Hash Tables

Practical way to implement symbol tables that doesn’t work with ordered ops- only search/ insert

Basically reducing the symbol table to an array

Use a hash function to convert the ST key to an integer (an array index).

Save items in a key-indexed table- at that location in the array.

Issues:

* Computing the hash function (can get complicated for complicated types if data)
* Equality test (method for checking whether two keys are equal); NOT COMPARETO()
* Collision resolution: algorithm and data structure to handle two keys that hash to the same array index

Classic space-time tradeoff:

* If no space limitation: trivial hash function with key as index
* If no time limitation: trivial collision resolution with sequential search
* Space and time limitations: hashing (the real world)

Hash function implementation:

Idealistic goal: scramble keys uniformly to produce table index

* Needs to be efficient
* Each table index should be equally likely for each key

E.g. Phone numbers:

* Bad: first 3 digits (many bases on single geographic regions)
* Better: last three digits (assigned in chrono order within geo region)

Practical challenge: need different approach for each key type

All Java classes inherit a method hashCode() that returns a 32-bit int

**If x == y, y == x / if x != y, y != x**

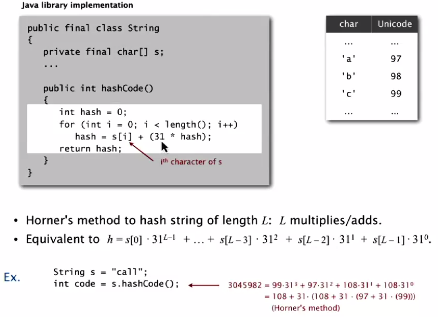
Default implementation: memory address of x

Customized implementations: Integer, Double, String, File, etc

User defined implementations: obviously, you’re on your own\

Implementation:

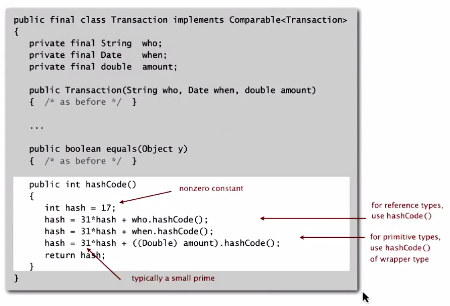
* Integer: return value
* Boolean: return 1231 for true, 1237 for false
* Double: convert to 64 bit, xor most significant 32 bits with least significant
* String: treats string as huge number and computes as number % 32



Performance optimization: cache hash value in inst variable (since strings are immutable)

Next User Defined Types @10:24

Example of implementing a hashCode() for our own data type:



Make use of all pieces of data that we have and use hashCode imps for types of data that we’re using.

1. Use small prime numbers and add them to hashCodes
2. Take bits, scramble bits and use them

Standard recipe for user-defined types:

* Combine each significant field using the 31x + y rule
* If field is a primitive, use wrapper type hashCode()
* If field is null return 0
* If field is a reference type, use hashCode (recursively)
* If field is an array, apply to each entry

In practice: recipe works reasonably well; is used in Java libraries

In theory: “universal” hash functions exist

Basic rule: if computing your own, use the whole key. For state-of-the-art hash codes, consult an expert

Hash codes return an int between -231 and 231-1

If we have a table of size M (an array to store the keys), we need an int value between 0 and M-1. M is usually a prime or a power of 2. Reason: in order to bring hash value between 0 and M-1, we mod the hash code % M



Assumption: each key equally likely to hash to an integer between 0 and M-1

Bin and ball (the birthday problem): how many balls do you need to throw (if thrown at random) before two balls end up in the same bin?

C:\Users\Zach\AppData\Local\Microsoft\Windows\INetCache\Content.Word\ballbin.png

* Birthday problem: expect 2 balls in the same bin after ~√πM/2 tosses
* Coupon collector: expect every bin has >= 1 ball after ~M log M tosses
* Load balancing: after M tosses, the most loaded bin has O(log M / log log M ) balls

Example below (even distribution of hashed words in Tale of Two Cities)



Separate chaining

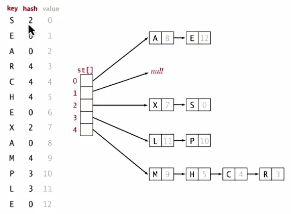
We use this when two separate keys hash to the same index (birthday problem)

Otherwise, you need a quadratic amount of memory in order to avoid such collisions

Based on the coupon collector + load balancing we know that collisions will be evenly distributed.

Option 1:

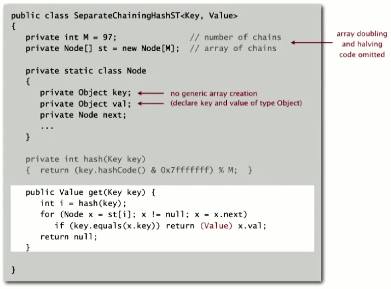
Linked List for each array index (size of M < N)



Insert at front of chain at each index

Search requires looking through a list at the index, but that should only be length up to N/M (due to the even distribution of collisions)

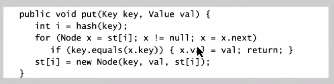
Separate chaining ST implementation:



Keys and values must be objects, since you cannot have an array of generics

When get(), we will need to cast

Put() implementation:

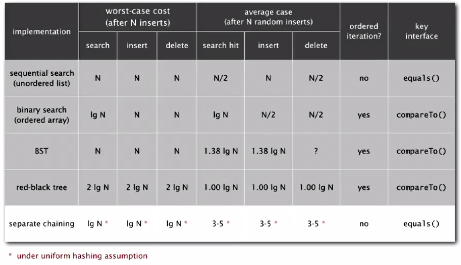


Analysis of separate chaining:

Proposition: under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of N / M is extremely close to 1

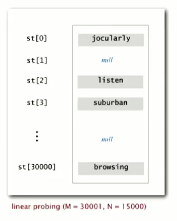
Consequence: number of probes for search/insert is proportional to N / M:

* If M too large -> too many empty chains
* If M too small -> chains are too long
* Typical choice is M ~ N/5 (constant time ops and not much extra space)



Open Addressing:

When a new key collides, find the next empty slot and insert there. Instead of using space for linked lists, allocate a larger array. Array size must be larger than the number of keys we expect.

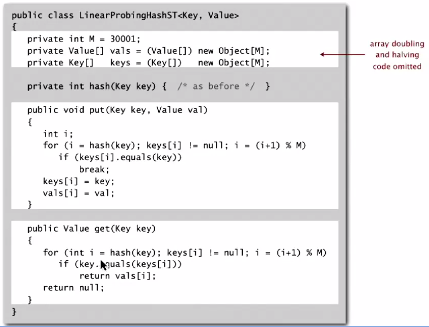


Purpose: eliminate length of list that would be required when search through a linked list implementation

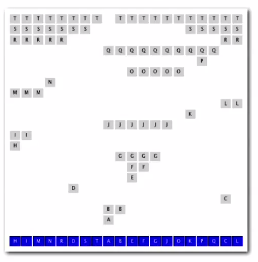
Hash: Map key to integer I between 0 and M-1

Insert: Put at table index I if free; if not try i+1, i+2, etc. (using parallel arrays)

Note: Array size M must be greater than the number of key-value pairs N- AND must resize if/when hash table becomes too full. The table should remain about half empty at all times



The problem of clustering: (a cluster is contiguous block of items) once a hash table begins to fill, new keys are likely to hash into the middle of large clusters and take significant amounts of time for search/insert

’

Knuth’s parking problem

C:\Users\Zach\AppData\Local\Microsoft\Windows\INetCache\Content.Word\knuthparking.png

Imagine cars on a one-way street with M parking spaces. Each desires a random space i: if space i is taken, try i+1, i+2, etc.

When M/2 cars (**half** parking spaces are occupied), the mean displacement is ~3/2 (on average, half people find parking after one space and half have to look one extra)

**Full**: With M cars, the mean displacement is ~ √ π M/8

Under uniform hashing assumption, the average number of probes in a linear probing hash table of size M that contains N = a M keys is:

*(a is alpha, or number of keys, as a fraction of M. so a (keys) out of M (array size)*

C:\Users\Zach\AppData\Local\Microsoft\Windows\INetCache\Content.Word\time.png

Parameters:

* M can’t be too large, uses too much memory. Use array resizing to maintain a ~half-full array
* M can’t be too small, search time explodes
* Typical choice: a = N / M ~ ½ (number of probes for search hit is ~ 3/2 and search miss ~ 5/2 )



Linear probing provides another option for constant time search, insert, delete (unordered)

**Separate chaining vs linear probing**

Separate chaining:

* Easier to implement delete
* Performance degrades gracefully
* Clustering is less sensitive to poorly designed hash functions

Linear probing

* Less wasted space
* Better cache performance

**Context of hashing and practical applications**

Cost of computing some large keys (like long strings) be higher than search/insert. String hashes in Java were originally computed by checking every 8-9 evenly spaced characters, but this lead to many collisions and poor performance due to excessive collisions.

Uniform hashing *assumption* is not necessarily good enough (i.e. in quicksort, we *create* randomness, but in this case we *hope for* randomness)

Examples: aircraft control, nuclear reactors and pacemakers need guaranteed times. It would be better to use red-black search trees in these situations

Consideration: Java publishes hashing functions. Even with your own implementation, if someone learns your hashing function, they can maliciously send data that hashes to a single location to destroy performance.

**One-way hash functions (for security)**

“Hard” to find a key that will hash to a desired value (or for two keys that hash to the same value).  
For example, SHA-2, WHIRLPOOL, RIPEMD-160, etc

Good for things like passwords, digital fingerprints- but is far too expensive for ST implementations

Many improved versions

* Two-probe hashing: (separate chaining)
  + Hash to two positions and put key in shorter of the two chains
  + Expected length of the longest chain is log log N
* Double hashing
  + Use linear probing, but skip variable amount (not just 1 each time)
  + Effectively eliminates clustering
  + Can allow table to become nearly full
  + More difficult to implement delete
* Cuckoo hashing
  + Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur)
  + Constant worst case time for search

Hash tables vs balanced search trees

Hash tables:

* Simpler to code (if you don’t have to do hash function)
* No effective alternative to hashing for unordered keys
* Faster for simple keys (a few arithmetic ops versus log N compares)
* Better system support in Java for strings (e.g. cached hash code)

BSTs:

* Stronger performance guarantee
* Support for ordered ST operations
* Easier to implement compareTo() correctly than equals() and hashCode()