# W4231: Analysis of Algorithms

9/28/1999

- Design and Analysis of Data Structures
- Hash Tables



## Representing Sets

We look at data structures that represent collections $a_1, \ldots, a_n$.

Each $a_i$ is an element of some fixed data type having a unique integer key.



Important operations:

1. Insert.
2. Delete element(s) with key $x$.
3. Find element(s) with key $x$.
4. Find element with minimum key.
5. Find/delete most recently inserted element.
6. Find/delete least recently inserted element.



## More general scenario: union

For certain algorithms, we want to maintain simultaneously several sets.

For each set, we are interested in the typical set operations.

In addition we want a *union* operation between sets.



## Abstract Data Structures We Consider

- Set (Dictionary): `insert`, `delete`, `find`.
- Ordered Set: `insert`, `delete`, `find`, `find-min`.
- Priority queue: `insert`, `delete-min`, `find-min`.
- Priority queue + union: `insert`, `union`, `delete-min`, `find-min`, `increase-key`.



Other abstract data structures:

- Stack: `insert`, `find`/`delete` most recent.
- Queue: `insert`, `find`/`delete` least recent.

## Simple Implementations

**Set with a linked list.** `insert` $O(1)$. All the others $O(n)$.

**Stack with a vector or with a linked list.** `insert` $O(1)$. `find`/ `delete` most recent $O(1)$. Other operations not supported.

**Queue with a linked list and two pointers.** `insert` $O(1)$. `find`/ `delete` least recent $O(1)$. Other operations not supported

**Heap.** `insert` $O(\log n)$. `find-min` $O(1)$. `delete-min` $O(\log n)$.



## More Advanced Data Structures

**Hash Tables.** `insert`, `find`, `delete` $O(1)$ time *on average*.

**Balanced Search Trees.** All set operations $O(\log n)$ time.

**Binomial Heap.** Priority queue + union operations $O(\log n)$ time.

**Fibonacci Heap.** Priority queue + union operations $O(1)$ time, except `delete` $O(\log n)$ time , *amortized*.



## General Picture

| Abstract data structure | Algorithmic implementation | Performance |
|---|---|---|
| Set | List<br>Balanced tree<br>Hash table | up to $O(n)$ worst case<br>$O(\log n)$ worst case<br>$O(1)$ average case |
| Ordered Set | List<br>Balanced tree | up to $O(n)$ worst case<br>$O(\log n)$ worst case |
| Priority Queue | List<br>Balanced tree<br>Heap | $O(n)$ worst case for insert<br>$O(\log n)$ worst case<br>$O(\log n)$ |
| Priority Queue + union | List<br>Binomial heap<br>Fibonacci heap | $O(n)$ worst case<br>$O(\log n)$ worst case<br>$O(1)$, $O(\log n)$ `delete`, amortized |



## Lower Bounds

Assuming that the keys associated to the items are only accessed using *comparisons*, then

- It is impossible to implement all Priority queue operations in $o(\log n)$ time.
  [One can sort $n$ items using $n$ `insert`, $n$ `find-min`, and $n$ `delete-min`]
- `find` requires $\Omega(\log n)$ time.



## Hash Tables

Implement the **Dictionary** abstract data structure.

- `insert`. $O(1)$ worst case.
- `delete`. $O(1)$ average case. $O(n)$ worst case.
- `find`. $O(1)$ average case. $O(n)$ worst case.

Each entry in the dictionary is (or contains) an **integer key** in the range $1, \ldots, M$.



## Avoiding the comparisons-only lower bound

Hash tables avoid the lower bound because they use the value of the key to do addressing.

Let's see an oversimplified (and inefficient) way of doing so.

Maintain a vector with M entries. Each entry is a pointer. Initialize all the entries to NIL.

`insert`(a): set the $(a.key)$-th entry of the vector to point to $a$.

`delete`(k): set the $k$-th entry of the vector to NIL.

`find`(k): output the $k$-th entry of the vector.

## Problems

- Initialization takes $O(M)$ time. This can be avoided (see homework 2).
- Memory use is $O(M)$. If keys are 32-bits integers we are already in trouble. If keys are strings of up to 80 characters, and each character is represented in ASCII, and we use the standard representation of strings as integers, then $M = (256)^{80} = HUGE$.

  We like much better data structures using $O(n)$ memory, where $n$ is the (maximum) number of stored elements.



## Hash Functions

We represent the dictionary using an array $T$ of $m$ entries, where $m$ is much smaller than $M$ (and around $n$).

A hash function is a function $h : \{1, \ldots, M\} \to \{1, \ldots, m\}$. We'll see later how to choose $h$ intelligently.

An element $a$ is stored in position $h(a.key)$.



## Content of $T$

As before, we'd like $T$ to be a vector of (pointers to) elements of the set.

This creates ambiguity if there are $k, k'$ such that $h(k) = h(k')$. Since $m < M$, such ambiguous pairs must exist (regardless of the intelligent choice of $h$).

Then, we let $T$ be a vector of *lists*: for each $i$, $T(i)$ contains the list of elements $a$ in the dictionary such that $h(a.key) = i$.



## `insert`, `find`, `delete`

`insert`$(a)$. Compute $i = h(a.key)$; insert $a$ in the list $T[i]$.

`find` $(k)$. Compute $i = h(k)$; look for an $a$ with $a.key = k$ in the list $T[i]$.

`delete`$(k)$. Compute $i = h(k)$; look for an $a$ with $a.key = k$ in the list $T[i]$, and delete it from the list.



## Worst Case Analysis

Assuming that computing the hash function takes constant time.

**Insert** always takes constant time.

Let $l_1, \ldots, l_m$ be the length of the lists $T[1], \ldots, T[m]$ at a given moment.

Then `find` and `delete` on a key $k$ takes $O(1 + l_{h(k)})$ time in the worst case.



## An example of a hash function

$$\boxed{h(x) = x \bmod m}$$

Empirically:

- It's better that $m$ be prime.
- It's better that $m$ not be close to a power of two.

If $m$ is a power of two, then binary strings with the same suffix are mapped in the same entry of the table. This is bad! Having $m$ prime avoids other potential conflicts.

## Properties

If keys $x_1, \ldots, x_n$ are uniform and independent, and $M$ is a multiple of $m$, then the outputs $h(x_1), \ldots, h(x_k)$ are uniform and independent.

If $M$ is not a multiple, then one has "almost" uniformity.

[Note: It is possible that all the items we want to put in the table end up in the same entries of $T$. E.g. consecutive powers of $m$.]



## Average-Case Analysis

Consider a series of `insert`/`delete`/`find` where each item to be processed has a key randomly and uniformly distributed in $\{1, \ldots, M\}$.

Assume that $h$ has the property that if $x$ is uniformly distributed in $\{1, \ldots, M\}$, then $h(x)$ is uniformly distributed in $\{1, \ldots, m\}$.

Suppose at a given time there are $n$ elements in the set.

Call $\alpha = n/m$.

Then `delete` and `find` are done in expected time $O(\alpha)$.



## Proof

If we want to `find`/`delete` an item with key $k$, it takes time $\leq c(1 + l_{h(k)})$, where $c$ is a constant.

If the key $k$ is uniform, then so is $h(k)$, so we have that the expected time is at most

$$\sum_{i=1}^{m} \frac{1}{m} c(1 + l_i) = c + c\frac{1}{m}\sum_i l_i$$

$$= c + cn/m = c + c\alpha = O(\alpha)$$



## Load Factor

The ratio $\alpha = n/m$ is called the load factor of the table.

Small $\alpha$ correspond to faster average query, but also to wasted space.

Typical values are $1/2 \leq \alpha \leq 3$.

Once we *decide* the load factor and we *know* the final value of $n$ we can fix $m$ and create the table.

Hash tables are problematic when $n$ is unknown.



## Other Implementation

Instead of having $T$ be a vector of lists, we can let $T$ be a vector of items.

The hash function $h(\cdot, \cdot)$ now takes two parameters.

When inserting $a$ into $T$, we put it in position $T(h(1, a.key))$, if it is free.

Otherwise we try $T(h(2, a.key))$, and so on.

`find` and `delete` are similar.

Advantage: no need for the additional memory needed to store lists. Disadvantage: even `insert` has now worst-case $O(n)$.