Dynamic Perfect Hashing

Based on perfect hash we saw before
- Continues to have deterministic $O(1)$ for search.
- Probabilistic amortized cost of:
  - $O(1)$ for Delete
  - $O(1)$ for Insert

Key ideas:
- Allocate $O(n)$ sub-tables, each $2(|S_j|)^2$ big (to allow room for growth)
- Use existing $h_j$ on insert; it likely won’t cause a collision
- Generate a new $h_j$ if insert into a subtable causes a collision using existing $h_j$
- Rehash subtable into larger subtable once it doubles in size
- Completely re-hash everything (new $h$, new $h_j$s) once $c \cdot n$ non-Search operations have taken place
- Also re-hash if sum of subtables gets too big
- Mark deleted items as deleted but don’t remove from the table until a rehash

Dynamic Perfect Hashing

Phase: Time during which one top-level hash function $h$ is in-use
- Phase ends when either:
  - We exceed overall space in our subtables
  - We’ve executed $c \cdot n$ non-Search operations

Lemma 1
- If we start a phase with $n$ keys stored in the dictionary the memory space used for the phase is $O(n)$

Lemma 2
- If we start a phase with $n$ keys stored in the dictionary, the expected time for the phase is $O(n)$

Lemma 3
- If we rehash with a top-level function, the expected time until we need to rehash again after $c \cdot n$ updates have been performed is $O(n)$

Lemma 4
- If we rehash with a top-level function $h$, with probability $>1/2$, we'll end the phase without exceeding space in our subtables
Dynamic Perfect Hashing

Allocate $O(n)$ sub-tables, each $2(|S_j|)^2$ big (to allow room for growth)
- Total space is still linear (with probability $> 1/2$)
- Time/space of $O(n)$
- Extra room allows inserting more items with good chance that $h_j$ will hash to unique location

Use existing $h_j$ on insert; it likely won't cause a collision
- The chance that $h_j$ can handle $S_j$ additional insertions (until subtable doubles in size) is $> 1/2$
  - We doubled the table size, but also doubled $n$. Previous analysis (w.r.t perfect hashing) about probability of collisions in subtable holds

Generate a new $h_j$ if insert into a subtable causes a collision using an existing $h_j$
- Since chance of collisions is $< 1/2$, for a given table, the expected number of new $h_j$ chosen (during time from table being a certain size to doubling in size) is $< 2$
- Time to rehash with new $h_j$ is the size of the table
- Since sum of table sizes remains $O(n)$, amortized cost of $n$ operations (not requiring table resizes) to rehash with new $h_j$s is $O(n)$.

Rehash subtable into larger subtable once it doubles in size
- The cost of this rehashing across all subtables is $O(n)$
- If sum of subtables is too big, we're doing a big rehash anyway
Dynamic Perfect Hashing

Completely re-hash everything (new \( h \), new \( h_j \)) once \( c \cdot n \) non-Search operations have taken place

- Cost of this rehash is \( O(n) \)
  - The new table could be:
    - bigger (lots of inserts)
    - smaller (lots of deletes)
    - same size (mixture of inserts/deletes)
- Since we’ve done \( c \cdot n \) operations, amortized cost is \( O(1) \)

Also re-hash if sum of subtables gets too big

- Probability is \(<1/2\) that sum of subtables will get too big before \( c \cdot n \) non-Search operations occur

Mark deleted items as deleted but don’t remove from table

- They’ll be removed at the beginning of the next phase
- Table size will be based on current number of elements
  - Table can shrink if items are deleted

Conclusion

- Given a table with \( n \) items, we can handle a sequence of \( n \) Insert/Deletes with expected amortized time:
  - Space used: \( O(n) \) (note that constant is large: 35)
  - Time used: \( O(n) \)
  - Searches are \( O(1) \)
Polyhash
Larry Carter—1979

Choose a finite field (e.g., integers modulo a prime)
- \( p = 2^{31} - 1 \) is prime and \( \mod p \) is easy to compute

For each \( x \) in the field, we define \( h_x(key) \)
- Write key as blocks: \( key = a_0a_1...a_{s-1} \)
- \( h_x(key) = a_0 + a_1x + a_2x^2 + ... + a_{s-1}x^{s-1} \)

- Number of hash functions that cause a \( b \) in \( U \) to collide is \(< s \).
  - \( h_x(a) = h_x(b) \) iff
    - \( a_0 + a_1x + ... + a_{s-1}x^{s-1} = b_0 + b_1x + ... + b_{s-1}x^{s-1} \)
    - or, \( (a_0-b_0) + (a_1-b_1)x + ... + (a_{s-1}-b_{s-1})x^{s-1} = 0 \)
  - The above function has at most \( s-1 \) solutions

- Probability of \( h_{ij} \), chosen at random, causing two distinct keys to collide is \( s/p \)

Computing \( k \) mod \( 2^{31}-1 \)
- \( k = 2^{31}q + r \) (shifting is cheap)
- \( 2^{31} = 1 \mod (2^{31}-1) \)
- \( k = (q + r) \mod (2^{31}-1) \)

If \( q+r \) is bigger than \( 2^{31}-1 \) (overflow), repeat process

Compute polyhash using Horner's rule
- \( a_0 + a_1x + ... + a_{s-1}x^{s-1} = a_0 + x(a_1 + x(a_2 + ... + x(a_{s-1}))...) \)
- \( s-1 \) additions, multiplications, and mods

If \( m < 2^{31} \)
- Take hashed value mod \( m \)
- \( E[X_{ij}] < s/p + 1/m \)

Chance polyhash collides  Chance mod \( m \) causes collision