1 Hashing

Hashing is covered by undergraduate courses like "Algo I". However, there is much more to say on this topic. Here, we focus on two selected topics: *perfect hashing* and *cockoo hashing*. In general, the task of hashing is to store a set S of n *elements* where each element $x \in S$ has a unique *key*, $k(e) \in U = \{0, \dots u-1\}$, subject to the following operations

- insert(x): insert a new element x
- delete(k): remove the element x with $k(x) = k$
- search(k): return (a pointer to) the element x with $k(x) = k$

Observe that the third operation is essentially the *access* operator in arrays, with U being the set of array indices. For this reason, data structures for hashing are often called "associative arrays". (Other names are „hash tables", „dictionaries" and many more).

Indeed, if $n = \Theta(u)$, a normal array of size u will solve the problem optimally. Hence, we focus on the case $n = o(u)$ in the following, and we only want to use $O(n)$ space.

Baseline Algorithms

	linked list	balanced seach tree	sorted resizable array
insert	$\Theta(1)$	$O(\log n)$	$O(n)$
delete	$O(n)$	$O(\log n)$	$O(n)$
search	$O(n)$	$O(\log n)$	$O(\log n)$

We already know that there are much better solutions, e.g. hash tables with chaining (or linear probing):

	hashing with chaining	
insert	$O(1)$	amortised (resize)
delete	$O(1)$	expected (collision), amortised (resize)
search	$O(1)$	expected (collision)
space	$O(n)$	worst case

1.1 Perfect Hashing

We can do better if the keys are *static* (known in advance), namely achieve $O(1)$ *worst case* search time.

	perfect hashing	
search	$\Theta(1)$	*worst case*
construction	$O(n)$	expected
space	$O(n)$	expected

Recomended Reading

- M.E. Fredman, J.Komlós, E. Szemerédi: Storing a Sparse Table with $O(1)$ Worst Case Access Time. J. ACM 31(3):538-544 (1984)
- T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein: Introduction to Algorithms (3rd ed.). MIT Press, 2009. Chapter 11.5
- K. Mehlhorn, P.Sanders: Algorithms and Data Structures: The Basic Toolbox. Springer, 2008. Chapter 4.5

We begin with the observation that if the hash table is large enough, then even a *single* collision is unlikely to occur:

Lemma. *If we store n keys in a hash table of size $m = n^2$ with a universal ("truly random") hash function, the probability of a single collision is at most $1/2$.*

Proof. By the properties of universal hashing, the probability that a pair (x, y) collides is $1/m$:

$$P[h(x) = h(y)] = 1/n^2 \text{ for } x \neq y \ ,$$

if h denotes the hash function chose at random from a universal class of hash functions. In total, these are $\binom{n}{2}$ pairs. Hence, the expected number of collisions is

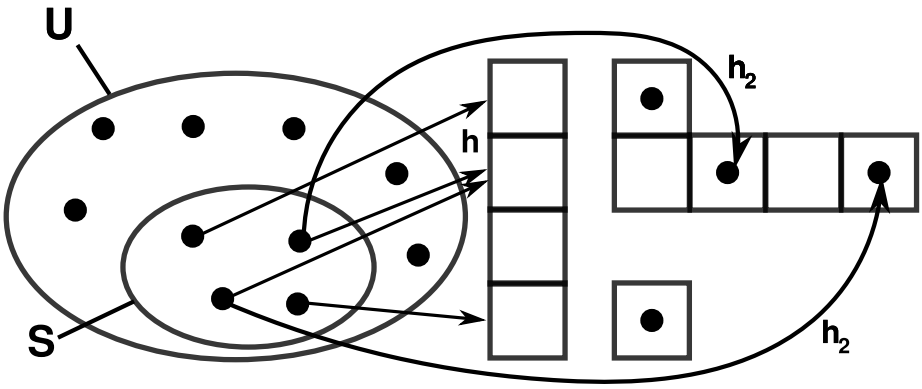
$$\begin{aligned} E[\#\text{collision}] &= \binom{n}{2}\frac{1}{n^2} \\ &= \frac{n^2-n}{2}\cdot\frac{1}{n^2} \\ &= \frac{1}{2}-\frac{1}{2n} \\ &< \frac{1}{2} \ . \end{aligned}$$

Now *Markov's inequality* says $P[X \geq t] \leq E[X]/t$ for any random variable X and any t. Applying this with $X = \#\text{collisions}$ and $t = 1$, we get

$$P[\#\text{collision} \geq 1] \leq \frac{E[\#\text{collision}]}{1} < \frac{1}{2} \ .$$

□

However, $O(n^2)$ space is prohibitive. To bring this space down to linear, we adopt a two-level approach by first hashing into a table of size $O(n)$ as in hashing with chaining, and then, instead of chaining, using the quadratic approach from above for each first level bucket separately.



We only have to make sure that the sum of the sizes of the secondary tables is not too big. For this, we let n_i denote the number of items that get hashed to the i-th bucket with the primary hash function ($n_i = |\{x \mid h(x) = i\}|$). We can then show the following

Lemma. *With universal hashing and a hash table of size $m = n$,*

$$E\left[\sum_{i=1}^{m} n_i^2\right] \leq 2n$$

.

Proof. First note

$$\begin{aligned}\sum n_i^2 &= \sum (2\binom{n_i}{2} + n_i) \\ &= 2\sum \binom{n_i}{2} + \sum n_i \\ &= 2\sum \binom{n_i}{2} + n \ .\end{aligned}$$

Now

$$E\left[\sum n_i^2\right] = E\left[2\sum \binom{n_i}{2} + n\right] = n + 2E\left[\sum \binom{n_i}{2}\right] \ .$$

But $\sum \binom{n_i}{2}$ is the *total* number of pairs that collide (as $\binom{n_i}{2}$ is the number of pairs colliding in bucket i). And because $P[h(x) = h(y)] = 1/m$ for $x \neq y$ (using universal hashing), we have

$$\begin{aligned}E\left[\sum \binom{n_i}{2}\right] &= \underbrace{\binom{n}{2}}_{\text{all pairs}} \cdot \underbrace{\frac{1}{m}}_{\text{prob. of collision}} \\ &= \frac{n-1}{2} \ ,\end{aligned}$$

so

$$\begin{aligned} E\left[\sum n_i^2\right] &= n + 2\frac{n-1}{2} \\ &= 2n - 1 \\ &< 2n \end{aligned}$$

□

With Markov's inequatlity (on $X = \sum n_i^2$ and $t = 4n$), we can show that $P\left[\sum n_i^2 \geq 4n\right] \leq 1/2$. So, on average we need less than 2 trials for the primary hash functions to find one such that the combined *sizes* of the secondary hash functions is at most $4n$. Also, for any secondary hash function we need on average two trials such that they are collision free, so the *construction time* is linear in expectation.

1.2 Cuckoo Hashing

	Cuckoo hashing	
insert	$O(1)$	expected (rehash), amortized (rebuild)
delete	$O(1)$	expected(rehash), amortized (rebuild)
search	$O(1)$	worst case
space	$O(n)$	worst case

Cuckoo Hashing should be directly compared with hashing with chaining: for search, we get worst case instead of expected, amortized times. (It can be shown that the expected worst case time for hashing with chaining is $\Theta(\log n / \log \log n)$ or, in the presence of two hash tables, $\Theta(\log \log n)$. Similar worst case bounds apply for linear probing.)

Recomended Reading

- R. Pagh, F.F. Rodler: Cuckoo Hashing. J. Algorithms 51(2): 122-144 (2004)
- R. Pagh: Cuckoo Hashing for Undergraduates. Lecture note IT University of Copenhagen. Available online at `http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf` (2006)
- A. Schulz: Kuckucks-Hashing. Lecture note, Universität Münster, WS10/11. Available at `http://wwwmath.uni-muenster.de/logik/Personen/Schulz/WS10/VL16.pdf`

The idea of cuckoo hashing is to use two hash tables T_1 and T_2 of size $m = 2n$ each, and two hash functions h_1 and h_2, such that any key x is stored at either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. Hence, the search procedure is particularly easy:

```
function search(x):
if(T_1[h_1(x)] = x or T_2[h_2(x)] = x )return true;
```

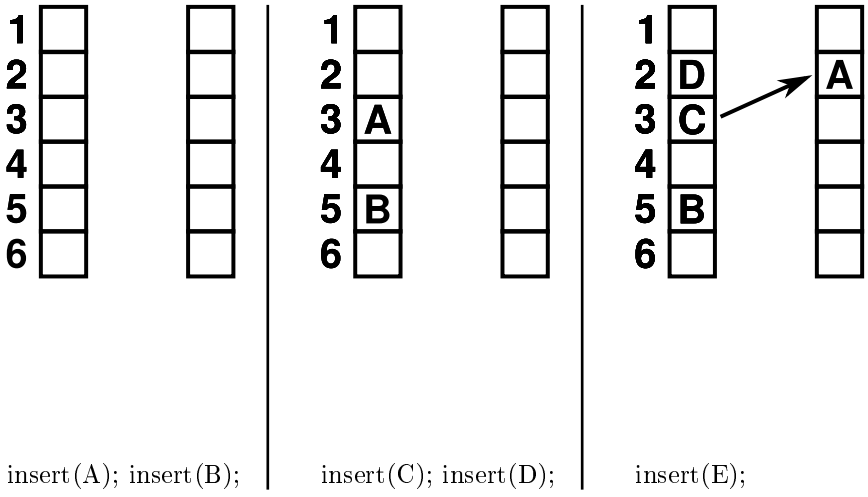
Also, the deletion of keys is as simple as the search (apart from rebuilding the table when n becomes too small - we will consider this at the end of this lecture).

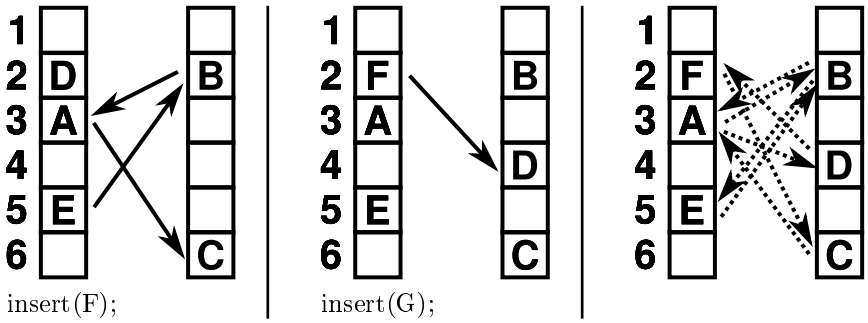
The real "problems" arise when trying to insert a new element -– it could be that both possible positions are already occupied! We first give the insertion procedure and then analyse its performance.

```
procedure insert(x):
if search(x) then return;
k := 1;
repeat maxLoop times:
    swap x with T_k[h_k(x)];
    if(x = ⊥):
        n++;
        if(n > m/2):
            rehash(2m);
        return;
    k := 3 − k;
rehash(m);
insert(x);
```

Example.

$x \in S$	$h_1(x)$	$h_2(x)$
A	3	2
B	5	2
C	3	6
D	2	4
E	5	2
F	2	6
G	3	4





The algorithm can not insert G because the elements are shifted in an endless loop. Eventually *maxLoop* iterations are performed and the table is rehashed.

In the analysis below, we will show that the probability of making maxLoop iterations is low (similar to hashing with chaining, where one shows that the probability of large buckets is low).

We need the concept of (c, k)-universal hash functions.

Definition. A family $\{h_i\}_{i \in I}$, $h_i : U \to \{0, \ldots, m-1\}$is called (c, k)-universal if for any k distinct elements $x_1, \ldots, x_k \in U$ and all $y_1, \ldots, y_k \in \{0, \ldots, m-1\}$

$$P\left[h_i(x_1) = y_1, \ldots, h_i(x_k) = y_k\right] \leq c/m^k$$

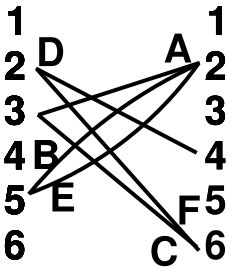
for a uniform random $i \in I$.

(Observe that $(1, 1)$-universal classes are exactly those we need for "standard" universal hashing.) In words, the above definition says that for any k distinct elements $x_1, \ldots, x_k$, their hash codes $h_i(x_1), \ldots, h_i(x_k)$ are independent random variables, and the hash code for any fixed x_i is uniformly distributed in $\{0, \ldots, m-1\}$.

For cuckoo hashing, we need two $(1, O(\log n))$-universal hash functions. (By a result of Siegel [FOCS'89, pp. 20-25]), such functions exist, can be computed in $O(1)$ time, and need $O(\log n)$ space.)

We introduce the cuckoo graph: it is a bipartite graph where the vertices correspond to the $2m$ table entries of T_1 and T_2, and there is an edge between $T_1[i]$ and $T_2[j]$ iff $\exists x \in S : h_1(x) = i$ and $h_2(x) = j$ (hence n edges in total).

Example. Continuing the previous example, after the instertion of F the cuckoo graph looks as follows:



Now observe that a sequence of consecutive *replacements* in the insertion algorithm corresponds to a *walk* in the cuckoo graph. We want to analyze the probability that an insertion fails, i.e., that there are more than maxLoop replacements. The walk $h_1(x_1), \ldots, h_{1/2}(x_t)$ may contain up to two distinct cycles. We make a case distinction on the number of cycles. Let us fix maxLoop$= 6 \log n$.

1. The walk contains no cycle so a rehash is performed if $t >$maxLoop. For a walk to have the length t all the cells on the walk have to be occupied. The probability that some cell x is occupied is

$$\begin{aligned} P\left[T_1\left[h_1(x)\right] \text{ occupied}\right] &\leq \sum_{\substack{y \in S \\ x \neq y}} P\left[h_1(y) = h_1(x)\right] \\ &\leq \frac{(n-1)}{m} \\ &\leq \frac{1}{2} \ . \end{aligned}$$

   and the probability that two cells are occupied is

$$\begin{aligned} & P\left[T_1\left[h_1(x_1)\right] \text{ occupied and } T_1\left[h_2(x_2)\right] \text{ occupied}\right] \\ \leq & \sum_{y_1, y_2 \in S} P\left[h_1(y_1) = h_1(x_1), h_2(y_2) = h_2(x_2)\right] \\ = & \sum_{y_1, y_2 \in S} P\left[h_1(y_1) = h_1(x_1)\right] \cdot P\left[h_1(y_2) = h_1(x_1)\right] \\ \leq & \frac{n^2}{m^2} \\ = & \frac{1}{4} \end{aligned}$$

   and so on and so forth. In general,

$$P\left[t \text{ cells occupied}\right] \leq 1/2^t = 1/n^6 \ ,$$

   so a rehash occurs with probability $\leq 1/n^6 \leq 1/n^2$. Assuming that no rehash occurs, the expected running time in this case is therefore

$$\begin{aligned} E\left[\#\text{cells occupied}\right] &\leq \sum_{t=1}^{6 \log n} t \cdot 2^{-t} \\ &\leq \sum_{t=1}^{\infty} t \cdot 2^{-t} \\ &= 2 = O(1) \ . \end{aligned}$$

2. The walk contains exactly one cycle. A rehash is performed if $t >$maxLoop. Look at the three parts of the walk and their length:

(a) from $h_1(x_1)$ to $h_2(x_j)$, $l_1 = j$

(b) from $h_1(x_{j+1})$ to $h_2(x_{i+j-1})$, $l_2 = i - 1 < l_1$

(c) from $h_2(x_{i+j})$ zo $h_1(x_t)$, $l_3 = t - (i + j) + 1$

If $l_1 < t/3$ then $l_1 + l_2 < 2t/3$ and hence $l_3 > t/3$. If $l_3 < t/3$ then $l_1 + l_2 < 2t/3$ and hence $l_1 > t/3$ so at least one of the parts 1 or 3 are of length $> t/3$, and both parts consist of distinct elements from S. Hence, by the analysis of first case we get

$$\begin{aligned} P\,[\text{rehash}] &< P\left[\frac{t}{3} \text{ different cells occupied}\right] \\ &\leq \frac{1}{2^{t/3}} = \frac{1}{n^2} \ . \end{aligned}$$

Also, the expected running time (assuming that no rehash occurs) is again constant.