CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 9: Randomized Data Structures: Hash Tables

Harvard SEAS - Fall 2021 *Sept. 30, 2021*

1 Announcements

Recommended Reading:

- CLRS Sec 9.2
- Roughgarden II 12.0–12.4
- CLRS 11.0–11.4
- Add/Drop Monday
- PS4 posted, due Wed 10/6
- Salil OH today after class + Monday 1:30-2:30 zoom

2 Runtime of QuickSelect

Recall the QuickSelect algorithm:

```
1  QuickSelect(A, i)
   Input  : An array A = ((I_0, K_0), ..., (I_{n-1}, K_{n-1})), where each K_j ∈ ℕ, and i ∈ ℕ
   Output : An item key-pair (I_j, K_j) such that K_j is an i'th largest key.
2  if n ≤ 1 then return (I_0, K_0);
3  else
4      p = random(n);
5      P = K_p ;                                   /* P is called the pivot */
6      Let A_< = an array containing the elements of A with keys < P;
7      Let A_> = an array containing the elements of A with keys > P;
8      Let A_= = an array containing the elements of A with keys = P;
9      Let n_<, n_>, n_= be the lengths of A_<, A_>, and A_= (so n_< + n_> + n_= = n);
10     if i < n_< then return QuickSelect(A_<, i);
11     else if i ≥ n_< + n_= then return QuickSelect(A_>, i − n_< − n_=) ;
12     else return A_=[0];
```

Algorithm 1: QuickSelect

Theorem 2.1. *On every input array of length n, QuickSelect has expected runtime $O(n)$.*

In class, we are only going to sketch the proof of this analysis, since we will not be asking you to do sophisticated probability proofs during the course. For cs120, our goal is for you to understand the concept of randomized algorithms and why they are useful. You can develop more skills in

rigorously analyzing randomized algorithms in subsequent courses like CS121, CS124, and other CS22x courses. However, we will include a full proof in the detailed lecture notes in case you are interested.

Proof. Let $T(n)$ be the worst-case expected runtime of `QuickSelect()` on arrays of length n. Fix an array A of length $n \geq 1$. The proof combines the following three claims:

$$\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)\right] = \mathrm{E}_p\left[\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)|p\right]\right], \tag{1}$$

where $|p$ denotes conditioning on a fixed value of p.

$$\mathrm{E}_p\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)|p\right] \leq cn + \max\{T(n_<), T(n_>)\} = cn + T(\max\{n_<, n_>\}). \tag{2}$$

$$\mathrm{E}_p[\max\{n_<, n_>\}] \leq 3n/4, \tag{3}$$

Equation (1) is a general probability fact known as the Law of Iterated Expectations. Inequality (2) follows from inspection of the algorithm.

To prove Inequality (3), we observe that the lengths $n_<, n_>, n_=$ of the partitioned array depend only on $P = K_p$, and that the distribution of P remains the same regardless of the ordering of the elements of A. In particular, if we consider a *sorted* version $A' = ((I'_0, K'_0), \ldots, (I'_{n-1}, K'_{n-1}))$ of A and choose an index p' uniformly at random from $[n]$, then $P' = K'_{p'}$ has exactly the same distribution as P, and the lengths $n'_<, n'_>, n'_=$ of the corresponding partition of the sorted array has the same distribution as $n_<, n_>, n_=$.

With this choice, we observe that $n'_< \leq p'$ and $n'_> \leq n - p' - 1$, so we have the following (assuming n is odd for simplicity):

$$\begin{aligned}
\mathrm{E}_p[\max\{n_<, n_>\}] &= \mathrm{E}_{p'}[\max\{n'_<, n'_>\}] \\
&< \mathrm{E}_{p'}[\max\{p', n - p' - 1\}] \\
&= \frac{1}{n}\sum_{p'=0}^{n-1} \max\{p', n - p' - 1\} \\
&\leq \frac{2}{n}\sum_{q=(n-1)/2}^{n-1} q \\
&= \frac{2}{n} \cdot \frac{n-1}{2} \cdot \frac{3(n-1)}{4} \\
&\leq \frac{3n}{4}.
\end{aligned}$$

(The case of even n is similar.)

Equations/Inequalities (1), (2), and (3) suggest a recurrence like:

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

which would imply that

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

However, it is not true in general that for an arbitrary function f and random variable X that

$$\mathrm{E}[f(X)] = f(\mathrm{E}[X]) \tag{4}$$

(We are interested in $f = T$ and $X = \max\{n_<, n_>\}$.) However, this Equation (4) is true when the function f is linear, which is the bound we are trying to prove on T. So we can prove our desired bound that $T(n) \leq 4cn$ for all natural numbers $n \geq 1$ by induction on n.

For the base case, we have $T(1) \leq 4c$ by inspection (for a large enough choice of the constant c. For the induction step, assume that $T(k) \leq 4ck$ for all natural numbers $1 \leq k \leq n-1$, and let's prove that $T(n) \leq 4cn$ for $n \geq 2$. That is, we need to show that for every array A of length n, the expected runtime of QuickSelect on A is at most $4cn$. We do this as follows:

$$\begin{aligned}
\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)\right] &= \mathrm{E}_p\left[\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)|p\right]\right] && \text{(Equation (1))} \\
&\leq \mathrm{E}_p[cn + T(\max\{n_<, n_>\})] && \text{(Inequality (2))} \\
&\leq \mathrm{E}_p[cn + 4c \cdot \max\{n_<, n_>\})] && \text{(Induction Hypothesis, since } n_<, n_> < n) \\
&\leq cn + 4c \cdot \mathrm{E}_p[\max\{n_<, n_>\}] && \text{(Linearity of Expectations)} \\
&= cn + 4c \cdot (3n/4) && \text{(Inequality (3)} \\
&= 4cn,
\end{aligned}$$

as desired.

□

3 The Power of Randomized Algorithms

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: 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 determinstic one. You can learn more about this in courses like CS121, CS221, and CS225.

4 Randomized Data Structures

We can also 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 item–key pairs (like we've been studying) but where we are *not* interested in the ordering of the keys (so min/max/predecessor/successor/selection aren't relevant).

Updates :	Insert or delete an item-key pair (I, K) with $K \in \mathbb{N}$ into the set
Queries :	Given a key K, return a matching item–key pair (I, K) from the set (if one exists), or $\perp$ if none 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 — get time $O(1)$.

Let's assume our keys come from a finite universe U. Then we can get $O(1)$ time as follows:

A deterministic data structure:
Preprocess(U):
Initialize an array A of size U.
Insert(I, K):
Place (I, K) into the linked list at slot $A[K]$.
Delete(K):
Remove an element from the linked list at slot $A[K]$.
Search(K):
Return the head of the linked list at $A[K]$.

Problem: U can be very large. If we consider keys of 64 bit words (common in practice), we would need an array of size 2^{64}, which is completely infeasible.

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(I, K):
Place (I, K) into the linked list at slot $A[h(K)]$.
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 - if there are two distinct keys K_1, K_2 with $h(K_1) = h(K_2)$, on the query Search(K_1) we could return an item-key pair (K_2, I). To bound this error probability, consider a query Search(K). In order for us to return the wrong value, we must have inserted a distinct key that hashes to the same value as $h(K)$. If we've inserted items with keys $K_0, \ldots, K_{n-1}$ that are different than K, we have:

$$\begin{aligned}\Pr[\text{Search}(K) \text{ returns incorrect value}] &\leq \Pr[h(K) \in \{h(K_0), \ldots, h(K_{n-1})\}] \\ &\leq \sum_{i=0}^{n-1} \Pr[h(K) = h(K_i] \\ &= n \cdot \frac{1}{m}\end{aligned}$$

where in the final line we used that h was a random mapping from $[U]$ to $[m]$.

A Las Vegas data structure:
The same data structure as above, except the linked list at every array index stores (K, I) pairs. Then when we query Search(K), go to 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 $O(1 + n/m)$. Notice that here we get $O(1)$ expected time even if $n > m$, provided that $m = \Omega(n)$.