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:

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.

Problem:

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 $A[h(K)]$.

Idea 2: Random function. Choose a *random* function $h : [U] \to [m]$, and put K at $A[h(K)]$.

Problem:

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,

[1]There is some optional reading on it in the detailed lecture notesin case you are curious.

except that:

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:

A Las Vegas data structure ("Hash Table"):

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

2. Store in a binary search tree (balanced and appropriately augmented)

3. Run Randomized QuickSelect

4. Store in a hash table

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?

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
- The *length* of a walk w is
- The *distance* from s to t in G is
- A *shortest walk* from s to t in G is

Q: An algorithm immediate from the definition?
A:

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.

□

Q: With this lemma, what is the runtime of exhaustive search?
A:

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