Hashing

Theory and Terms

* technique used for performing insertions, deletions, and searches in constant O(1) AVERAGE time
* hash table maps huge set of ***possible*** keys into a much smaller number of N buckets by applying a compression function to each hash code
  + THIS POSSIBLY SPEEDS UP FINDING DATA!!!
* 2 functions required
  + hashCode
  + compression
* overall process
  + key and value (data set)
  + key portion is entered into hashCode function
  + hashCode value is created from the key
  + hashCode is entered into a compression function
  + a bucket number is created (0,..N-1)
  + key and value are entered into that bucket (probing)
* n = number of total keys (words) we are storing in the table (< 1,000,000 English words in reality)
* N = number of BUCKETS to store the keys (N is a bit larger than n)

|  |
| --- |
| The Big Picture (but flawed) |
| Starting with a data set “n” |
| How would we find a person out of 319,166,000 (n) people?  http://images.all-free-download.com/images/graphiclarge/us_map_silhouette_vector_144872.jpg  What would be an easier way to finding a person that we broke up the search? |
| Compressing the data into Buckets (N) |
| How about the 50 states?  http://www.jewishvirtuallibrary.org/jsource/images/us-map2.gif |

|  |
| --- |
| Check if this works (Distribution) |
| This an even distribution?    called Uniform Hashing Assumption |
| Load (N/n) |
| 319,169,000/50 = 6,383,380 people per bucket is evenly distributed |

|  |
| --- |
| The Big Picture |
| File:Hash table 5 0 1 1 1 1 1 LL.svg  Notice the “collision” of John Smith and Sandra Dee |

Key and Value

* not always spelled out
* key, usually a unique identifier
* value is the data that belongs to that key
* value could be optional (but not always applicable)
  + very limited in usability
  + ***value is not changed!!***
* key will be used to hash (determine what bucket the whole group <key, value> will be placed)

|  |
| --- |
| Key and Value entered into a Hash Table |
|  |

* the key might have to be translated into a distinct value!
  + tic-tac-toe board
  + treat all tic-tac-toe layouts (hash table) as a base 3 number
    - spaces are = 0
    - X = 1
    - O = 2
    - combine all 9 squares into a number
      * 102000000

Collisions and Probing

* collision
  + where several keys hash into the ***same*** bucket
  + but a collision is unavoidable, but at least keep the number of collisions small

|  |
| --- |
| Chaining – Collision handling |
| http://www.cs.wcupa.edu/~rkline/DS/images/chaining.gif |

* “probing”
  + **used to avoid a collision**
  + finding that empty spot to place the data
  + several different strategies below to do this
  + may take several probes
    - this is not good (or bad), but keep an eye on since this may become an ordeal
* chaining (open hashing)
  + **accepts the collision**
  + each bucket will not store just one value, but a linked list of entries
  + linked list is called the chain
  + can put as many keys into that bucket as you need
  + in storing English words you might have a word that has several meanings

|  |
| --- |
| Chaining in an example |
|  |

|  |
| --- |
| ****Chaining Example**** |
| **h(k) = k mod 10**  **N = 10 (# of buckets!!)**  **inserting keys 12, 18, 13, 2, 3, 23, 5 and 15**  **Result:** |

If the data portion was a linked list, why would inserting at the rear be inefficient?

|  |
| --- |
| Chaining Exercise |
| N = 13    **h(k) = k mod 13**  Using the hash table pictured above, enter these keys, and show the resulting table. (Hint, first one is already done) Answerb:   1. 83 46 87 5 64 21 4 60 27 43 81 96 7 37 12 63 18 56 13 59 86 26 53 76 92 36 25 49 2. What would the AVERAGE search time be for this example? |

* linear probing (closed hashing)
  + if a collision is detected, moves to next available spot
  + Each attempt to find an open slot (i.e. calculating h( k, i )) is called a ***probe***
    - next “spot” is usually +1 (intervals between probes) from where it should have been placed
    - but this can repeat if THAT spot is taken, with the next +1 spot
    - ***can wrap back around to 0!!***
  + can have an issue in clustering
    - when the table starts to fill up, the performance nears O(N)

|  |
| --- |
| Wrapping in Probing |
|  |

|  |
| --- |
| ****Linear Probing Example**** |
| **h(k) = k mod 10**  **N = 10**  **inserting keys 12, 18, 13, 2, 3, 23, 5 and 15**  **Result:** |

|  |
| --- |
| Linear probing Exercise |
| N = 29    **h(k) = k mod 29**  Enter these keys, and show the resulting table. Answerb:  19 83 46 87 5 64 21 4 60 27 43 81 96 7 37 12 63 18 56 13 59 86 26 53 76 92 36 25 49 |

* quadratic probing (closed hashing)
  + “spread the love”, spreads the overall data better
  + uses PART of a quadratic function for placement of the value
    - f( i ) = c2i2 + c1i + c0
    - but usually f( i ) = i2
  + again, if already full, tries the NEXT increment of i and puts it through the quadratic function

|  |
| --- |
| Quadratic Probing i2 Example |
| N = 10  **h(k) = k mod 10**  **inserting keys 8**9, 18, 49, 58, 69  **Result:** |

|  |
| --- |
| Quadratic Probing Exercise |
| N = 29    **h(k) = k mod 29**  Enter these keys, and show the resulting table. Answerb:  62 34 44 47 61 85 46 70 77 90 26 71 92 11 69 55 40 23 27 96 58 83 |

Load Factor - performance of a hash table

* how much stuff do you pack into a hash table
* how large should the buckets get before we may have to resize?
  + N/n (division method, there is a multiplication method)
    - n = elements
    - N = buckets
  + 0.8 or 80% percent full ***is good!!!***
* N should be a prime
  + so a key does not divide into it evenly
  + pick smallest Prime value where
  + Nish = (expected # of entries)/0.8 (again, division method)
    - increment Nish to next available Prime number
* load factor stays low and hashCode and compression function are good, and no duplicate keys
  + **then** linked lists (chains) stay short (1 – 4 items)
  + **then** each operation takes AVERAGE constant time O(1)
    - running time depends on how big the chain is
    - if chain is longer, time will increase
  + if load factor gets really BIG (n >> N) (n way bigger than N)
    - n keeps getting bigger, but N does not adjust
      * will start to approach linear time Theta(n), worst case
* Why not choose a really BIG N?? (To reduce the load factor, to an extreme)
  + start up cost, create it will take O(N) time
  + memory usage (and possible waste)

|  |
| --- |
| Load Factor Exercise |
| N = 13    **h(k) = k mod 13**  19 83 46 87 5 64 21 4 60 27 43 81 96 7 37 12 63 18 56 13 59 86 26 53 76 92 36 25 49  Remember the **chaining** problem above?   1. Calculate the load for this **already completed** example? (BTW, it’s N/n) 2. What would be a better N? 3. Redo problem above with your new N number of buckets. 4. What would the AVERAGE PROBING time be for this example with a new N? Remember to +1 if we had to chain it.  |  |  | | --- | --- | | **Value Placed** | *# of movements to place* | | **19** | 1 | | **83** |  | | **46** |  | | **87** |  | | **5** |  | | **64** |  | | **21** |  | | **4** |  | | **60** |  | | **27** |  | | **43** |  | | **81** |  | | **96** |  | | **7** |  | | **37** |  | | **12** |  | | **63** |  | | **18** |  | | **56** |  | | **13** |  | | **59** |  | | **86** |  | | **26** |  | | **53** |  | | **76** |  | | **92** |  | | **36** |  | | **25** |  | | **49** |  | |

Hash Code and Compression functions

* some bad hash codes/compression functions will create more collisions!
* remember a key gets mapped to a hash code
  + h(k)
  + should be easy to compute
  + should be able to distribute <key, value> pair uniformly from 0 to N
* then that value gets mapped by compression function
* then that value get mapped to buckets [0… N – 1]
* IMPORTANT
  + many hash’s INCLUDE the compression within the function
  + you can identify the compression easily by finding % (modulus)

|  |
| --- |
| Example Java Hash Function |
| /\*\*  \* A hash routine for String objects.  \* @param key the String to hash.  \* @param tableSize the size of the hash table.  \* @return the hash value.  \*/  public static int hash( String key, int tableSize )  {  int hashVal = 0;  for( int i = 0; i < key.length( ); i++ )  hashVal = 37 \* hashVal + key.charAt( i );  hashVal %= tableSize;  if( hashVal < 0 )  hashVal += tableSize;  return hashVal;  }  Hash function portion, and Compression function portion  Notice that tableSize is our N |

* Ideally we would like to accomplish where we can map each key to a random bucket, each bucket ***equally likely***.
  + Collisions cost us time, slow search, remove, everything…
  + Our ***TARGET*** of O(1) will be AVERAGE of all items catalogue in the hash table
* careful, hash codes are often negative

Hash functions for Integers

* some bad compression functions will create more collisions!
* having a Prime number SOMEWHERE is important so values aren’t divisible by some number
  + collisions are greatly reduced
* if N is not a Prime
  + then use a Prime number somewhere in your hash code to reduce collisions
* with a Prime N (buckets), doesn’t matter if values are divisible by a common value

|  |
| --- |
| Good Hash code Ex. #1 |
| ((a \* hashCode + b) % P) % N)  a,b,P: positive  P some large prime, should be significantly larger than N  Now N (buckets) doesn’t not have to be prime |

Hash Functions for Strings

* most keys will be strings!
* base 127 number, convert to int
  + ASCII table has 127 characters
  + 127 is a Good prime number too!!
* not all hash functions are good
  + we demonstrate the bad ones so you don’t make the same mistake
* sum all ASCII values from a String
  + order of characters does not matter

|  |  |  |
| --- | --- | --- |
| Summing Strings for a hash Code value | | |
| N = 13 | | |
| inserting Lupoli (ASCII sum = 629, hash value = 5 (629%13)) | | |
|  | | |
| bucket (638%13) = 1 | New York (751 % 13) = 10 | Ocean Park (916 % 13) = 6 |

What are the issues with this in a large table? (Which it really would be)

|  |
| --- |
| Issues with Summing String hash function |
| 1. If the table size is LARGE, many strings that are smaller (which are MANY) will be placed in the first half of the table! (not a question) 2. Will there be any values at 0? 3. What would the MIN value for a 3 CAPITAL letter word? 4. What would the MAX value for a 3 CAPITAL letter word? 5. Would the % have any affect at all on smaller words with a LARGE N 6. Would distribution be uniform? |

|  |
| --- |
| Good Hash code for Strings #1 |
| public static int hash( String key, int tableSize )  {  int hashVal = 0;  for( int i = 0; i < key.length( ); i++ )  hashVal = 37 \* hashVal + key.charAt( i );  hashVal %= tableSize;  if( hashVal < 0 )  hashVal += tableSize;  return hashVal;  }  from Weiss book |

|  |
| --- |
| Good Hash code for Strings #2 |
| public static int hash( String key, int tableSize )  {  int hashVal = 0;  for( int i = 0; i < key.length( ); i++ )  hashVal = (127 \* hashVal + key.charAt( i ) % 16908799 );  // the Prime is very big so hashVal does not go OVER the limit for int  hashVal %= tableSize;  if( hashVal < 0 )  hashVal += tableSize;  return hashVal;  } |

|  |
| --- |
| Bad hash codes on Strings (English Words) |
| * some letters come up more than other in our English Dictionary example * a few ***bad hash*** codes   + sum ASCII value of characters     - most words are short     - w/ 70000 buckets, there will be many collisions since most will only fit the first 500 or so buckets     - adding just does not thinking about the order the letters appear     - pat, apt, tap, all collide   + choose first 3 letters of a word     - with 26^3 buckets     - biases in English words       * like “pre-“     - but no words with some other letters       * like “xzq”     - will not have an even distribution of key |

Chaining Hash Table ADT

* public functions
  + boolean contains( AnyType x )
    - returns true if x is present in the table.
  + void insert (AnyType x)
    - if x already in table, do nothing.
    - otherwise, insert it, using the appropriate hash function.
  + void remove (AnyType x)
    - remove the instance of x, if x is present.
    - otherwise, does nothing
  + void makeEmpty()
* private functions
  + void rehash()
    - Create new double-sized, empty table, then copies values to new list
  + int myhash(AnyType x)
    - hash and compression function
    - returns hashVal which will be the index value used to drop
  + int nextPrime(int n)
    - method to find a prime number at least as large as n.
  + Boolean isPrime(int n)
    - test if a number is prime.

Determine what scenarios would each function below performance at it’s:

|  |  |  |
| --- | --- | --- |
|  | contains() | remove() |
| best |  |  |
| worst |  |  |
| average |  |  |

Answerb:

Why not always use a hash table?

* Does not support ordered output. (No findMin or findMax)
* Does not support finding all elements in a given range
* How to design the hash function is not always clear.

Insert multiple copies of a key??

* maybe difference definitions (value)
* 2 approaches

1. insert both, find arbitrarily returns one entry

could create a find ALL (idea for project?)

1. replace it. Only allowed to be on entry with that key. Overwrite old value

(again, project idea, value could be another linked list, where each definition is stored for a single key (word))

can’t change key since it would now be in the wrong bucket

Double hashing

* another collision resolution strategy, to eliminate clustering problem
* probing strategy (in order)
  + hash1(k) mod N
  + if there ***IS A COLLISION***
  + ( hash1(k) + 1 \* hash2(k) ) % N
  + if there is ***STILL*** a collision
  + ( hash1(k) + 2 \* hash2(k) ) % N
  + …
  + if the probe has *another* collision, we don’t recalculate hash2, but we multiple the value given by hash2
    - saves some time
* requires a second has function
  + should be different than hash1
  + this function can NEVER evaluate to 0 since it would put it right back in the same spot
  + one suggestion
    - hash2( k ) = R - ( k mod R ) with R a prime smaller than N

|  |
| --- |
| Double Hashing Example |
|  |

Resizing Hash Tables

* Too full?
  + when resizing happens, one operation can take linear time O(n) but this only happens one time
* performance depends on the load factor
  + if you find a lot of items in buckets after starting an application, we have to resize the hash table
* if load factor N/n is too large, we lose O(1) time
  + enlarge hash table when load factor > some small constant “c”
    - typically c = .075 or slightly greater
    - for chaining
  + allocate a whole new array to hold our hash table
    - new array should be at least twice as large
  + walk through the old array, find all entries stored, hash them into new hash table
    - go through recomputed it’s hash code, recomputed it’s compression function, then move into move array into the right bucket
    - we are changing the old compression function
    - many items are going to be placed into a new bucket
* Too empty?
  + shrink when load factor N/n drops below a < 0.25
  + frees memory

FYI Section

Applications of hash Trees

* Speed Game Trees
  + tic tac toe
    - there is a way of completing the same tic-tac-toe board various steps (moves)
    - some grids can be reached through many sequences of moves
      * many many
      * be evaluated many times
    - show drawing at 29:30
  + description is at 29:00
  + hash gable remember all of the boards given, and we can store the scores
  + maintain hash table of previous encountered grids
    - not all
    - duplicates only start to happen at depth 3 (or 3 turn in the game)
  + key is the grid
  + value is the score for that grid
  + min and max algorithm
    - checks if each gird it visits is in the hash table
      * Yes: return score
      * No: evaluate the score (by running min and max recursively) and store it in the hash table

36:00 when describing how the trees could be compared for the same board layout

* empty the table each time they move
  + all depths are not needed anymore
  + review around 41:00 again

Answers

|  |
| --- |
| Chaining Exercise |
|  |

|  |
| --- |
| Linear Probing Exercise |
|  |

|  |
| --- |
| Quadratic Probing Exercise |
|  |

|  |
| --- |
| Performance of contains() and remove() |
| **Performance of contains()**  best: selected list is empty or key is first → O(1)  worst: Let N be number of elements in the hash table. Worst case: all N elements are in one list (all have the same hash value) and key not there → O(n)  average: Suppose there are M buckets and N elements in table. Then, expected list length = N/M → O(N/M) = O(N) if M is small  = O(1) if M is large  **Performance of remove:**  best:  k is 1st element on list, or list is empty: O(1)  worst:  all elements on one list : O(n)  average:  O(N/M) : O(1) for λ (load factor)<= 1 |

Sources

In general

(at 6:00) <http://www.youtube.com/watch?v=UPo-M8bzRrc>

<http://www.youtube.com/watch?v=LmkfLAPj5Xo>

<http://research.cs.vt.edu/AVresearch/hashing/index.php>

Chaining

<http://www.cs.usfca.edu/~galles/visualization/OpenHash.html>

Linear and Quadratic Probing

<http://www.cs.usfca.edu/~galles/visualization/ClosedHash.html>

String Sum hashing

<http://research.cs.vt.edu/AVresearch/hashing/strings.php>

Tic-Tac-Toe Hash table

<http://www.seas.gwu.edu/~simhaweb/champalg/chess/chess.html>

<http://scienceandcode.moomug.com/?p=70>

www.cs.ucf.edu/courses/cot4810.spr2003/rwilson1.ppt‎