COMP3557
Design of Algorithms and Data Structures
Part 1: Cuckoo Hashing

Tom Friedetzky
MCS 2004
tom.friedetzky@durham.ac.uk

2022/23

Introduction

About **dictionary data structures**, used for storing a set of items

Support three basic operations:

- Lookup(x): returns true if x is in current set
- Insert(x): adds item x to current set if not already present
- Delete(x): removes x from current set if present

Trivial solution: **linked list**; drawback: worst-case (and even average) linear time for every operation (assuming insertions first check whether element is already in list)

A bit better: **balanced search trees** (red/black, AVL, splay, ...)

Here: **hashing-based**

First simple **chaining**, then **Cuckoo hashing**

Hashing (here!)

Probabilistic data structures

Performance bounds will often not hold in worst case but in **expected case** – average over all random choices (algorithms will make random choices) – worst case can be properly bad

Often can show that true "**with high probability**", that is, extremely unlikely to deviate much from expected values

Behaviour determined by one or two **hash functions**: take items as input, return "random" values in some set $\{1, \ldots, r\}$.

Space usage will be bounded in terms of n (will in fact be $O(n)$, modest constant-factor overhead)

Balanced search trees usually logarithmic time bounds on operations, we will achieve **constant expected** per operation

Assumptions

1. All items to be stored have same size, and we can compare any two items in constant time
2. Have access to hash functions h_1 and h_2 such that any function value $h_i(x)$ is equal to a particular value in $\{1, \ldots, r\}$ with probability $1/r$

   (Only possible if hash functions chosen in random fashion – will discuss this later)

   Function values are probabilistically independent of each other (one function value says nothing about other ones)

   Hash functions values can be computed in constant time
3. There is fixed upper bound n on number of items in the set

Main idea of hashing-based dictionaries is to let hash function(s) decide where to store items

Item x will be stored at "position $h_1(x)$" in an array of size $r \geq n$

For this, h_1 must really be a **function**, that is, $h_1(x)$ must be fixed value

Collisions

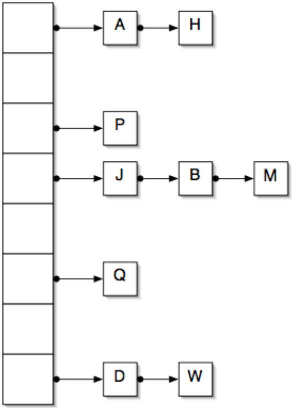
Likely to get **collisions**: items $x \neq y$ with $h_1(x) = h_1(y)$

For each value $a \in \{1, \ldots, r\}$ there is some **set** S_a of items having this particular hash value of h_1

Obvious idea: make **pointer** from position a to some **data structure** holding set S_a; often called a **bucket**

Perhaps surprisingly, very simple **linked list** sufficient (say, doubly-linked for convenience)

Example



Items J, B and M have same value of h_1 and have been placed in the same "bucket", a linked list of length 3, starting at position $h_1(J)$ in the array

Analysing hashing with chaining

Starting with two observations:

Observations

1. For any two distinct items x and y, the probability that x hashes to the bucket of y is $O(1/r)$ (from assumptions on h_1)

   ... **why?**

2. The time for an operation on an item x is bounded by some constant times the number of items in the bucket of x

Let's analyse an operation on item x

By second observation, can bound time by bounding expected size of x's bucket

For any operation might be case that x is stored in data structure when operation begins, but this can cost only constant-factor time extra, compared to case where x is not in list

May therefore assume that bucket of x contains only items different from x

Claim

The expected time for any operation is constant.

Proof

- Let S be set of items that were present at beginning of operation (all of them, not just in x's bucket)
- For any $y \in S$, first observation says that prob that operation spends time on y is $O(1/r)$
- Therefore, expected ("average") time consumption charged to y is $O(1/r)$
- To get total expected time, must sum up expected time for all elements in S
- By linearity of expectation, this is $|S| \cdot O(1/r)$, which is $O(1)$ as r was chosen such that $r \geq n \geq |S|$

Well, that was not too bad.

What however if we absolutely **must** guarantee **worst-case constant lookups**, rather than only expected?

One approach is to make the table huge.

Another is to use **hash functions with no collisions**

- called **perfect hash functions**; would allow us to insert items directly into array, rather than having to use lists
- can be made to work, but is rather complicated, especially for insertions

Will consider simpler way, first described in

R. Pagh and F. Rodler
Cuckoo Hashing
Proceedings of European Symposium on Algorithms, 2001

and subsequently refined many times, by many people

Idea

Instead of requiring x be stored at position $h_1(x)$, we give **two alternatives**: $h_1(x)$ and $h_2(x)$

We allow at most one element to be stored at any position:
no need for a data structure holding colliding items

Allows us to **look up** an item by inspecting just two positions in the array!

When **inserting** a new element x may of course still happen that there is no space since both positions $h_1(x)$ and $h_2(x)$ are occupied

Pull a cuckoo: throw out current occupant y of position $a = h_1(x)$ to make room, and place x there instead

This in turn leaves y homeless:

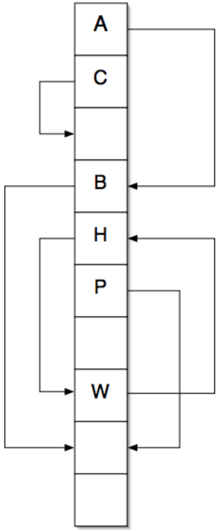
- if alternative position for y (that is, if $h_1(y) = a$ then $h_2(y)$, and vice versa) is vacant, move it **there**
- otherwise, y repeats x's trick and displaces **that** positions current inhabitant

Continue until the procedure finds a vacant position, or has taken too long

In latter case, choose two new hash functions, throw everything in the air, and start from scratch (mustn't happen too often!)

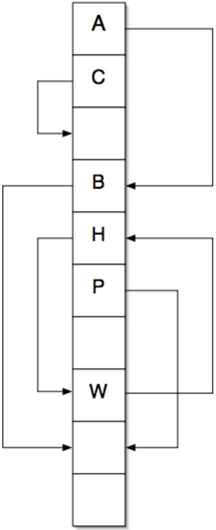
Example

- arrows show alternative position of each element
- if want to insert new item in position of A, A would be moved to its alternative position, currently occupied by B, which would move B to its alternative position, which is currently vacant
- if want to insert new item in position of H, would be unsuccessful: H is part of cycle (together with W)



Discussion

- can notice similarities with chaining, only now chains are in the array itself!
- when inserting item in position A, procedure will traverse path (chain) involving positions of A and B
- indeed, can see that when inserting new element x, procedure will only visit positions to which there is a path in the "cuckoo graph" from either $h_1(x)$ or $h_2(x)$
- let's call this path the bucket of x
- may have more complicated structure than in chaining: cuckoo graph can be cyclic
- need to rehash only when graph cyclic



Summary operations

If everything works as advertised, then

- look-ups: $O(1)$
- deletions: $O(1)$
- insertions: we'll see

Analysis

Insertion procedure can loop n times **only** if there is cycle in graph

Every insertion will succeed so long as there is no cycle

Time spent will be bounded by a constant times size of "bucket", so Obs 2 from analysis of chaining still holds

Recall: Obs 2

The time for an operation on an item x is bounded by some constant times the number of items in the bucket of x

Will now show that **expected insertion time** (in absence of rehash) is **constant** (rehash probability separate)

For this, will show that Obs 1 holds as well

Recall: Obs 1

For any two distinct items x and y, the probability that x hashes to the bucket of y is $O(1/r)$

In proof, will consider undirected cuckoo graph (no orientation on edges)

Analysis

Note: y can be in x's bucket **only** if there is a path between one of x's possible positions and position of y

Lemma

For any positions i and j, and any $c > 1$, if table size $r \geq 2cn$ then the probability that in the undirected cuckoo graph there exists a path from i to j of length $\ell \geq 1$, which is a shortest path from i to j, is at most $\mathbf{c}^{-\ell}/\mathbf{r}$.

- c can be any constant > 1, maybe think $c = 2$
- Lemma says that if number r of nodes (size of array) is large enough compared to number n of edges then we have low probability that any two nodes i and j are connected by a path
  - prob that they are connected by a path of constant length is $O(1/r)$
  - prob that a path of length ℓ exists (but no shorter path) is exponentially decreasing in ℓ

Proof of Lemma

Induction on ℓ

Base case $\ell = 1$ (just an edge):

- there is set S of at most n items that have i **and** j as possible positions
  - we're only interested in paths of length 1 here
  - those are edges, and an edge represents an item's the two hash values
- for each item, prob that this is true is at most $2/r^2$
  - first goes here and second there (each with prob $1/r$), or other way around
- overall prob bounded by $\sum_{x \in S} 2/r^2 \le 2n/r^2 \le c^{-1}/r$ ✓

  (as $r \ge 2cn \Rightarrow 2n/r \le c^{-1} \Rightarrow 2n/r^2 \le c^{-1}/r$)

Proof of Lemma

Inductive step

- need to bound prob that there **exists** a path of length $\ell > 1$, but **no shorter path**
- this is only if, for some position k,
  - there is a shortest **path** of length $\ell - 1$ from i to k that does not go through j, and
  - there is an **edge** from k to j
- by induction hypothesis, prob for former is bounded by $c^{-(\ell-1)}/r = c^{-\ell+1}/r$
- argument for latter is exactly the same as in base case (one edge between k and j), so that's c^{-1}/r
- prob that **both** conditions hold for **particular** choice of k is no more than $\frac{c^{-\ell+1}}{r} \cdot \frac{c^{-1}}{r} = \frac{c^{-\ell}}{r^2}$
- prob that there **exists** a k with both conditions satisfied is $r \cdot \frac{c^{-\ell}}{r^2} = \frac{c^{-\ell}}{r}$ ✓

Claim

Observation 1:

For any two distinct items x and y, the probability that x hashes to the bucket of y is $O(1/r)$.

still holds.

Proof

- if x and y are in same bucket, then there is path of some length ℓ between some $z_x \in \{h_1(x), h_2(x)\}$ and some $z_y \in \{h_1(y), h_2(y)\}$
- by Lemma, this happens with prob at most

$$4 \cdot \sum_{\ell=1}^{\infty} \frac{c^{-\ell}}{r} = \frac{4}{r(c-1)} = O(1/r) \qquad \checkmark$$

Rehashing

Q: But isn't rehashing very expensive, and destroys everything?
A: It's expensive alright, but it's not going to happen very often

Simple argument (can be made much more precise and tight):

- Can prove (using Lemma) that expected number of rehashes during insertion of $\Theta(n)$ elements is $O(1)$ ($\Longleftarrow$ **your homework**)
- If time for one rehash is $O(n)$, then expected time for all rehashes is $O(n)$, which is $O(1)$ per insertion
- $\Rightarrow$ **amortised** cost of rehashing is constant

Problem

When doing analysis carefully, can show that with arbitrary constant $\epsilon > 0$ and $2(1+\epsilon)n$ slots, for at most n keys, get

- look-ups $O(1)$ worst-case
- deletions $O(1)$ worst-case
- insertions $O(1)$ expected, amortised

But: Insertion of a set of n keys fails and triggers rebuild of the whole data structure with probability $O(1/n)$. Doesn't sound like much, but...

In some applications, e.g.,

- high-performance routing (packet statistics)
- database indexing

a failure probability of $O(1/n^3)$ could already lead to a failure rate that is too high.

$\Rightarrow$ Cuckoo hashing not applicable, although its performance is suitable for such applications.

Task: Preserve the performance and lower the failure probability.

Kirsch, Mitzenmacher and Wieder:

- add a small constant-sized piece of memory, the so-called **stash**
- move elements that cannot be inserted to this stash

They prove: Using a stash of size s lowers failure probability from $O(1/n)$ to $O(1/n^{s+1})$.

Proof is technically involved ("Poissonization", "Markov Chain coupling"). Assumes fully random hash functions.

Idea: detect when looping, and break up cycles (remove one edge and stash).