Advanced Data Structures
Lecture 4

Lecture by Shay Mozes
Typeset by Steven Karas

2016-12-01
Last edited 18:12:26 2016-12-28

1 Hashing

Chained hashing analysis First, assume the cost of inserting x elements is the length of the chain for x. Second, for all pairs of keys x, y, the chances of x and y are in the same chain is $\leq \frac{1}{m}$.

$$E[T(Insert(x))] = E[T(Insert(y))] \leq \sum_{y \in S} Pr[y \text{ in same chain as } x] \leq n \cdot \frac{1}{m} = O(1)$$

FkS perfect static hashing $O(1)$ lookups with no collisions. $O(n)$ space.

Construction $O(n)$ average time, but can be much worse.

Perfect dynamic hashing In contrast to static hashing, here we need to support adding and removing elements from our "universe".

Strong Universal hashing Not covered in this course, but makes stronger guarantees about collision probability.

1.1 Cuckoo hashing

Invented by Pagh, Rodler in 2001. Uses 2 hash functions h_1, h_2. Delete and member both take $O(1)$. Insert takes $O(1)$ takes average amortized time.

Invariant Every key x is either in $T[h_1(x)]$ or $T[h_2(x)]$.

Insertion Insert x into $T[h_1(x)]$. If it's already occupied, move the old value to the other space it can occupy. [1]

Collision Loops If we detect a cycle in the insertion process, we choose new hash functions h_1, h_2 and reinsert all the elements into the new table. Typical implementations will either continue until taking $\log n$ actions, even though a cycle is only pigeonholed after n steps.

[1] Visual example given in video

1.1.1 Analysis

We assume that the size of the universe is smaller than some load factor N. If there are no cycles, we can show the average complexity is $O(1)$.

If we go over N elements, we can rebuild the table, and keep our amortized complexity

Rebuild complexity The average complexity of rebuilding the hash functions/table over a sequence of N elements is $O(N)$. The amortized time is $O(1)$.

Complexity We want to leverage our proof of chained hashing. Define a Cuckoo Graph be the graph of the hash table, where the edges are the elements in the table, with the endpoints of $T[h_1(x)]$ and $T[h_2(x)]$. The insertion of an element $x \in S$ takes no longer than the size of the connected subgraph that includes $T[h_1(x)]$.

We say that x and y are in the same "cuckoo chain" if there is a path from $\{h_1(x), h_2(x)\}$ to $\{h_1(y), h_2(y)\}$.

We will show that the probability that there is a path from x to y is $O(\frac{1}{m})$.

Lemma For any pair of vertices i, j and any constant c; if $m \geq 2cn$, then the probability that there is a path from i to j with length smaller than l is less or equal to $\frac{1}{m \cdot c^l}$. Let this probability be $Pr(\exists P(i,j))$.

Proof of lemma By induction on l.

$$Pr(\exists P(i,j)) \leq \sum_{x \in S} Pr(h_1(x) = i \wedge h_2(x) = j) + Pr(h_1(x) = j \wedge h_2(x) = i)$$

$$Pr(\exists P(i,j)) \leq 2n \cdot \frac{1}{m^2} \leq \frac{2n}{2cn} \cdot \frac{1}{m} = \frac{1}{mc}$$

There exists a path of length $\leq l$ from i to j if and only if

$$\sum_k Pr(E_1^k \wedge E_2^k) = \sum_k Pr(E_2^k | E_1^k) \cdot Pr(E_1^k)$$
$$\leq m \cdot \frac{1}{mc} \leq \frac{1}{mc^{l-1}}$$

$$Pr(x, y \text{ are in the same chain}) \leq 4 \cdot Pr(\exists P(i,j)) = \frac{4}{m} \sum_{l=1}^{\infty} \frac{1}{c^l} = O(\frac{1}{m})$$

Cycle Complexity The probability of having a cycle is actually really small. If the table is empty or sparse, the chances are even less, so we'll just analyze the case where the table is half full, and we're filling it up. Let $c = 3$.

$$Pr[\exists C(i,j)] \leq \sum_k \sum_{l=1}^{\infty} [\exists P(k,k) \geq l]$$

$$Pr[\exists C(i,j)] \leq m \sum_{l=1}^{\infty} \frac{1}{mc^l} = \frac{1}{c-1} \leq \frac{1}{2}$$

On average, the number of times we need to choose new hash functions is at most 2. Rebuilding the table takes $O(n)$ average time.

Implementation details Fully independent random hash functions are difficult/impossible, so it's sufficient to use $\log n$ independent hash functions. The load factor is typically around 1/6th. Because empty cells are only a pointer, this can be relatively OK, unless you have a ton of elements.

2 Predecessor/Successor Integer Problem

Given a set S of n integers from some universe $U = 1, ..., u - 1$, where $n << u$

Operations

- member
- insert
- delete
- succ
- pred

Various Implementations

f	Data structure	Time	Space
1962	Binary Trees	$O(\log n)$	$O(n)$
1975	van Emde Boas trees	$O(\log \log n)$	$O(u)$
1983	y-fast trie	$O(\log \log n)$	$O(n)$
1993	fusion trees	$O(\frac{\log n}{\log \log u})$[2]	$O(n)$

Word RAM model Words of size w, which can store a pointer or a value. $w > \log u$. An instruction on a word or pair of words takes $O(1)$. Following a pointer takes $O(1)$.

This is called cache-oblivious. Reality is quite different

2.1 van Emde Boas Trees

The intuition is that we do a binary search on the representation, which gives us $O(\log \log n)$.

Basically, store 0 or 1 in a giant array. Then build a tree on top of it, where each node is 1 if any of the children are 1, and 0 is all of them are 0. Note that the depth of this tree is $O(\log u)$.

Recursive trees

[2]Note that if $\log \log u = \omega(\sqrt{\log n})$, then we can sort integers in $O(n\sqrt{n})$