fix a universe size U. On the other hand, the expected runtime bound is then based on an unproven conjecture about the hash function, and also these hash functions,

while quite fast, are not quite computable in $O(1)$ time. By combining them with a little bit of randomization, they can also be made somewhat resilient against adversarial data, where an adversary tries to learn something about the hash function by interacting with the data structure and uses that knowledge to construct data that makes the data structure slow.

3 Storing and Search Synthesis

We have seen several approaches to storing and searching in large datasets (of key-value pairs). For each of these approaches, describe a feature or combination of features it has that none of the other approaches provide.

1. Sort the dataset and store the sorted array
   Selection queries can be done in $O(1)$ (after preprocessing of $O(n \log n)$ to initially sort the array).

2. Store in a binary search tree (balanced and appropriately augmented)
   A BST is a *dynamic* data structure which allows insert, delete, predecessor, and selection all in $O(h) = O(\log n)$ time.

3. Run Randomized QuickSelect
   One selection query takes $O(n)$, which is quicker than the time to preprocess/insert the entire dataset of the above options.

4. Store in a hash table
   Search (+ updates) can be done in $O(1)$ time, as long as it's not very overloaded. Hash tables are good for unordered data.

4 Graph Algorithms

Recommended Reading:

- Roughgarden II Sec 7.0–7.3, 8.0–8.1.1
- CLRS Appendix B.4

Motivating Problem: Google Maps. Given a road network, a starting point, and a destination, what is the shortest way to get from the starting point s to the destination t?

Q: How to model a road network?

A: Graphs!

Definition 4.1. A *directed graph* $G = (V, E)$ consists of a finite set of *vertices* V (sometime called *nodes*), and a set E of ordered pairs (u, v) where $u, v \in V$ and $u \neq v$.

Example:

Q: What possibilities doesn't this model capture and how might we augment it?

- Sometimes we allow *multigraphs*, where there can be more than one edge from u to v, and possibly also loops (edges $\{u, u\}$). Our definition as above is that of a *simple* graph.
- We have defined an *unweighted* graph, but we may also want to assign weights/costs/lengths to each edge (e.g. modelling travel time on a road).
- An *undirected* graph has unordered edges $\{u, v\}$. Equivalently, we can think of this as a directed graph where if $(u, v) \in E$, we also have $(v, u) \in E$. (We could think of this as a road network with no one-way roads.)
- A graph is *planar* if it can be drawn in a 2D plane without edge crossings. Road networks are mostly but not entirely planar (e.g. consider overpasses).
- Some real-world graphs have some additional (e.g. hierarchical) structure that might be useful to exploit in algorithms (e.g. we may know that usually the best way to drive from one city to another is to use local roads to get to/from a highway).

Unless we state otherwise, assume *graph* means a **simple, unweighted, undirected** graph, and a *digraph* means a **simple, unweighted, directed** graph.

Remark: as we'll see, graphs are useful for modelling a vast range of different kinds of relationships, e.g. social networks, the world wide web, kidney donor compatibilities, scheduling conflicts, etc.

5 Shortest Walks

Motivated by a (simplified version) of the Google Maps problem, we wish to design an algorithm for the following computational problem:

Input	**:** A digraph $G = (V, E)$ and two vertices $s, t \in V$
Output	**:** A *shortest walk* from s to t in G, if any walk from s to t exists

Computational Problem ShortestWalk

Definition 5.1. Let $G = (V, E)$ be a directed graph, and $s, t \in V$.

- A *walk* w from s to t in G is a sequence $v_0, v_1, \ldots, v_\ell$ of vertices such that $v_0 = s$, $v_\ell = t$, and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, \ell$.
- The *length* of a walk w is length(w) = the number of edges in w (the number ℓ above).

- The *distance* from s to t in G is

$$\text{dist}_G(s,t) = \begin{cases} \min\{\text{length}(w) : w \text{ is a walk from } s \text{ to } t\} & \text{if a walk exists} \\ \infty & \text{otherwise} \end{cases}$$

- A *shortest walk* from s to t in G is a walk w from s to t with $\text{length}(w) = \text{dist}_G(s,t)$

Q: An algorithm immediate from the definition?
A: Enumerate over all walks from s in order of length, and terminate after finding the first that ends at t.

But when can we stop this algorithm to conclude that there is no walk? The following lemma allows us to stop at walks of length $n-1$.

Lemma 5.2. *If w is a shortest walk from s to t, then all of the vertices that occur on w are distinct. That is, every shortest walk is a* path *— a walk in which all vertices are distinct.*

Proof.
Suppose for contradiction that there is a shortest walk $w = (s = v_0, v_1, \ldots, v_\ell = t)$ that does *not* satisfy this property, i.e. $v_i = v_j$ for some $i < j$. But then we can cut out the loop $(v_i, v_{i+1}, \ldots, v_j)$ and produce the walk $w' = (s = v_0, \ldots, v_{i-1}, v_i = v_j, v_{j+1}, \ldots, v_\ell)$. We have the length of w' is strictly less than that of w and has the same start and endpoints. But then w is not a shortest walk, so we have a contradiction. □

Q: With this lemma, what is the runtime of exhaustive search?
A: $(n-1)! \cdot O(n)$

Q: How can we get a faster algorithm?
A: Breadth-first search (BFS)—which we'll talk about next lecture!