CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 9: Dictionaries

Harvard SEAS - Fall 2024 *Oct 1, 2024*

1 Announcements

- Add/drop 10-07
- The Computer Science Sophomore Concentration Declaration Advising Event is 11am-12noon, Tues, Oct 1, in SEC LL2.224.
- SRE next Tuesday
- Pset 3 feedback survey.
- Mark your pages when uploading on gradescope. 0.1/3 points off Pset 3 onwards if you forget.

2 Loose ends from Lecture 8

Expected runtime of QuickSelect: Given an array of size n, the size of the subarray that we recurse on is bounded by $\max\{n_{\text{smaller}}, n_{\text{larger}}\}$. On average over the choice of our pivot, the subarrays will be fairly balanced in size, and indeed it can be shown that

$$\mathrm{E}\left[\max\{n_{\text{smaller}}, n_{\text{larger}}\}\right] \leq \frac{3n}{4}.$$

Thus, intuitively, the expected runtime of QuickSelect should satisfy the recurrence:

$$T(n) \leq T(3n/4) + cn,$$

for $n > 1$. Then by unrolling we get:

$$T(n) \leq cn + c \cdot \left(\frac{3}{4}\right) \cdot n + c \cdot \left(\frac{3}{4}\right)^2 \cdot n + \cdots = 4cn.$$

The power of randomization: A fundamental question is whether allowing randomization (in either the Las Vegas or Monte Carlo ways) actually adds power. Here are some examples of problems that reflect a potential gap between randomized and deterministic algorithms:

- Selection: QuickSelect is a simple $O(n)$ time Las-Vegas algorithm. There *is* a deterministic algorithm with runtime $O(n)$, which uses a more complicated strategy to choose a pivot (and has a larger constant in practice).
- Primality Testing: Given an integer of size n (where n=number of words), check if it is prime. There is an $O(n^3)$ time Monte Carlo algorithm, $O(n^4)$ Las Vegas algo and $O(n^6)$ determinstic algorithm (proven in the paper "Primes is in P").

- Identity Testing: Given some algebraic expression, check if it is equal to zero. This has an $O(n^2)$ time Monte Carlo algorithm, and the best known deterministic algorithm runs in $2^{O(n)}$!

Nevertheless, the prevailing conjecture (based on the theory of pseudorandom number generators) is: **randomness is not necessary for polynomial-time computation.** More precisely, we believe a $T(n)$ time Monte-Carlo algorithm *implies* a deterministic algorithm in time $O(n \cdot T(n))$, so we can convert any randomized algorithm into a deterministic one. You can learn more about this conjecture in courses like CS1210, CS2210, and CS2250.

3 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 formal definition can be obtained by referring back to the definition of a dynamic data structure in Lecture 5 and adding calls to `random` - we skip the details for brevity.

3.1 Dynamic Dictionaries

A canonical data structure problem where randomization is useful is the *dictionaries* problem:

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

Data-Structure Problem(Dynamic) Dictionaries

Dynamic Dictionaries Problem can be solved with operations (Insert, Delete, Search) in time $O(\log n)$ using Balanced BSTs. However, Balanced BSTs seem to be overkill, since min/max/next-smaller/selection queries are not a part of the Dynamic Dictionaries Problem. Indeed, we can construct a simple data structure with improved runtime of insert, delete and search, as long as U is not too large.

Claim 3.1. *Dynamic Dictionaries has a solution in which Insert, Delete and Search can be achieved in $O(1)$ time and Preprocessing involves initializing an empty memory of size $O(U)$ and additional $O(1)$ time.*

Proof. The algorithm is as follows (see Figure 1):

- Preprocess: initialize an array A of length U, with each entry being the start of an empty linked list.
- Insert: for a key-value pair (K, V), add (K, V) as the head of the linked list starting at $A(K)$.

- Delete: for a key K, delete the head of the linked list starting at $A(K)$ (if it exists).
- Search: for a key K, return the key-value pair at the head of the linked list starting at $A(K)$, if it exists. If not, return $\perp$.

Runtime: The costs associated to Preprocess are clear from the description. Insert/delete/search operations take $O(1)$ time as only the head of a linked list is manipulated. □

Essentially this data structure previously appeared in `SingletonBucketSort()`.

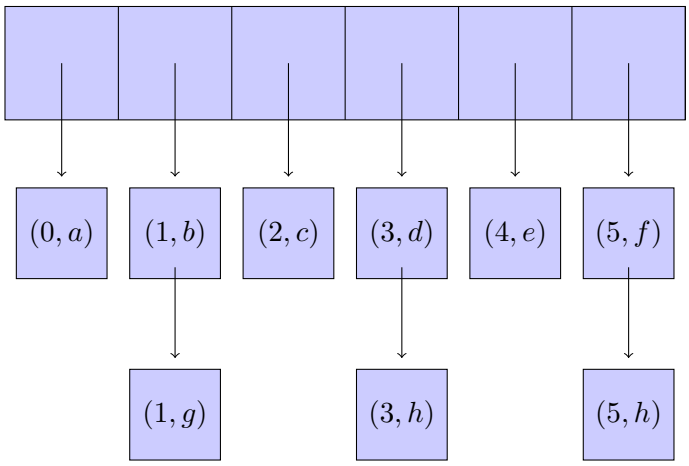

Figure 1: Deterministic data structure for Dynamic dictionaries for $U = 6$. A key-value pair (K, V) is inserted in the linked list starting at $A(K)$.

If U is very large in comparison to the maximum number of key-value pairs stored at any time, then the preprocessing cost of the above algorithm is prohibitive. The theorem below significantly reduces the cost of preprocessing in such a case, maintaining $O(1)$ cost of updates and queries in reasonable parameter regime, using randomness.

Theorem 3.2. *For each positive integer m, Dynamic Dictionaries has a Las Vegas randomized solution in which Insert, Delete, and Search can be achieved in $O(1 + \frac{n}{m})$ expected time and Preprocessing involves initializing an empty memory of size $O(m)$ along with additional $O(1)$ runtime. Here, n is the maximum number of key-value pairs in the data structure till the time under consideration.*

Whenever $m = \Theta(n)$, the expected runtime of updates and queries is $O(1)$. If n becomes larger, one has to update the data structure by choosing a larger value of m (we defer this discussion to the end of this Section). The data structure needed for this theorem is a variant of the data structure in 3.1 and called a hash table, as it makes use of *random hash functions.*

Random hash functions: These are a collection of functions $h_\ell : [U] \to [m]$, where $\ell \in [R]$ for some number R that can be large, with the following properties:

- *Efficient evaluation:* Given $\ell \in [R]$ and $K \in [U]$, evaluating $h_\ell(K)$ takes time $O(1)$.

- *Low chances of collision:* For any two $K, K' \in [U]$, the fraction of ℓ such that $h_\ell(K) = h_\ell(K')$ is $O(1/m)$. That is,

$$\Pr_\ell[h_\ell(K) = h_\ell(K')] = O\left(\frac{1}{m}\right)$$

when ℓ is chosen uniformly at random from $[R]$.

Remarks:

- Section 3.3 describes an example of the hash function, where one can choose $R = O(U^2)$. *This is an optional read.*
- The first property holds in the Word-RAM model when the word size w is large enough to store the elements of $[U]$. If $U > 2^w$, the runtime estimates would change to $((\log U)/w)^{O(1)}$.
- The second property allows hash functions to behave like completely random functions as far as any two inputs K, K' are considered: a completely random function from $[U]$ to $[m]$ would take K to a random number in $[m]$ and K' to another independent random number in $[m]$. The probability that the two numbers are equal would be exactly $1/m$.
- The first property makes hash functions significantly more useful that completely random functions from computational point of view. Choosing and storing a completely random function in the Word RAM model would take $\Theta((U \cdot \log m)/w)$ time and space, since there are about m^U possible random functions and it thus take $\log(m^U) = U \cdot \log m$ bits to describe one. This is close to the space cost in Claim 3.1 (even when $2^w \geq U$ so that elements of $[U]$ fit in one word), which we would like to avoid.

Proof of Theorem 3.2. The algorithm is as follows (see Figure 2) .

- Preprocess: initialize an array A of length m, with each entry being the start of an empty linked list. Choose and store a random $\ell = \texttt{random}(R)$, indexing a hash function h_ℓ from our collection.
- Insert: for a key-value pair (K, V), add (K, V) as the head of the linked list starting at $A(h_\ell(K))$.
- Delete: for a key K, consider the linked list starting at $A(h_\ell(K))$. Traverse this linked list from the head and delete the first key-value pair with key K (if it exists).
- Search: for a key K, consider the linked list starting at $A(h_\ell(K))$. Traverse this linked list from the head and return the first key-value pair with key K, if it exists. If no key-value pair with key K is found, return $\perp$.

Correctness: The algorithm correctly returns the search queries, by construction. Hence it is a Las-Vegas data structure.

Expected Runtime: The preprocessing step initializes an empty memory of size $O(m)$ and takes $O(1)$ step for choosing the random number. The runtime of insert operation is the time taken to evaluate $h_\ell(K)$ and add (K, V) at the start of the linked list, both of which is $O(1)$.

The runtime of delete and search operations is dominated by the time taken to traverse the linked list at $A(h_\ell(K))$ up until the key K is found (as above, other operations such as evaluating

$h_\ell(K)$ is $O(1)$ time). This time is $O(\text{length}_\ell(K))$, where $\text{length}_\ell(K)$ is the length of the linked list from the head to the first occurrence of key K at the location $h_\ell(K)$, under ℓ-th hash function. If we can show that the expected length

$$\mathrm{E}_\ell\left[\text{length}_\ell(K)\right] = O\left(1 + \frac{n}{m}\right),$$

the expected runtime bound concludes. We will skip the details of the proof, but note the high level idea: the probability that two keys go to the same index is $1/m$. Thus, the expected number of keys different from K that are in the same linked list as K is $O(n/m)$. This shows that the number of key-value pairs starting from $A(h_\ell(K))$ to K is $O(1 + n/m)$. A formal proof appears in Section 3.4, which is an optional read. □

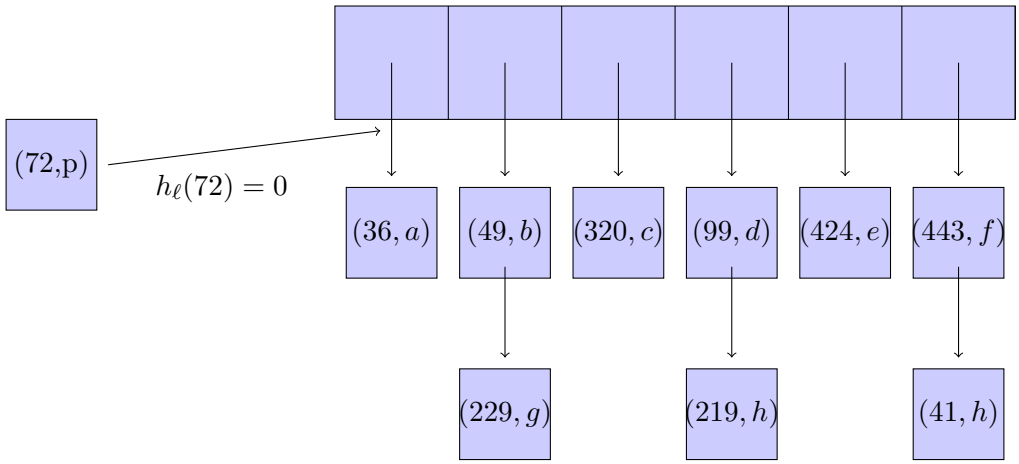

Figure 2: Hash table with $U = 601, m = 6$. A key-value pair (K, V) is inserted in the linked list starting at $A(h_\ell(K))$, for a randomly chosen ℓ.

Load of hash table: We call $\alpha = n/m$ the *load* of the data structure, so the expected runtime is $O(1 + \alpha)$. Notice that we get $O(1)$ expected time as long as $m = \Omega(n)$. To maintain the time efficiency, we need to tailor m to the size n of the dataset, which we may not know advance. This can be solved by rebuilding the data structure as the hash table gets too full. For example, when we reach load $\alpha = 2/3$, we can increase m by a factor of 2 (and use new hash functions) to bring us back to $\alpha = 1/3$. The cost associated to the rebuilding of data structure is proportional to the number of key-value pairs that it took to increase the load from $1/3$ to $2/3$, and hence the average cost per key-value pair (referred to as the *amortized cost*) is $O(1)$.

Need for randomization: One may wonder why a random choice of hash functions is needed in Theorem 3.2. The reason is that randomization is necessary for a fairly balanced hash table. If we use a fixed function $h(K) : [U] \to [m]$, we will have $\lceil U/m \rceil$ keys $K_0, K_1, \ldots$ with the same value $h(K_0) = h(K_1) = \ldots$ (since $U \gg m$). Assume $U > mn$ (similar argument can be made if $U \leq mn$) and consider a series of n insertions $(K_0, V_0), (K_1, V_1), \ldots (K_{n-1}, V_{n-1})$. This will lead to a linked list of length n in which (K_0, V_0) is at the tail of the linked list. If a search query for K_0 is now called, $O(n)$ time will be needed to traverse the linked list, which is much larger than $O(1 + \frac{n}{m})$ time guaranteed in Theorem 3.2.

In CS50, Hash tables are introduced as deterministic data structures. The goal there is similar to ours - keep the hash table balanced by 'fairly uniformly distributing the data'. The art there is to use a fixed hash function, so that 'typical' or 'real-world' sequences of key-value pairs get fairly uniformly distributed. However, as described above, a fixed function cannot achieve this for a worst-case dataset. Randomizing the hash function ensures that our expected time will be small even on worst-case datasets.

3.2 Storing and Searching Data Synthesis

Question: 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.

3.3 (Optional) Examples of Random 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 texts, and courses like CS 1240, CS 1270, CS 2220, CS 2230, and CS 2250.

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 if $p \leq 2^w$, 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 1270.) 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.

3.4 (Optional) Proof of runtime of Las Vegas data structure

This section is optional reading on the formal proof of runtime analysis, in case you are interested.

Let $\text{Ind}(j = h_\ell(K))$ be the indicator that a key $K \in [U]$ was hashed to the location $j \in [m]$. Let the keys in the data structure be $K_0, \ldots K_{n-1}$. The runtime bound on the delete and search operation on a key K (which may or may not be in the data structure) is obtained by upper bounding the length of the linked list at $h_\ell(K)$ from the start to the first occurrence of K - if it is in the data structure - or the length of the linked list at $h_\ell(K)$ - if K is not in the data structure (see the Proof of 3.2). That is, we need to consider the quantity:

$$\sum_{j\in[m]} \text{Ind}(j = h_\ell(K)) \cdot \left(1 + \sum_{i\in[n]:K_i\neq K} \text{Ind}(j = h_\ell(K_i))\right).$$

We have added $+1$ to account for the case where K is in the linked list; if K is not in the linked list, we still obtain an upper bound on the length. Taking expectation with respect to ℓ, we find that the expected length is upper bounded as

$$\begin{aligned}
&\sum_{j\in[m]} \left(\text{E}_\ell[\text{Ind}(j = h_\ell(K))] + \sum_{i\in[n]:K_i\neq K} \text{E}_\ell[\text{Ind}(j = h_\ell(K)) \cdot \text{Ind}(j = h_\ell(K_i))]\right) \\
&= \sum_{j\in[m]} \Pr(j = h_\ell(K)) + \sum_{i\in[n]:K_i\neq K} \sum_{j\in[m]} \text{E}_\ell[\text{Ind}(j = h_\ell(K)) \cdot \text{Ind}(j = h_\ell(K_i))] \\
&= 1 + \sum_{i\in[n]:K_i\neq K} \Pr(h_\ell(K) = h_\ell(K_i)) = 1 + \frac{n-1}{m} \leq 1 + \frac{n}{m}.
\end{aligned}$$