Dictionary

Definition

This data structure that does the same thing as a physical dictionary.

Let \( U \) be the universe of the strings with \( |U| = N \) (very large) and \( W \) (words to be stored in the dictionary) with \( W \subset U \)

The cardinality of \( U \) is much smaller when compared to the cardinality of \( U \) \( |W| = n << N \)

In addition, each element in \( W \) may have some additional data mapped to it

Following are the functions that are to be expected from the dictionary data structure:

**Lookup**: Given an element \( x \in U \),

- Determine if \( x \in W \)
- If so, look up the associated data

**Expected complexity:**

- Space complexity : We want the expected size of the data structure to be \( O(n) \)
- Time complexity: Lookup should be fast, ideally \( O(1) \) time

**Static Dictionary**: Given \( W \), build the dictionary \( SD \) based on it. Provides only lookups on the constructed \( SD \).

**Dynamic Dictionary**: In addition to building the dictionary and the lookups (that are provided the static dictionary), a dynamic dictionary should also support inserting and deleting entries \( x \in W \) in real time. This is also expected to be done reasonably fast.

Implementation

Easy ways to implement these, by the algorithms we already know:

- For the static dictionary, a sorted array should suffice with lookups done by binary search.
- For the dynamic dictionary, we might want a binary search tree which would support insert and delete.

Unfortunately, for both the implementations are guaranteed to be \( O(\log n) \). We now turn our attention to the idea of hashing which can provide us with faster lookup times.
Hashing

Hashing is a way of indexing into the array.

For hashing, we need would be needing a hash function \( h : U \rightarrow [m] \). We would store \( x \) at location \( h(x) \) and when we want to look-up \( x \), we again compute \( h(x) \) as our index into the array.

This way, time taken for lookup is equal to the time taken to compute the hash function \( h(x) \).

However not any function can be a hash function, we would need certain desired properties from a hash function:

- Firstly, the hash function has to be easy to compute
- It needs to have as few collisions as possible on \( W \) (The next section discusses the idea of collision)
- We should be able to record identifier (which \( h \) we are using) using as few bits as possible. (This idea will be made clear in the hash family section)

Collision

Collision is the entire problem of hashing summarized. Following is the definition of collision:

\[
h(x) = h(y); x \neq y; x, y \in W
\]

Further, in this course, we would look at how to deal with collisions. But for now, we would stick our discussion to what kind of \( h \) would lead us to as few collisions as possible.

We also note that when \( m \) (length of range of \( h \)) < \( |U| \), by pigeon principle there would some elements in \( U \) that would collide. So demanding no collisions from the hash function would be unreasonable, therefore we would bring down our demand to keeping the collision as low as possible.

By the discussion so far in the course, it is easy to see that any deterministic hash function doesn’t work as an adversarial input can be given a lot of input that would map to the same hash value. (Such an input can be constructed adversarially, as most of the languages we would consider as \( W \), are mostly structured. An example of \( W \) we might deal with is all the words in the English language, which by themselves are all not just a random bunch of letters)

**Fix:** The fix would be an \( h \) that behaves randomly, in which no adversarial input (as discussed above) can be generated.

If we assume \( h \) be a uniform random function:

\[
h : U \rightarrow [m]
\]

Expected number of collision = \( E_N[\#\{x, y \in W; h(x) = h(y); x \neq y\}] \)

\[
= \sum_{x \neq y \in W} 1_{h(x) = h(y)}
\]
\[
\sum_{x \neq y \in W} \frac{1}{m} = \binom{n}{2}/m \approx \frac{n^2}{m}
\]

**Birthday Paradox:** This problem of collision is similar to that of birthday problem or birthday paradox. Birthday paradox concerns the probability that, in a set of \( n \) randomly chosen people, some pair of them will have the same birthday. The solution of the problem there would be more than one collision expected if there are more than \( \sqrt{m} \) people, where \( m \) is the number of possible birthdays which is 365 here.

Another corollary of above is if \( n \approx m \), then we would end with \( O(n) \) collisions.

Another thing to note is that \( h \) has to a consistent function i.e., for a given \( x \), we should get the same \( h(x) \) every time.

For \( h \) we assumed above as consistent uniform random function, it becomes hard to store as we need to store the mapping from every possible \( x \) to \( h(x) \). The way we solve this leads to the next part

**Hash Family**

**Definition:** Probability distributions over functions \( h : U \rightarrow [m] \) with following desired properties:

- We want an easy way to store an element of this family (identifier)
- We should be able to evaluate \( h \) given identifier and input quickly
- In addition, we would also need some guarantee. (A guarantee that could get through our analysis that the collisions are low, like our analysis for uniform random function)

**K-wise independence**

This is a stronger guarantee in the context of the guarantee that was talked about earlier

**Definition:** For distinct \( X_1, X_2, ... X_k \in U \) and \( Y_1, Y_2, ... Y_k \in [m] \), \( Pr_{h}(h(X_1) = Y_1 \& h(X_2) = Y_2 ... \& h(X_k) = Y_k) = 1/m^k \)

For 2-wise independence \( P(h(x) = h(Y)) = 1/m \), which gets through the analysis of expected number of collisions.

**Standard \( k \)-wise independent family:** If \( U = F_q \& [m] = F_q \), then \( h(X) \) is a random \( k-1 \) degree polynomial of \( x \).

**Standard polynomial interpolation:** Given \( X_1, ... X_k \) and \( Y_1, ... Y_k \), there exists a unique \( k-1 \) degree polynomial \( P \) such that \( P(X_1) = Y_1, ... P(X_k) = Y_k \)

For the analysis of the number of bits needed in specifying the polynomial (identifier) to the hash function \( h \), assume that \( U \subset F_{2^t} \) then \( h \) would be \( h : U \rightarrow F_{2^t} \). As range of \( h \) would not include as many elements as \( U \), we can throw away some bits and retain \( s \), where \( s \leq t \).

Now, the bits needed to specify \( h \) is \( K * log(N * m) \leq K * (s + t) \)