Cuckoo Hashing

Alex Broihier

Outline

Section 1

Hashing Functions and Families

Hash Functions

- h is a hash function if it has the form $h : U \to \{0, 1, \ldots, n-1\}$ for some set U and some constant n
- Example: if U is the integers, $h(x) = x \bmod n$ is a hash function
- Often used to assign an element an index in an array of size n
- This alone is not useful; hash functions are typically used to take an arbitrarily input and give back a seemingly random output

Hash Families

- A hash family is a set of hash functions, $\{h_1, h_2, \ldots\}$
- They provide a source of randomness: you can randomly sample a hash function to use from a hash family
- This will allows us to analyze hash families probabilistically

Universal Hash Families

- A universal hash family is a hash family with the property

$$Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{n} \qquad x \neq y$$

(c, k) Universal Hash Families

- We can generalize universal hash families (and get stronger guarantees while we are at it)
- A hash family is (c, k) universal if for all $x_1, x_2, \ldots, x_k \in U$ and for all $y_1, y_2, \ldots, y_k \in \{0, 1, \ldots, n-1\}$

$$Pr_{h \in H}[h(x_1) = y_1, h(x_2) = y_2, \ldots, h(x_k) = y_k] \leq \frac{c}{n^k}$$

- The previously discussed "standard" universal hash family is $(1, 2)$ universal

Aside: Amortized vs Expected Cost

- Randomness is present in our hashing algorithms, so we need the language to properly describe this. Amortized and expected runtime are different things.
- Amortized runtime is guaranteed to "average out." We analyze the runtime across numerous runs (even though a worst case individual run could be expensive)
  - Example: push back for a dynamically sized array takes amortized $O(1)$ time
- Expected runtime is what will probably happen. The absolute worst case can be very bad (but will very rarely occur)
  - Example: quicksort with randomized pivots takes expected $O(n \log n)$ time

Section 2

Hash Tables and Hashing Strategies

Hash Tables

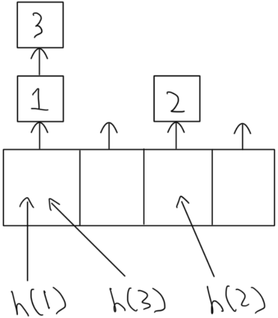
- A hash table uses hashing to implement a "dictionary" data structure, which uses keys to access values
- Fundamental Operations:
  - Add key / value pair
  - Lookup the value for a given key
  - Remove key / value pair

Hash Table Implementation

- Use some hash function h (may be randomly sampled from some hashing family)
- Store an array of size r to hold n elements; we keep $\frac{n}{r} \leq C$, where C is some constant and $\frac{n}{r}$ is the "load factor"
- For each key / value pair, store it at index $h(\text{key}) \bmod n$
- Hidden Operations:
  - Rehash (resample hash function h)
  - Resize (grow or shrink the internal array)
- What happens if two key / value pairs are assigned to the same index in our array?

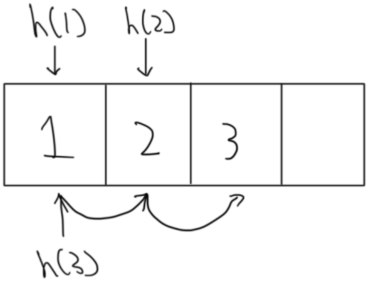
Resolving Hash Table Collisions

- We say a collision happened when more than one key / value pair is assigned to the same index
- Separate Chaining
  - Store a linked list (or some other data structure) of key / value pairs at each index



Resolving Hash Table Collisions (Continued)

- We say a collision happened when more than one key / value pair is assigned to the same index
- (Linear) Probing
  - Upon collision, keep searching until you find the first unoccupied entry in the array



Separate Chaining Analysis

- Insert is $O(1)$ time
- Lookup and Delete are expected $O(1)$ time
  - Lookup and Delete depend on how many elements are in the linked list at index $h(\text{key}) \bmod n$
  - If we treat h as random (use a (c, k) universal hash family), we would expect $\frac{n}{r}$ elements at each index
  - We bound our load factor above by a constant, so we can treat it as that upper bound
  - Thus the linked list at index $h(\text{key}) \bmod n$ has expected $O(1)$ elements, so lookup and delete are expected $O(1)$ time

Linear Probing Analysis

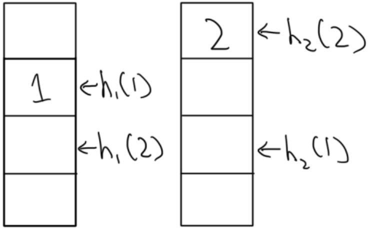
- Insert, Lookup, and Delete are expected $O(1)$ time
  - These all depend on the length of the chain starting at index $h(\text{key}) \bmod n$
  - Can show that the expected chain length is a linear function of $\frac{n}{r}$ (which we bound above)

Section 3

Cuckoo Hashing

Cuckoo Hashing

- What if instead of storing one array, what if we store two arrays of length r?
- Pick $r \geq (1+\epsilon)n \implies \frac{r}{n} \geq 1+\epsilon$
- Each array corresponds to one of two independent hash functions, h_1 and h_2, which are randomly sampled from hash family H

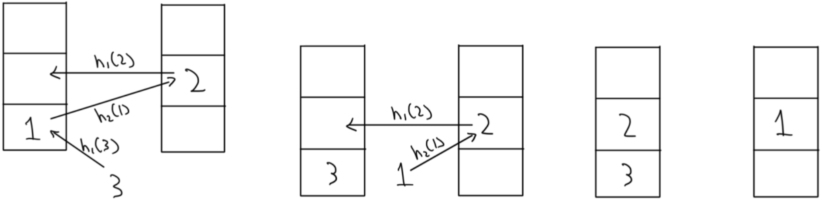


Cuckoo Hashing: Lookup and Delete

- To lookup a key, we use h_1 to index into the first array; if we don't find the key there, we use h_2 to index into the second array
- To delete a key, we follow a similar process
- Importantly, we search at most two locations (one entry per array), so lookup and delete take worst case $O(1)$ time
- But how does insert work?

Cuckoo Hashing: Insert

- Hash the key using h_1 and check to see if the corresponding spot in the array is available
- If the location is occupied:
  - Put our new key / value in the location, take the old key / value pair
  - Hash the old key with h_2 to find its spot in the other array
  - If the spot is in the other array is occupied, repeat this process (loop)



Cuckoo Hashing: Insert (Continued)

- Problem: Insert could infinite loop
- Solution: If we loop more than $Max\_Loop = 3\log_{1+\epsilon} r$ times, rehash the entire table

Cuckoo Hashing: Insert (Continued)

- Our present understanding: cuckoo hashing is good if we can afford costly inserts and want $O(1)$ time lookup and delete
- Claim: cuckoo hashing insert is expected amortized $O(1)$ runtime

Insert Analysis

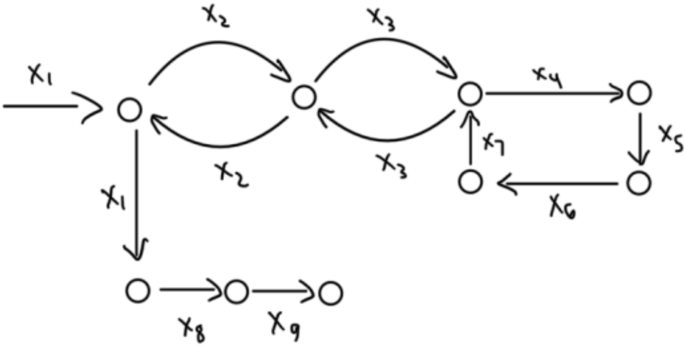
- Let h_1 and h_2 be from a universal hash family H that is at least as strong as $(1, \mathit{Max\_Loop})$ universal
  - Research has shown that with probability $1 - O(\frac{1}{n^2})$, we can treat h_1 and h_2 as independent random functions
- Let $x_1, x_2, \ldots, x_k$ be the sequence of keys we encounter during insert
  - We call $x_1, x_2, \ldots, x_k$ nestless keys

Insert Analysis (Continued)

- Case 1: $x_1, x_2, \ldots, x_k$ are all distinct (and thus we have a finite sequence)

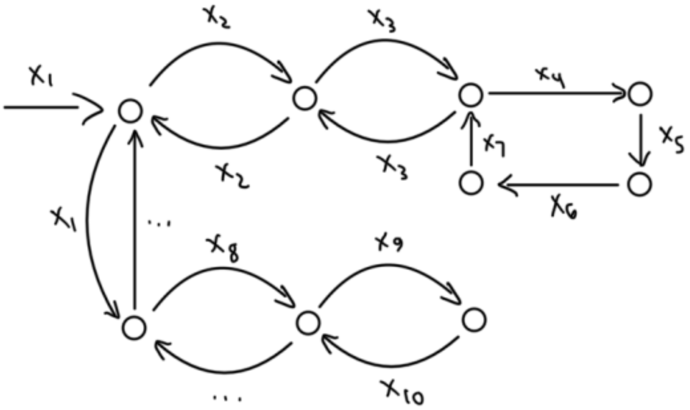
Insert Analysis (Continued)

- Case 2: $x_1, x_2, \ldots, x_k$ has some repeated value $x_i = x_j, i \neq j$, but we have a finite sequence



Insert Analysis (Continued)

- Case 3: $x_1, x_2, \ldots, x_m, \ldots$ has some repeated values and forms an infinite sequence (so we need to rehash the table)



Insert Analysis (Continued)

- With probability $1 - O(\frac{1}{n^2})$ we treat h_1 and h_2 as random functions and continue on with the analysis
- With probability $O(\frac{1}{n^2})$ the worst case might as well happen and we rehash

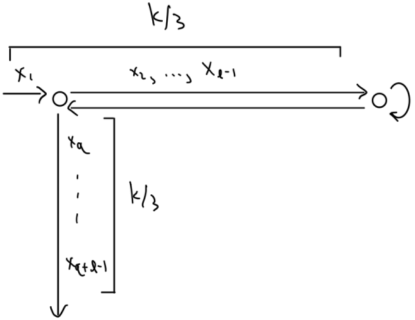
Insert Analysis: Lemma

Lemma

For a sequence of nestless keys that has not formed a closed loop, $x_1, \ldots, x_k$, there exists a consecutive subsequence $x_q, \ldots, x_{q+\ell-1}$ of distinct keys where $x_1 = x_q$ and $\ell \geq \frac{k}{3}$.

"Proof" by Picture

Worst case:



Insert Analysis: Cases 1 and 2 Bounds

- By the previous lemma, there exists a sequence of at least $\frac{k}{3}$ distinct nestless keys, $b_1, \ldots, b_v$
- Then $h_1(b_1) = h_1(b_2), h_2(b_2) = h_2(b_3), h_1(b_3) = h_1(b_4), \ldots$ (or same thing, but with h_1 and h_2 swapped)
- Less than n^{v-1} ways to have v distinct keys ($v - 1$ since we treat x_1 as fixed)
- Since we are treating h_1 and h_2 are random, each way to select the distinct keys has probability $r^{-(v-1)}$

Insert Analysis: Cases 1 and 2 Bounds (Continued)

- Recall that $\frac{r}{n} \geq 1 + \epsilon$
- Probability for this case is $2n^{v-1}r^{-v+1} = 2(\frac{r}{n})^{-v+1} \leq 2(1+\epsilon)^{-\frac{k}{3}+1}$
- Thus the probability that we get case 1 or 2 and see k keys is at most $2(1+\epsilon)^{-\frac{k}{3}+1}$

Insert Analysis: Case 3 Bounds

- For a sequence of k nestless keys with a closed loop, let v be the number of distinct keys
- Once again, we have less than n^{v-1} way to chose the remaining distinct keys and r^{v-1} ways to put them in the table
- There are at most v^3 ways to pick the start and end of the first loop and the start of the second loop
- Each arrangement of nestless keys occurs with probability r^{-2v} (r^{-v} for each hash function)

Insert Analysis: Case 3 Bounds (Continued)

- Altogether, the probability is bounded by

$$
\begin{aligned}
\Sigma_{v=3}^{\ell} v^3 n^{v-1} r^{v-1} r^{-2v} &= \frac{1}{nr} \Sigma_{v=3}^{\ell} v^3 n^v r^v r^{-2v} \\
&\leq \frac{1}{nr} \Sigma_{v=3}^{\infty} v^3 (\frac{r}{n})^{-v} \\
&\leq \frac{1}{nr} \Sigma_{v=3}^{\infty} v^3 (1+\epsilon)^{-v} \\
&= \frac{1}{nO(n)} O(1) \\
&= O(\frac{1}{n^2})
\end{aligned}
$$

Insert Analysis (Continued)

- We can now calculate an upper bound on the expected value for the number of nestless keys:
  - 1: there is always at least one nestless key
  - $\sum_{k=2}^{2*Max\_Loop} 2(1+\epsilon)^{-\frac{k}{3}+1}$: case 1 or 2 with between 2 and $2 * Max\_Loop$ nestless keys
  - $\sum_{k=2}^{2*Max\_Loop} O(\frac{1}{n^2})$: case 3 with $2 * Max\_Loop$ nestless keys
- All together we have an expected number of nestless keys of $1 + \Sigma_{k=2}^{2*Max\_Loop}(2(1+\epsilon)^{-\frac{k}{3}+1} + O(\frac{1}{n^2}))$

Insert Analysis (Continued)

$$
\begin{aligned}
1 + \Sigma_{k=2}^{2*Max\_Loop}(2(1+\epsilon)^{-\frac{k}{3}+1} + O(\frac{1}{n^2})) \\
&\leq O(1) + O(\frac{Max\_Loop}{n^2}) + \sum_{k=2}^{\infty} 2(1+\epsilon)^{-\frac{k}{3}} \\
&\leq O(1) + O(1) + O(1) = O(1)
\end{aligned}
$$

- Thus we expect to encounter $O(1)$ nestless keys, so ignoring when we need to rehash or resize, we have an expected $O(1)$ insert runtime

Cuckoo Hashing: Rehash Analysis

- We rehash when we have a sequence of $2 * MAX\_LOOP$ keys
- This can occur if:
  - h_1 and h_2 are not random with $O(\frac{1}{n^2})$ probability
  - There is a closed loop with $O(\frac{1}{n^2})$ probability
  - We have a $k = 2 * MAX\_LOOP$ sequence of keys that do not form a closed loop with probability

$$
\begin{aligned}
\leq 2(1+\epsilon)^{-\frac{k}{3}+1} &= 2(1+\epsilon)^{-\frac{2}{3} * MAX\_LOOP + 1} \\
&= 2(1+\epsilon)^{-2\log_{1+\epsilon} r + 1} \\
&= O(\frac{2}{r^2}) \\
&= O(\frac{1}{n^2})
\end{aligned}
$$

Cuckoo Hashing: Rehash Analysis (Continued)

- Each insert has $O(\frac{1}{n^2}) + O(\frac{1}{n^2}) + O(\frac{1}{n^2}) = O(\frac{1}{n^2})$ probability of causing a rehash
- We need to reinsert n items with expected $O(1)$ time per item, so this takes $O(n)$ time
- This holds unless we need to rehash again; n items have $O(\frac{1}{n^2})$ probability to cause a rehash, so we have $O(\frac{1}{n})$ probability of rehashing
- We have a decreasing geometric series $\implies O(n)$ expected time to rehash
- With an expected $O(1)$ time insert about $1 - O(\frac{1}{n^2})$ of the time and an expected $O(n)$ time insert the remaining $O(\frac{1}{n^2})$ of the time, we get an expected amortized $O(1)$ insert runtime

Section 4

Conclusion

Recap

- We saw looked at hash functions and families, in particular (c, k) universal hash families
- We looked at hash tables and well known hashing strategies
- We examined a new hashing strategy, Cuckoo Hashing, that has guaranteed $O(1)$ time lookup and delete, along with expected amortized $O(1)$ time insert

Questions?

Probability theory is nothing but common sense reduced to calculation.

— PIERRE-SIMON LAPLACE (1814)

Bibliography I

Charles Chen.

An overview of cuckoo hashing.

Accessed: 03-14-2024.

Erik Demaine and Oren Weimann.

Mit 6.851: Advanced data structures (spring 2007) lecture 11.

Accessed: 03-30-2024.

Jeff Erickson.

Algorithms, June 2019.

Accessed: 03-14-2024.