CS 591, Lecture 6
Data Analytics: Theory and Applications
Boston University

Babis Tsourakakis

February 8th, 2017

Universal hash family

Notation: Universe $U = \{0, \ldots, u-1\}$, index space $M = \{0, \ldots, m-1\}$, n size of set S.

[Carter and Wegman, 1979]

- A family $\mathcal{H}$ of hash functions is **2-universal** if for any $x_1 \neq x_2$,

$$\mathbf{Pr}\left[h(x) = h(y)\right] \leq \frac{1}{m}.$$

  for a uniform $h \in \mathcal{H}$.

- A family $\mathcal{H}$ of hash functions is **strongly 2-universal** if for any $x_1 \neq x_2$,

$$\mathbf{Pr}\left[h(x_1) = y_1, h(x_2) = y_2\right] = \frac{1}{m^2}.$$

  for a uniform $h \in \mathcal{H}$.

Does it remind you of anything from previous lectures?

Universal Hashing

Reminder from Lectures 4,5:

- We defined the notion of k-wise independent family of hash functions
- When we say k-universal, we usually mean strongly k-universal.
- We discussed how one can construct a 2-wise independent family

$$h(x) = ax + b \mod p.$$

[Carter and Wegman, 1979]

Avoiding Modular Arithmetic

- Modular arithmetic can be slow
- [Dietzfelbinger et al., 1997] proposed the following hash function (collisions twice as likely):
- For each k, l they define a class $\mathcal{H}_{k,l}$ of hash functions from $U = [2^k]$ to $M = [2^l]$

$$\mathcal{H}_{k,l} = \{h_\alpha | h_\alpha = (ax \bmod 2^k) \text{ div } 2^{k-l}.$$

- **Claim:** If α is a random odd $0 < \alpha < 2^l$, and $x_1 \neq x_2$, then

$$\mathbf{Pr}\left[h(x) = h(y)\right] \leq 2^{-l+1}.$$

Perfect Hashing

- So far, we've seen that the average case behavior of hashing is significantly superior to the worst case.
- However, we can get excellent worst case performance if the set of keys is static.
- Perfect hashing requires $O(1)$ memory accesses in the worst case.

Theorem: If $\mathcal{H}$ is 2-universal, $|S| = n$, $m \geq \alpha\binom{n}{2}$, then

$$\mathbf{Pr}\left[\text{h is perfect for } S\right] \geq 1 - \frac{1}{\alpha}.$$

Perfect Hashing

Proof sketch:

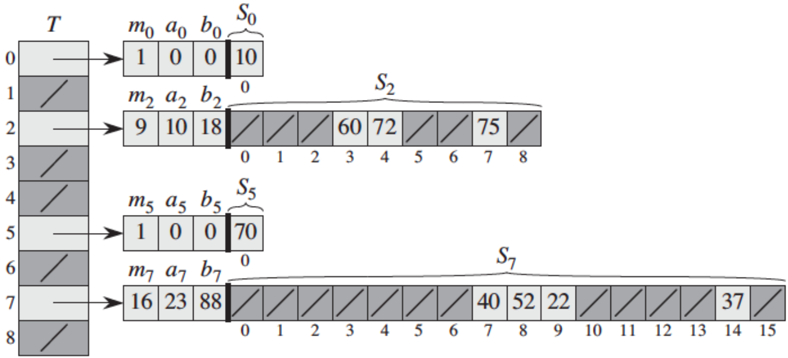
- Define $X = \#$ collisions, and let's compute $\mathbb{E}[X]$
- $X = \sum_{i \neq j} X_{ij}$
- $\mathbf{Pr}[X_{ij} = 1] = \frac{1}{m}$
- By linearity of expectation $\mathbb{E}[X] = \frac{\binom{n}{2}}{m} \leq \frac{1}{\alpha}$
- Apply Markov's inequality

$$1 - \mathbf{Pr}[X = 0] = \mathbf{Pr}[X \geq 1] \leq \mathbb{E}[X] \leq \frac{1}{\alpha}.$$

Perfect Hashing

- Issue: $O(n^2)$ space
- Question: Can we get away with $O(n)$ space?
- Yes: Fredman-Komlós-Szemerédi [Fredman et al., 1984].
- Idea: Two level hashing,
  1. Hash using a universal hash function to $n = |S|$ bins.
  2. Rehash perfectly within each bin at second level.

Perfect Hashing



Source: CLRS book

Perfect Hashing

Claim:

$$\mathbb{E}\left[\sum_{j=0}^{n-1} n_j^2\right] \leq 2n.$$

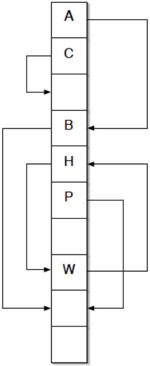
Proof: We count the number of ordered pairs that collide.

$$\mathbb{E}\left[\sum_{j=0}^{n-1} n_j^2\right] = \mathbb{E}\left[\sum_i \sum_j X_{ij}\right] = n + \sum_i \sum_{j \neq i} \mathbb{E}\left[X_{ij}\right]$$
$$\leq n + \frac{n(n-1)}{m} < 2n.$$

Cuckoo Hashing

- An alternative to perfect hashing, that also allows dynamic updates
- Introduced by [Pagh and Rodler, 2001]
- Further extensions, e.g., Hopscotch hashing

Cuckoo Hashing

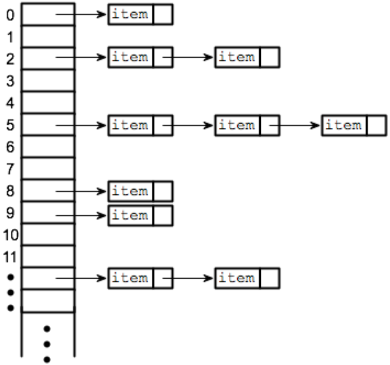


procedure insert(x)
  if $T[h_1(x)] = x$ **or** $T[h_2(x)] = x$ **then return**;
  pos $\leftarrow h_1(x)$;
  loop n **times** {
    if T[pos] $=$ NULL **then** { T[pos] $\leftarrow x$; **return**};
    $x \leftrightarrow T$[pos];
    if pos$= h_1(x)$ **then** pos$\leftarrow h_2(x)$ **else** pos$\leftarrow h_1(x)$;}
  rehash(); insert(x)
end

Figure 2. Cuckoo hashing insertion procedure and illustration. The arrows show the alternative position of each item in the dictionary. A new item would be inserted in the position of A by moving A to its alternative position, currently occupied by B, and moving B to its alternative position which is currently vacant. Insertion of a new item in the position of H would not succeed: Since H is part of a cycle (together with W), the new item would get kicked out again.

Source: "Cuckoo Hashing for Undergraduates" by Rasmus Pagh

Separate Chaining



Source: Hackerearth

Insertion

```
vector <string> Table[20];
int hashTableSize=20;

void insert(string s)
{
// Compute the index using Hash Function
int index = hashFunc(s);
// Insert the element
Table[index].push_back(s);
}
```

Search

```cpp
 void search(string s)
{
  int index = hashFunc(s);
  for(int i = 0;i < Table[index].size();i++)
  {
      if(Table[index][i] == s)
      {
          cout << s << " is found!" << endl;
          return;
      }
  }
  cout << s << " is not found!" << endl;
}
```

Separate Chaining

Load factor α:

$$\boxed{\alpha := \frac{n}{m}.}$$

Claim: Under the assumption of simple uniform hashing, an unsuccessful search takes $O(1+\alpha)$ time.

Proof sketch: $\mathbb{E}[n_j] = \alpha$ for all $j \in \{0, \ldots, m-1\}$.

Why distinguish between unsuccessful and successful searches?

Separate Chaining

Claim: Under the assumption of simple uniform hashing, a successful search takes $O(1+\alpha)$ time.
Proof sketch:

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\Big(1+\sum_{j=i+1}^{n}X_{ij}\Big)\right] &= \frac{1}{n}\sum_{i=1}^{n}\Big(1+\sum_{j=i+1}^{n}\frac{1}{m}\Big) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\alpha/2-\alpha/(2n) \\
&= O(1+\alpha).
\end{aligned}$$

Linear Probing

- Sequential memory accesses are fast
- Values stored directly to hash table
- We hash x to $h(x)$. If this cell is already occupied, then we check $h(x) + 1, h(x) + 1$ and so on (mod arithmetic).
- [Pagh et al., 2007] proved that if hash function is 5-wise independent, then $\mathbb{E}[\text{operation}] = O(1)$.

Insertion

```
// Linear probing
void insert(string s)
{
  int index = hashFunc(s);
  while(Table[index] != "")
     index = (index + 1) % hashTableSize;
  hashTable[index] = s;
}
```

Search

```
void search(string s)
{
 int index = hashFunc(s);
 while(Table[index] != s&&Table[index] != "")
        index = (index+1)%hashTableSize;
 if(Table[index] == s)
      cout << s << " is found!" << endl;
 else
       cout << s << " is not found!" << endl;
}
```

Quadratic Probing

- Difference from linear probing is the choice between successive probes or entry slots

  index = index % hashTableSize

  index = (index + 1^2) % hashTableSize

  index = (index + 2^2) % hashTableSize

  index = (index + 3^2) % hashTableSize

Insertion

```
void insert(string s)
{
  int index = hashFunc(s);
  int h = 1;
  while(hashTable[index] != "") {
     index = (index + h*h) % hashTableSize;
     h++;}
  Table[index] = s;
}
```

Search

```
void search(string s)
{
  int ind = hashFunc(s);
  int h = 1;
  while(Table[ind] != s&&Table[ind] != ""){
     ind = (ind + h*h) % hashTableSize;
     h++;}
  if(Table[index] == s)
    cout << s << "␣is␣found!" << endl;
   else
    cout << s << "␣is␣not␣found!" << endl;
}
```

Double Hashing

- Difference from linear probing is that the interval between probes is computed by using two hash functions.

  indexH = hashFunc2(s);

  index = (index + 1 * indexH) % hashTableSize;

  index = (index + 2 * indexH) % hashTableSize;

Insertion

```
void insert(string s)
{
 int index = hashFunc1(s);
 int indexH = hashFunc2(s);
 while(hashTable[index] != "")
     index = (index+indexH)%hashTableSize;
   hashTable[index] = s;
}
```

Search

```
void search(string s)
{
  int index = hashFunc1(s);
  int indexH = hashFunc2(s);
  while(Table[index]!= s&&Table[index]!= "")
    index = (index + indexH)%hashTableSize;
  if(Table[index] == s)
    cout << s << " is found!" << endl;
   else
    cout << s << " is not found!" << endl;
}
```

references I

Carter, J. L. and Wegman, M. N. (1979).
Universal classes of hash functions.
Journal of computer and system sciences, 18(2):143–154.

Dietzfelbinger, M., Hagerup, T., Katajainen, J., and Penttonen, M. (1997).
A reliable randomized algorithm for the closest-pair problem.
Journal of Algorithms, 25(1):19–51.

Fredman, M. L., Komlós, J., and Szemerédi, E. (1984).
Storing a sparse table with 0 (1) worst case access time.
Journal of the ACM (JACM), 31(3):538–544.

Pagh, A., Pagh, R., and Ruzic, M. (2007).
Linear probing with constant independence.
In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318–327. ACM.

references II

Pagh, R. and Rodler, F. F. (2001).
Cuckoo hashing.
In *European Symposium on Algorithms*, pages 121–133. Springer.