The UNIVERSITY of VERMONT

Benjamin Fenster
CS-104 (Lee)
18 December 2004

Hashing

I. Preliminaries
   a. Key Indexed Array
      i. Stores an item x with key k in the array at position k. a [k] = x
      ii. Pro: Insertion / access is always O(1)
      iii. Con: If there's even one element with a large key, we need a huge array (example: keys of 0, 2, 3, 5, 1000000)
   b. Want to fix the array size at M
      i. Map keys to integers in (0, M – 1)
      ii. May get two keys mapping to the same position (called a "conflict")
      iii. On average N / M conflicts
   c. Hash Table
      i. Definition: Hash table is an array where the array index is mapped from a key using a hash function
      ii. Note: Large array yields fewer collisions.
      iii. More Space = Faster Search
   d. Hash Function
      i. Given a hash table of size M, hash function is a mapping from any key to a hash table address (i.e. array index)
      ii. What makes a good hash function?
         1. Want to minimize collisions
         2. Modular hash function! H(k) = k mod M
         3. M should be a prime number to best minimize collisions
   e. Key Generation
      i. Convert non-integer keys into integers
      ii. Floating Point
         1. If keys are in [s, t] then take $k = ((f - s) / (t - s)) * 2^b$ where $2^b > M$
         2. So if M = 17, use $k = ((f - s) / (t - s)) * 2^5$
         3. Results in $0 \leq k \leq 2^b$
      iii. ASCII Keys
         1. Option 1
            a. Add all the individual codes (0 to 127)
            b. Works very well for short keys
            c. Example: Consider a 2-character key
               i. Range of possible keys is 0 to 127 * 2
               ii. So 255 distinct keys are possible
               iii. Have 128^2 possible strings (16384)
               iv. So 64 distinct ASCII keys will be converted to a single integer key (64 strings to each integer) on average
            d. Example: An 8-character string
               i. 128^8 possible strings, almost 1024 possible keys
               ii. About $128^8 / 1024$ strings map to each integer key. Way too crowded
         2. Option 2
            a. Transform the keys piece by piece
            b. Treat an ASCII string as a base 128 number
            c. ABD = $64 * 128^2 + 66 * 128^2 + 68 = \ldots$
            d. Generates keys too large to even be stored!
            e. We will convert the string to an integer one character at a time.
            f. Horner's Rule
               i. $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$
               ii. $P(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_n)\ldots))$
               iii. $P(x) = ((..(a_n)x + a_{n-1})x + \ldots + a_3)x + a_2)x + a_1)x + a_0$
            g. Consider "a long key" (8 characters)

i. Using Horner's rule.
ii. $97 * 128^7 + 108 * 128^6 + \ldots + 121 * 127^6$
iii. (((97 * 128 + 108)127 + 111)127 + …)128 + 121

h. This is more *efficient* but we still have the overflow problem

i. Modulus Equivalence Rule
   i. (a x + b) % M = ((ax % M) + b) % M
   ii. Now calculate key (97 * 128 % M …

II. Collision Resolution

a. Separate Chaining
   i. Have some "bucket" structure (perhaps a linked list) to catch conflicts
   ii. May get a long list of elements at the same address. Then efficiency of the hash table drops (starting to look more like a linear search.
   iii. Average length of the chain is $N / M = \lambda$
   iv. λ is the "load factor"
   v. Variations
      1. Store the first element in the table itself; only start making a linked list if there's a conflict. Marginally better.
      2. Overflows are supposed to be rare, so linked lists are fine for bucket structures if you want to use separate chaining
   vi. Operations
      1. Find: Hash key and traverse list
      2. Insert: Hash, traverse, put at beginning if not found
      3. Remove: Find(), then remove from list
   vii. Runtime
      1. Chains are not sorted. Duplicates are illegal
      2. $N / M = \lambda$ for unsuccessful search, or insertion
      3. $1 + ((N - 1) / M) / 2) \approx 1 + \lambda / 2$ for successful search
      4. (At least one probe, average load *excluding* the element that is already counted in that "at least one")

b. Open Addressing
   i. Concept
      1. When an item conflicts, try some other address for it.
      2. Need some rule to find an open address
      3. Use addresses $h_i(k) = (h(k) + f(i)) \% M$ for I = 1, 2, 3, …
      4. Just change f(i) however you like.
   ii. Types
      1. Linear Probing: f(i) = i
      2. Quadratic Probing: $f(i) = i^2$
      3. Double Hashing: $f(i) = i(k \% M_2 + 1))$ for $M_2 \neq M$
   iii. Problem: Keys may get clustered together. Goal is to avoid / reduce clustering.
   iv. Operations
      1. Find(k): hash(k). If not found, probe new addresses of the table until k is found or an empty cell is found.
      2. Insert(elt). hash(elt → k). If not empty, probe new addresses until k or null found.
      3. Remove(k). hash(k). If not found, keep probing until k or empty cell found. Put a deletion marker so the cell won't be counted empty in future searches. Called a "tombstone." Otherwise would need to rehash all the elements that originally hashed to the same address as k.
   v. Runtime (Linear Probing)
      1. Unsuccessful: $0.5(1 + 1 / (1 - \lambda)^2)$ $\lambda \leq M$
      2. Successful: $0.5 (1 + 1 / (1 - \lambda))$
   vi. Quadratic Probing
      1. Tries to alleviate clustering problem.
      2. "Hopping distance" increases each time, so probably get fewer clusters.

vii. Double Hashing
   1. When there's a conflict calculate a new address with a new hash function
   2. Further improvement on quadratic probing
      a. Still get some clustering with quadratic probing
      b. From linear, called "primary clustering"
      c. From quadratic, "secondary clustering"
   3. Example
      a. Say $h(k) = k \% 20$ $h_2(k) = (k \% 11) + 1$
      b. $f(i) = I * (k \% M_2) + 1)$ where $M_2 \neq M$
   4. Runtime
      a. Unsuccessful: $1 / (1 – \lambda)$
      b. Successful: $(1 / \lambda) \ln (1 / (1 – \lambda))$

III. Dynamic Hashing
   a. Increases hash table size when necessary
   b. Rehashes all items each time the size is changed
   c. Can also reduce size when too few elements are left (but not done in all implementations)
   d. Double size when more than δ full. Halve size when less than δ full.
   e. This fixes runtime of open addressing, but occasionally an operation will be very slow as all elements are rehashed to a bigger / smaller hash table.
   f. Operations
      i. Find(k). Same as any open addressing scheme
      ii. Insert(Node* n). if $N > \delta_1 M$ elements after insertion, rehash to a new table of size minPrime(2M) (smallest prime greater than 2M)
      iii. Remove: If $N < \delta_2 M$ after deletion, rehash to table of size minPrime(M / 2)
   g. Runtime
      i. Rehashing tables O(N) probes
      ii. Inserting N elements takes O(N), rehashing takes O(N) so total O(N) on average
      iii. Still constant runtime for a single operation on average

IV. Extensible Hashing
   a. Concepts
      i. External hashing (on disk)
      ii. Uses disk accesses as runtime metric
      iii. Extendable hash table made up of *directory* and *data* pages (disk pages)
      iv. Closer to "hashed datafile" than a simple index
      v. Have a directory of first few bits of keys and pointers to *data* (key data) pages
   b. Operations
      i. Find(k). hash(k), into directory page, then search corresponding data page
      ii. Insert(Node* n)
         1. hash(n → k) into directory
         2. Insert n into the appropriate data page
         3. If that page overflows, split it; expand the directory to cover an additional
         4. Other pages don't need to split, so two entries point to the same page.
         5. If another split occurs later, directory is okay – just change one pointer.
      iii. Remove(k)
         1. Find(k), remove it.
         2. If underflows, merge with siblings (and shrink directory if applicable)
         3. This isn't done in all implementations
   c. Runtime
      i. Find, insert, remove take two page accesses – *one* if directory is cached
      ii. Plus one additional page access to retrieve the record itself
   d. Storage Requirements
      i. Let M be the number of elements that fit on a page. Let N be the number of elements inserted. Then the number of pages $D = N / (0.69 * M)$
      ii. On average, pages are 0.69 full.
      iii. Number of pages for directory = $O(N^{1 + 1/M} / M)$