Algorithms and Data Structures

Chapter 11

Catherine Durso

cdurso@cs.du.edu

Chapter 11

Stacks, queues and lists support $\Theta(1)$ insertion, and deletion. However, search is a worst case $\Theta(n)$ operation, where n is the number of items in the dynamic set. In this chapter we will study a data structure, the hash table, that, under reasonable assumptions, has $\Theta(1)$ expected search time. Hash tables are well-suited to implementing dictionaries, particularly if deletions are rare. For example, the map data type may be implemented with a hash table.

A hash table basically augments the structure of an array by storing data in indices computed from the data's keys, assumed to be distinct for distinct data items.

Warning

In this chapter only, array indices start at 0.

Direct Address Table

If we have a set of keys $U = \{0, 1, 2, ...m - 1\}$, and the number of values that we plan to store is less than m, but on the order of m, we can create an array T of length m, initialized to hold all NILs. We then populate T with records by storing the record with the key k in $T[k]$.

How are insertion, deletion, and search performed? What asymptotic behavior do they have?

Direct Address Operations

```
DIRECT-ADDRESS_SEARCH$(T, k)$
1.    return $T[k]$

DIRECT-ADDRESS_INSERT$(T, x)$
1.    $T[x.key] = x$

DIRECT-ADDRESS_DELETE$(T, x)$
1.    $T[x.key] =$ NIL
```

Large U

If U is large, storing a table of size $|U|$ may be impractical. If U is large relative to the total number of items, a direct adress table of size $|U|$ may be a poor use of memory. For example, storing data for 30 employees in a direct address table by social security number seems a bit silly.

Instead of directly using the key as an index, a hash table of size m uses a ***hash function*** $h : U \rightarrow \{0, 1, 2, ..m - 1\}$ to calculate an index for each key.

Division Method

One way to define h is by the ***division method***. if $U \subseteq \mathbb{Z}$, and the table is of size m, take $h(k) = k \mod m$. A prime not to close to an exact power of 2 is often a good choice for m. (What would determine $h(k)$ if $m = 2^p$?)

This form of h can be evaluatated quickly.

Example

Say $U = \{1, 2, ..10^9\}$, and we expect to store about 40 items. Choose $m = 83$, say.

$$
\begin{array}{ll}
k = 18 & h(k) = 18 \\
k = 115 & h(k) = 32 \\
k = 74062 & h(k) = 26 \\
k = 848 & h(k) = 18
\end{array}
$$

Is there a problem?

Collisions

If the size of the hash table is less than the size of the universe, then there will be distinct keys that hash to the same place, ie $k_1 \neq k_2$ with $h\left(k_1\right) = h\left(k_2\right)$. This is called a ***collision***. To implement a hash table, one must resolve collisions, that is, determine a strategy for storing items with the same hash value.

Chaining

One simple approach to collision resolution is for each index j to hold the head of a linked list of the items with keys k having $h(k) = j$.

```
CHAINED-HASH-SEARCH$(T, k)$
1.      search the list $T[h(k)]$ for an element with key $k$
```

```
CHAINED-HASH-INSERT$(T, x)$
1.      insert $x$ at the head of the list $T\left[h\left(x.key\right)\right]$
```

```
CHAINED-HASH-DELETE$(T, x)$
1.      delete $x$ from the list $T\left[h\left(x.key\right)\right]$
```

Performance of Chaining

Given x as opposed to the key $x.key$ and doubly linked lists, what is the worst-case asymptotic bound on the time required to perform search, insertion, and deletion?

As ever, to calculate average case performance, we need a model for the frequency of different cases. The assumption that any given element is equally likely to hash to any of the m slots, independently of where other elements have hashed, is called the assumption of ***simple uniform hashing***.

Performance Bounds

With doubly linked lists, CHAINED-HASH-INSERT and CHAINED-HASH-DELETE have $\Theta(1)$ worst-case time bounds. For CHAINED-HASH-SEARCH, the worst-case time bound is $\Theta(n)$ where n is the number of items in the dynamic set.

In terms of the load factor $\alpha = \frac{n}{m}$, under simple uniform hashing, the expected running time of an unsuccessful search is $\Theta(1+\alpha)$. The expected running time of a successful search under these assumptions is also $\Theta(1+\alpha)$. (Calculations follow.)

Expected Chain Length

Let n_j denote the length of the list $T\left[j\right]$. If j is equally likely to be any index in $\{0, 1, ..m - 1\}$, then
$E\left[n_j\right] = \sum_{i=0}^{m-1} n_i Pr(i)$
$= \sum_{i=0}^{m-1} n_i \frac{1}{m} = \frac{1}{m} \sum_{i=0}^{m-1} n_i = \frac{n}{m}$

Unsuccessful Search

Under simple uniform hashing, constant time evaluation of $h(k)$, and collision resolution by chaining, an unsuccessful search takes $\Theta\left(1+\alpha\right)$ time.

The unsuccessful search for a key k goes to the end of the $T\left[h\left(k\right)\right]$ list. The expected number of elements in this list is $\alpha = \frac{n}{m}$ by the argument above. The time to evaluate $h(k)$ is constant. Thus the expected time to find and traverse the $T\left[h\left(k\right)\right]$ list is $\Theta\left(1+\alpha\right)$.

Successful Search Setup

Under the assumption of simple uniform hashing, the assumption that any key in the table is equally likely to be the query key, and using collision resolution by chaining, a successful search takes average-case time $\Theta\left(1+\alpha\right)$.

The number of elements examined in the search of the $T\left[h\left(x.key\right)\right]$ list equals the number of elements inserted into $T\left[h\left(x.key\right)\right]$ after x.

Let x_i be the ith element inserted. Let $k_i = x_i.key$. For k_i and k_j, define the indicator random variable $X_{ij} = X_{\{T:h(k_i)=h(k_j)\}}$ on the probability space of tables formed by inserting n items under simple uniform hashing.

Successful Search Expectation

Because $Pr\left(h\left(k_i\right) = h\left(k_j\right)\right) = \frac{1}{m}$, $E\left[X_{ij}\right] = \frac{1}{m}$.

For a fixed k_i, the number of elements searched to find k_i is $1 + \sum_{j=i+1}^{n} X_{ij}$.

Any k_i is equally likely to be the query term, so the expectation over i and T is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E\left[X_{ij}\right]\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right)$$

some algebra

$$
\begin{aligned}
& E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
& = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
& = \frac{1}{n}\left(n+\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
& = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
& = 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
& = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) = 1-\frac{1}{nm}\left(\frac{2n^2-n^2-n}{2}\right) \\
& = 1+\frac{n^2-n}{2nm} = 1+\frac{n-1}{2m} \\
& = 1+\frac{\alpha}{2}-\frac{\alpha}{2n} = \Theta(1+\alpha)
\end{aligned}
$$

Implications for Hash Tables

If $n = O(m)$, that is, the number of items is bounded by a fixed multiple of the number of hash table slots, then $\alpha = O(1)$. Under these circumstances, and under the assumption of simple uniform hashing, CHAINED-HASH-SEARCH has $\Theta(1)$ expected running time.

The $\Theta(1)$ expected running time makes hash tables very useful in algorithms requiring fast data look-up on average.

For data that is not dynamic, ***perfect hashes***, hashes with no collisions, can be found fairly efficiently (Ch. 11, section 5, not covered in this course).

Assumption Violation

That bound is on expected running time, not worst case running time. For example, a couple years ago, it was noted (Klink and Wälde) that a number of languages used in web development used very predictable hash functions. Attackers could mount a Denial of Service attack by submitting POST form data that would cause the hash tables used to store the data to degenerate, slowing hash table operations.

In such an attack, the assumption of simple uniform hashing is dramatically false. Randomized hash functions, such as those provided by ***universal hashing***, are a defense against this type of attack.

Hash Functions

Ideally, a hash function approximates simple uniform hashing.

We have already mentioned the division method.

The multiplication method uses a real number $A \in (0, 1)$ along with the table size m. Given A and m, define the hash function $h(k) = \lfloor m (Ak - \lfloor Ak \rfloor) \rfloor$.

(This can be calculated efficiently using bit operations if $m = 2^p$ for some positive integer p, and $A = \frac{s}{2^w}$ where w is the word length and s is an integer with $0 < s < 2^w$. Then $sk = r_1 2^w + r_0$, so $Ak - \lfloor Ak \rfloor = r_0 2^{-w}$ and $\lfloor m (Ak - \lfloor Ak \rfloor) \rfloor = \lfloor r_0 2^{p-w} \rfloor$, the p most significant bits of r_0. $A \approx \left(\sqrt{5} - 1\right)/2$ has some credibility as a good choice.)

Open Address Hashing

In ***open addressing***, all keys are stored in the table itself. This is accomplished by there being a sequence of locations for each key, $h(k, i)$, $i \in \{0, 1, ..m - 1\}$. A key k is stored in T in the first unoccupied location in the sequence $h(k, i)$.

Probe Sequence Calculation

Standard methods for computing a probe sequence $h(k,0), h(k,1)...h(k,m-1)$ include linear probing: $h(k,i) = (h'(k)+i) \mod m$, quadratic probing: $h(k,i) = \left(h'(k)+c_1 i + c_2 i^2\right) \mod m$ for carefully chosen c_1 and c_2, and double hashing: $h(k,i) = (h_1(k) + i h_2(k)) \mod m$.

How would you search an open address hash table?

Search

To search for k, one follows the same sequence, $h(k, 0), h(k, 1) \ldots$, until finding k (successful search), or an empty slot (unsuccessful search).

How would you delete an item?

Deletion

Deletion is imperfect. Typically, the slot occupied by the deleted item is just marked as deleted. When too much 'dead wood' builds up, the table is rehashed.

Probe Sequence Issues

Linear probing, while straightforward, has the drawback of primary clustering. Under simple uniform hashing, the slot after a run of r occupied slots will be hashed to with probability $\frac{r+1}{m}$. Long runs of occupied slots get longer, and performance degrades.

Example: use linear probing with the division method to insert 14, 22, 3, and 70 into a hash table with $m = 11$.

Quadratic Probe Example

In order for quadratic probing:
$h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \mod m$ to take full advantage of the table, $h\left(k,i\right)$for $i \in \{0,1,2..m-1\}$ must be m distinct positions. If $m = 2^p$, $c_1 = \frac{1}{2}$ and $c_2 = \frac{1}{2}$, this will in fact be the case (problem 11-3).

Double Hashing Example

In order for double hashing: $h(k,i) = (h_1(k) + ih_2(k)) \mod m$ to take full advantage of the table, $h\,(k,i)$for $i \in \{0, 1, 2..m-1\}$ must be m distinct positions. If $h_2(k)$ and m are relatively prime, this will be true. This can be accomplished by setting $m = 2^p$ and arranging for $h_2(k)$ to be odd for all k.

Alternatively, one can take m to be prime, $h_1(k) = k \mod m$, and $h_2(k) = 1 + k \mod (m-1)$.

Analysis

The assumption of uniform hashing states that the probe sequence of k is equally likely to be any of the $m!$ possible probe sequences. Under this assumption, with a load factor $\alpha < 1$, the expected number of probes for an unsuccessful search is at most $\frac{1}{1-\alpha}$. Hence this is also the expected number of probes to insert k.

The expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha}\right)$, assuming uniform hashing, and that each key in the table is equally likely to be searched for.

Note that none of our methods satisfy the assumptions. Empirically, the performance of double hashing tends to be close to these bounds.