Hashing

David Fernández-Baca

Computer Science 311
Iowa State University

February 3, 2022

Dynamic Sets

Notation

- T is a table.
- k denotes a key value.
- x is a reference to an object
- $x.\texttt{key}$ denotes the key of x.

Operations

Search(T, k): Return a reference x to an object in T such that $x.\texttt{key} = k$. Return `null` if no such object exists.

Insert(T, x): Insert object referenced by x into T.

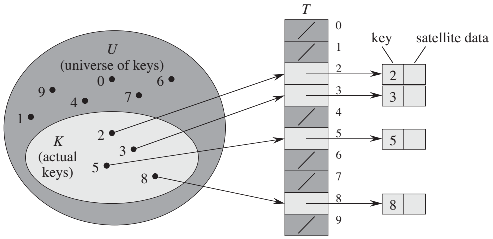
Delete(T, x): Remove object referenced by x from T.

Direct Address Tables

Direct-Address Tables

- Suppose each element has a distinct key drawn from the universe $U = \{1, \ldots, m\}$, where m is not too large.
- To represent a dynamic set, use an array $T[0 \mathinner{..} m-1]$where each slot corresponds to a key in U.
- Slot k points to an object in the set with key k. If the set contains no element with key k, then $T[k] = \texttt{null}$.

Direct-Address Tables



Source: CLRS

```
Search($T, k$)
    return $T[k]$

Insert($T, x$)
    $T[x.\texttt{key}] = x$

Delete($T, x$)
    $T[x.\texttt{key}] = \texttt{null}$
```

Direct-Address Tables

Upsides

- Simple.
- Each operation takes $\Theta(1)$ time in worst case.

Downsides

- If the universe U is large, storing a table T of size U may be impractical or impossible.
- If the set of keys actually stored is very small relative to U, most of the space allocated for T would be wasted.

Hash Tables

Hash Tables

- Frequently, the set K of keys is much smaller than $|U|$.
- A hash table reduces the storage to $\Theta(|K|)$, while maintaining $\Theta(1)$ average time.

Hash Functions

Definition

A hash function is a mapping $h : U \to \{0, 1, \ldots, m-1\}$, from the universe U of keys into the slots of a hash table $T[0 \mathinner{.\,.} m-1]$.

Idea

- Store element with key k in slot $h(k)$.
- Reduces the range of array indices, allowing $m << |U|$.

Terminology

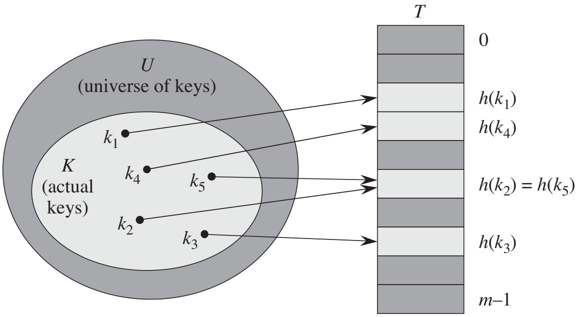
- The element with key k hashes to slot $h(k)$.
- $h(k)$ is the hash value of key k.

Assumption

Computing $h(k)$ takes $O(1)$ time.

Collisions

- A collision occurs when two keys k_1, k_2, $k_1 \neq k_2$ hash to the same slot; i.e., $h(k_1) = h(k_2)$.
- A good hash function can minimize the number of collisions.
- Impossible to avoid collisions altogether, though, because $|U| >> m$.



Source: CLRS

Handling Collisions

Chaining. Items with same key are stored in a list.

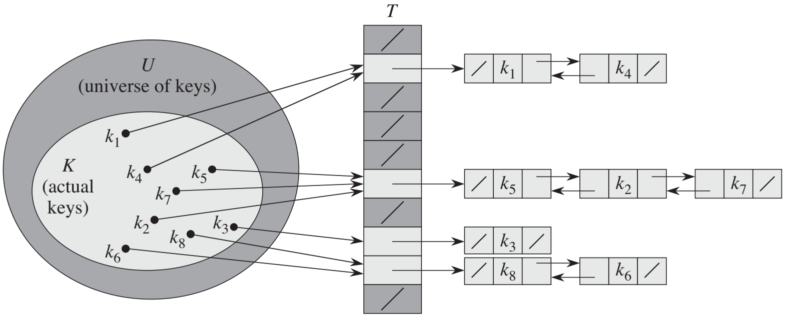
Covered next.

Open addressing. Scan table until empty slot is found.

Not covered in class — see CLRS if interested.

Chaining

- All elements that hash to the same slot go into the same linked list.
- Slot j contains a pointer to the head of the list of all stored elements that hash to j.
- If there are no such elements, slot j contains `null`.



Source: CLRS

- Used in Java.

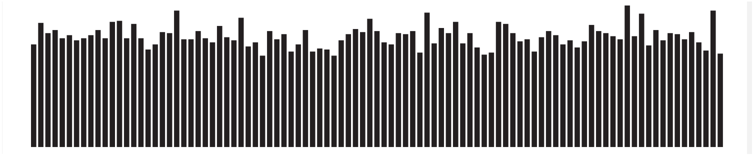
```
Search($T, k$)
    search for an element with key $k$ in list $T[h(k)]$.

Insert($T, x$)
    insert $x$ at the head of list $T[h(x.\texttt{key})]$

Delete($T, x$)
    delete $x$ from list $T[h(x.\texttt{key})]$
```

Analysis of Hashing with Chaining

Performance of Hashing with Chaining

Uniform Hashing Assumption

Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.



Hash value frequencies for words in Tale of Two Cities ($m = 97$)

Source: SW

Performance of Hashing with Chaining

Definition

Let T be a hash table with m slots string a total of n items. The load factor for T is the value $\alpha = n/m$.

Note

The default load factor in Java is $\alpha = 0.75$.

Lemma

In a hash table in which collisions are resolved by chaining, the expected length of the list stored in $T[j]$ is α, for each $j \in \{0, 1, \ldots, m-1\}$.

Theorem

In a hash table T in which collisions are resolved by chaining, the expected time for an unsuccessful search is $\Theta(1+\alpha)$, assuming simple uniform hashing.

Proof.

- Assuming simple uniform hashing, any key k not in T is equally likely to hash to any of the m slots.
- The expected time to search unsuccessfully for k is the expected time to search to the end of list $T[h(k)]$.
- The expected length of $T[h(k)]$ is α.
- Thus, the expected number of elements examined in an unsuccessful search is α.
- Hence, the total time (including the time to compute) $h(k)$ is $\Theta(1+\alpha)$.

□

Theorem

In a hash table in which collisions are resolved by chaining, the expected time for a successful search is $\Theta(1 + \alpha)$, assuming simple uniform hashing and that the element being searched for is equally likely to be any of the n elements stored in the table.

Lemma

The expected number of elements examined in a successful search for an element x in a hash table T is 1 plus the expected number of elements added to x's list after x was added to the list.

Proof.

- Let L be the list in slot $T[h(x.\texttt{key})]$.
- The number of elements examined during a successful search for x is $1 + \#$(elements that appear before x in L).
- Because new elements are placed at the front of L, elements before x in L were all inserted after x.

$\square$

Corollary

The expected number of elements examined in a successful search is the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list.

- Let x_i be the ith element inserted and $k_i = x_i.\texttt{key}$.
- For keys k_i and k_j, define the indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{If } h(k_i) == h(k_j), \\ 0 & \text{otherwise} \end{cases}$$

- By previous corollary, the expected number of elements examined in a successful search is

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

- By the uniform hashing assumption,

$$\Pr[h(k_i) = h(k_j)] = \frac{1}{m} \quad \Rightarrow \quad E[X_{ij}] = \frac{1}{m}.$$

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

Summary

- `Insert` takes $O(1)$ worst-case time.
- `Delete` takes $O(1)$ worst-case time if the lists are doubly linked and we are given a reference x to the object being deleted..
- Since $m \geq n$, $\alpha = n/m = O(1)$.
- Thus, `Search` takes $O(1)$ expected time.

Conclusion

Under the uniform hashing assumption, each hash table operation takes $O(1)$ expected time.

Hash Functions

Writing a Good Hash Function

A good hash function should

- be deterministic — equal keys must produce the same hash value —,
- be efficient to compute, and
- satisfy the uniform hashing assumption (within reason).

Bad Hash Functions

Bad Idea 1

Use the first three letters of a word, in a table with 263 buckets.

Why it is bad

- Distribution of keys is likely not to be uniform,
- Words beginning with `pre` are much more common than words beginning with `xzq`, so the former will be bunched up in one long list.

Bad Hash Functions

Bad Idea 2

Sum up the ASCII values of the characters.

Why it is bad

- The sum will rarely exceed ≈ 500, so most of the entries will be bunched up in a few hundred buckets.
- Anagrams like `pat`, `tap` and `apt` will collide.

Writing a Good Hash Function

General Principle

The hash code produced by the hash function should incorporate all the data in the key.

For instance, Java's hash function for strings uses all the characters, as well as their order.

```
hashCode()
    hash = 0
    for $i = 0$ to length() do
        hash = (hash × 31) + charAt($i$)
    return hash
```

Writing a Good Hash Function

We can use the same general principle to get hash functions for more complex objects, with multiple instance variables.

```
hashCode()
    hash = some initial value
    foreach instance variable $v$ used by equals() do
        $c = v$.hashCode()
        hash = (hash $\times$ 31) + $c$
    return hash
```

The Multiplication Method

$$h(k) = \lfloor m \cdot (\text{fractional part of } kA) \rfloor,$$

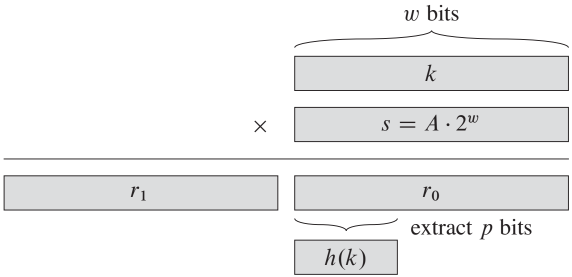
where

$$\text{fractional part of } kA = kA - \lfloor kA \rfloor.$$

- Choose $m = 2^p$, for some integer p.
- Suppose that word size is w bits and that k fits into a single word.
- Choose $A = s/2^w$, where $0 < s < 2^w$.
- $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$ appears to be a good choice.

Implementing the Multiplication Method

- Multiply k by the w-bit integer $s = A \cdot 2^w$.
- Result is a $2w$-bit value $r_1 2^w + r_0$.
- $h(k)$ consists of the p most significant bits of r_0.



Source: CLRS

Bloom Filters
(Extra Topic)

Operations on a Bloom Filter

There are two $O(1)$-time operations:

`Search(`k`)`: Returns `false` if k is not in the set; `true` otherwise*.

`Insert(`k`)`: Inserts k into the set.

No `Delete` operation.

Caveat

No False Negatives: If `Search(`k`)` returns `false`, then k is definitely not in the set.

False Positives are Possible: If `Search(`k`)` returns `true`, then there is a small probability that k is not in the set.

When to use Bloom Filters

Bloom filters are good when fast lookup is needed for a set of dynamically growing set of objects, space is at a premium, and a small number of false positives can be tolerated.

Sample Applications

- Spell checkers
- Maintaining sets of forbidden passwords
- Maintaining sets of blocked IP addresses

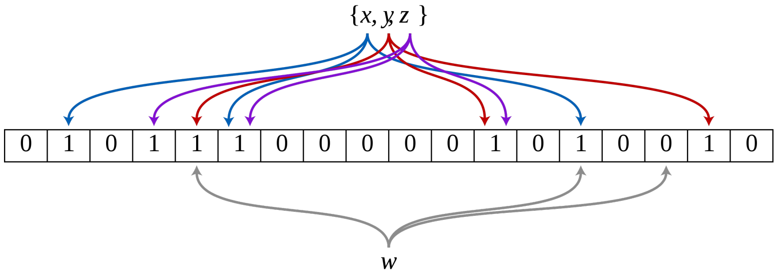
Bloom Filter Data Structures

- Maintain an m-bit array A.
- Initially, $A[i] = 0$, for $i = 0, \ldots m - 1$.
- There are r hash independent functions $h_1, \ldots, h_r$.
- For $i = 1, \ldots, r$, h_r maps key k to a number between 0 and $m - 1$.
- No keys or objects are actually stored.

```
Search($k$)
    for $i = 1$ to $r$ do
        if $A[h_r(k)] == 0$ then
            return false
    return true
```

```
Insert($k$)
    for $i = 1$ to $r$ do
        $A[h_r(k)] = 1$
```

Example



Source: Wikipedia

Heuristic Analysis

Notation

Let n be the number of keys. The bits-per key storage of a Bloom filter is

$$b = \frac{m}{n}.$$

Want $b \leq \#$(bits needed to represent an object or a pointer to an object).

Assumptions

1. h_i satisfies the uniform hashing assumption, for $i = 1, \ldots, r$.
2. h_i and h_j are independent of each other, for $i \neq j$.

Summary of Results of Heuristic Analysis

The false positive probability is approximately

$$\left(1 - e^{-\frac{r}{b}}\right)^r. \tag{1}$$

(1) is minimized when $r = (\ln 2) \cdot b \approx 0.693 \cdot b$, so

$$\text{false positive probability} \approx \left(1 - e^{-\ln 2}\right)^{(\ln 2)\cdot b} = \left(\frac{1}{2}\right)^{(\ln 2)\cdot b}.$$

False positive probability decreases exponentially fast with b:

b	8	16	32
r	6	12	23
False Positive Probability	2.14%	0.046%	0.00002%

Bibliography

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (3rd edition), MIT Press, 2009.

[R] Tim Roughgarden. *Algorithms Illuminated Part 2: Graph Algorithms and Data Structures*, 2018.

[SW] Robert Sedgewick and Kevin Wayne, *Algorithms* (4th edition), Addison-Wesley, 2011.