

## Dictionary with Hashing

This data structure stores a set of object, and we have a function or attribute  $\text{key}(x)$  that identifies object  $x$ . Implementing it by a hash table with chaining allows the following operations to be efficient:

- `Insert(D, a)` insert object  $a$  to dictionary  $D$ , time  $O(1)$  on the average
- `Delete(D, a)` delete object  $a$  from dictionary  $D$ , time  $O(1)$  on the average
- `Member(D, x)` return the pointer to object  $a$  in dictionary  $D$  such that  $\text{key}(a) = x$ , or `NULL` if no such object exists, time  $O(1)$  on the average
- `ListAll(D)` provide pointers to all objects in  $D$ , time  $O(n)$  if  $D$  stores  $n$  objects

### Support in Perl and Awk

In Perl and Awk we use notation for dictionaries that in C and other languages is used for arrays. Thus

`D[x] = y` inserts object  $(x, y)$  with key  $x$

`delete D[x]` deletes this object

`if (x in D)` tests if object with key  $x$  is present in  $D$

`for (x in D)` iterates through all keys of objects in  $D$   
(`ListAll` operation)

Because we do not declare the size of  $D$ , these are resizable dictionaries. This introduces a big convenience to the programmer, as well as a certain factor (not too large) of slowdown and larger memory than necessary.

## Implementation with chaining

It is described pretty well on page 266 in the textbook. Load factor close to one seems optimal: we do not slow down `ListAll` which has to inspect the entire array of chains (singly linked lists) and we have very fast other operations that require a computation of the hash function and the scan of a list that has length, on the average, "load factor+present", where "present" is 1 if key  $x$  is present in the dictionary, 0 otherwise.

The speed of computing the hash function is one consideration, the other: if we have particular key  $x$ , is its chain really as short as the average chain?

The second concern is rather tricky. "On the average" in the running time means the average that does not depend on the data, or the set stored in the dictionary, but on the random choice of parameters used to compute the hash function.

One formula proven to be good assumes that we use some prime number  $m$  as the number of chains, that a key  $x$  is a sequence of bytes  $(b_1^x, \dots, b_k^x)$  and that we randomly select integers  $(c_1, \dots, c_k)$  from the range 0 to  $m - 1$ ,

$$\text{Then } h(x) = \left( \sum_{i=1}^k c_i b_i^x \right) \bmod m.$$

This formula works well if  $m$  is larger than the value of bytes, so it is not practical to use  $m$  lower than 257. This is not a problem unless we want to use a very large number of very small dictionaries. In that case, we can merge all of them to one dictionary: rather than key  $x$  in dictionary  $D[i]$  we would use key  $(x, i)$  in `BIG_D`.

## Resizing

When the dictionary is very small, we can tolerate sub-optimally small load. Otherwise, when the load factor grows, say, to 4, we replace the array of chains with another, so the load factor drops to 1. This involves an overhead on the insertions that increased the size of the set from  $m$  to  $4m$ , we had  $3m$  such insertions, and we must follow them with  $4m$  — to insert  $4m$  objects again, to the larger table. Thus we have  $7/3$  insertions when we had 1.

We can do something similar when the load factor drops to, say, 0.25. Then we have overhead on  $0.75m$  deletions that decreased the load factor from 1 to 0.25: for three deletions we need one insertion.

## Application: nearest neighbor problem

We have an array of points  $P[n]$  and a distance measure, say, square of Euclidean distance,

$$\text{dist}(i, j) = (P[i].x - P[j].x)^2 + (P[i].y - P[j].y)^2$$

Our task is to find  $i, j$  such that  $\text{dist}(i, j)$  is minimal.

Think: why we would prefer to minimize the square of Euclidean distance rather than the Euclidean distance?

Stage one of the method — sampling.

Set probability  $p = n^{-0.75}$ . For each point in  $P[n]$ , with probability  $p$  select  $P[i]$  to `Sample[pn]`. Solve the problem in `Sample`. Let  $L$  be the resulting Euclidean distance.

Stage two of the method — make a grid

For each point in  $P[n]$  compute its grid square,

$$P[i] \rightarrow \left( \lfloor P[i].x/L \rfloor, \lfloor P[i].y/L \rfloor \right)$$

Create a dictionary of occupied grid squares, and for each such square, make a list of points that occupy it.

Stage three of the method — distance checks

For each occupied grid square  $(a, b)$

check all distances between the points in  $(a, b)$

for vectors  $V = (-1,1), (1,0), (0,1), (1,1)$

if square  $(a, b) + V$  occupied,

check all distances between the points in squares  $(a, b)$  and  $(a, b) + V$

We skip distance checks only between points that are not in adjacent grid squares, hence in distance larger than  $L$ . A probabilistic analysis shows that we perform, on the average,  $O(n)$  distance checks. The use of a dictionary of non-empty grid squares allows to efficiently find all pairs of points in adjacent grid squares. distance checks.