Chapter 5 – Hashing

Hashing –

- common operations: insertion, deletion, search **in constant time**

Some things we don't do with Hash table ADT: traversal, findMin, findMax, print in sorted order. Not an **ordered** structure.

Question – what kind of structure can guarantee **constant time** insertion? Deletion? Search?

Any of the ADTs we've studied so far?

Stack
Queue
Dequeue
Binary search tree

Hash table ADT allows us to access, insert or delete any element in **constant time**.

dictionary (logical data structure, an abstract data type) – a collection of data that supports operations:

- search
- insert,
- delete.

Any kind of data record can be stored in a dictionary. A key is used to identify data records. Sometimes referred to an associative container – as it associates (usually) unique keys to possibly unique values.

map (logical data structure, similar to dictionary) – unlike dictionary does not allow duplicate keys.

Hashing is an efficient way of implementing a dictionary – in fact, such a common way, people sometimes use the terms synonymously – Weiss appears to do that.

hashing – a storage implementation for fast access to data.

Involves an array that stores records or links to records – called a hash table

A **hash function**, hash, is applied to a key to get the storage location of the data; Must map a key into the array – that is, **hash(key)** must return an index of the hash table.

- Hash function should be cheap/efficient to apply
- Must distribute keys evenly across the table

Hashing is a technique for storing sparse data –

- huge range of keys – like student numbers, social security numbers
    - how many distinct student numbers?
- many fewer actual records expected
    - how many actual students?

Start here on day 2 (Spring 2013):

Recall – many keys, sparse data relative to number of keys. Table size established, usually of prime size. Hash function applied to key to generate index into table.

Key must be of a type for which equality can be checked.

Hash function:

- must be efficient to compute
- should provide a good (even) distribution of keys across the table

hash(key) must return same array index for many data records

Lots of research on what makes good hash functions….

Size of table is often set to be prime – seems to help in the even distribution of keys

See sample hash functions in Figures 5.2, 5.3, 5.4. Assume that the hash table size, *tableSize* = 10,007.

- Figure 5.2 – for the key "hello" what does hash(key) return?

The function sums the ascii values of the characters in "hello". That is, it computes 104 + 101 + 108 + 108 + 111 = 532 % *tableSize* = 532 % 10,007 = 532.

A key would have to have at least 78 digits to get a value bigger than 10007, and if we expect keys to be significantly smaller than that, we won't get a good distribution across the table. For example, if keys are limited to 8 characters are fewer, then the hash values generated will range only up to 1016, leaving most of the table unused.

- Figure 5.3 – for the key "hello" would compute: 104 + 27*101 + 729*108 = 81,563 % *tableSize* = 1507

While the number of different lower-case three-letter combinations is 26^3 = 2851, according to Weiss, the dictionary only shows that 2831 of them are used, so, at most, only 28% of the table will be used – again, not an even distribution of keys across the table.

- Figure 5.4 – for the key "hello" computes 104 *37^4+ 102*37^3 + 108*37^2 + 108*37 + 111 = ???? % *tableSize* = ????

This function uses all characters and will use all of the table, assuming that keys are of reasonable length. The function is efficient to carry-out, but could take too long IF the keys are very long.

Also, if keys are long, an overflow could occur – that is the number generated could get too big to fit into 32 bits and generate a negative value by overflowing into the sign bit. Thus, a check for a negative value is made and handled.

---
keys – or data records including keys, must provide an equals method so that the key value can be searched for

hashcode – in Java, a hashcode method (a hash function that returns an int for a key) is expected.

collision – occurs when more than one key hashes to the same hash address.

Can be expected even when the amount of data is less than the size of the hash table.

Collision resolution technique – process to follow to determine where to store data that hashes to an already used has address.

Assuming a good hash function, the probability of a collision grows as the table's load factor grows. Load factor = number of keys in table / tableSize

A load factor of 1.0 signifies a full table.
A table of size 10007, that contains 2600 keys has a load factor of 2600/10007 of .26

Two basic techniques for the physical data structure of a hash table, and the collision resolution technique(s) applicable are:

1. separate chaining – the hash table stores keys in linked lists; the data and keys for keys that hash to index x, are stored in a linked list that is accessible from table[x]. In effect, a hash table is an array of list references.
2. Open addressing – there are no lists

Implementation 1: Separate chaining – see picture in Figure 5.5, page 175.

For simplicity, a table size of 10 and hash function of key % 10 are used.

Separate chaining – keys that hash to same index are stored in a simple linked list – just add new key to the front of the list.

For this example, All keys that end in 1 hash into index 1, etc. If the keys are not random, that is if all end in the digit 1, then all will be hashed into the same address! (in which case, we'd choose a different hash function, but this is just an example).

If the chains get too long, then the tableSize has not been chosen well.
If there are too many collisions, with some buckets empty, then the hash function has not been chosen well.

See author's implementation for a chaining hash table, pages 175-178,

Note that typical key classes, like String, and Integer, both already implement:

hashCode and **equals.**

For any object-type that we want to store in a hash table, we need to

- Decide what the key is
- Provide an equals method for that object, that is based on the key value/field
- Provide a hashCode method for that object, that is based on the key value/field

Advantage of chained hash tables? Collision resolution simple
Disadvantage of changed hash tables? Use of second data structure, allocating new nodes adds to costs

Implementation 2: Open addressing – all data stored in the hash table; no chains (linked lists) stored. Recommendation: load fact (λ) < .5

Collision resolution – if a cell is already occupied, follow a pre-determined strategy for finding a new location.

Typically, the strategy is considered to be the application of a function such as f( ), which computes the next address as a distance from the original index. So, the following locations are tried h_0(key), h_1(key), etc. until an available location is found.

h_i(key) = hash(key) + f(i) % tableSize, where f(0) = 0

Strategies:

- Linear probing – just put the data item in the next available cell, wrapping around if the end of the table is reached. f(i) = i.
    - See the example on page 180.

Linear probing has the bad characteristic in leading to **primary clustering**. Lots of keys tend to hash to nearby locations.

- Quadratic probing – avoids primary clustering by using a slightly more complex strategy: $f(i) = i^2$. So the hash indices are:

hash(key), hash(key) + 1, hash(key)+4, hash(key) + 9, etc….

See example in Figure 5.13.

As with linear probing, the load factor should be .5 or less, for best results. But, it is possible that some indices will never be hashed to. To avoid this, make sure table size is a prime of the form $4k + 3$. Primes of this form include:

Primes of form 4i+3, for values i: 0..1000

i: 0; 3	i: 49; 199	i: 119; 479	i: 187; 751
i: 1; 7	i: 52; 211	i: 121; 487	i: 196; 787
i: 2; 11	i: 55; 223	i: 122; 491	i: 202; 811
i: 4; 19	i: 56; 227	i: 124; 499	i: 205; 823
i: 5; 23	i: 59; 239	i: 125; 503	i: 206; 827
i: 7; 31	i: 62; 251	i: 130; 523	i: 209; 839
i: 10; 43	i: 65; 263	i: 136; 547	i: 214; 859
i: 11; 47	i: 67; 271	i: 140; 563	i: 215; 863
i: 14; 59	i: 70; 283	i: 142; 571	i: 220; 883
i: 16; 67	i: 76; 307	i: 146; 587	i: 221; 887
i: 17; 71	i: 77; 311	i: 149; 599	i: 226; 907
i: 19; 79	i: 82; 331	i: 151; 607	i: 227; 911
i: 20; 83	i: 86; 347	i: 154; 619	i: 229; 919
i: 25; 103	i: 89; 359	i: 157; 631	i: 236; 947
i: 26; 107	i: 91; 367	i: 160; 643	i: 241; 967
i: 31; 127	i: 94; 379	i: 161; 647	i: 242; 971
i: 32; 131	i: 95; 383	i: 164; 659	i: 245; 983
i: 34; 139	i: 104; 419	i: 170; 683	i: 247; 991
i: 37; 151	i: 107; 431	i: 172; 691	i: 254; 1019
i: 40; 163	i: 109; 439	i: 179; 719	i: 257; 1031
i: 41; **167**	i: 110; 443	i: 181; 727	i: 259; 1039
i: 44; **179**	i: 115; 463	i: 184; 739	i: 262; 1051
i: 47; **191**	i: 116; 467	i: 185; 743	i: 265; 1063

i: 271; 1087
i: 272; 1091
i: 275; 1103
i: 280; 1123
i: 287; 1151
i: 290; 1163
i: 292; 1171
i: 296; 1187
i: 305; 1223
i: 307; 1231
i: 314; 1259
i: 319; 1279
i: 320; 1283
i: 322; 1291
i: 325; 1303
i: 326; 1307
i: 329; 1319
i: 331; 1327
i: 341; 1367
i: 349; 1399
i: 355; 1423
i: 356; 1427
i: 359; 1439
i: 361; 1447
i: 362; 1451
i: 364; 1459
i: 367; 1471
i: 370; 1483
i: 371; 1487
i: 374; 1499
i: 377; 1511
i: 380; 1523
i: 382; 1531
i: 385; 1543
i: 389; 1559
i: 391; 1567
i: 392; 1571
i: 394; 1579
i: 395; 1583
i: 401; 1607
i: 404; 1619
i: 406; 1627
i: 415; 1663
i: 416; 1667
i: 424; 1699
i: 430; 1723
i: 436; 1747
i: 439; 1759
i: 445; 1783
i: 446; 1787
i: 452; 1811
i: 455; 1823
i: 457; 1831
i: 461; 1847
i: 466; 1867
i: 467; 1871
i: 469; 1879
i: 476; 1907
i: 482; 1931
i: 487; 1951
i: 494; 1979
i: 496; 1987
i: 499; 1999
i: 500; 2003
i: 502; 2011
i: 506; 2027
i: 509; 2039
i: 515; 2063
i: 520; 2083
i: 521; 2087
i: 524; 2099
i: 527; 2111
i: 532; 2131
i: 535; 2143
i: 544; 2179
i: 550; 2203
i: 551; 2207
i: 559; 2239
i: 560; 2243
i: 562; 2251
i: 566; 2267
i: 571; 2287
i: 577; 2311
i: 584; 2339
i: 586; 2347
i: 587; 2351
i: 592; 2371
i: 595; 2383
i: 599; 2399
i: 602; 2411
i: 605; 2423
i: 611; 2447
i: 614; 2459
i: 616; 2467
i: 625; 2503
i: 632; 2531
i: 634; 2539
i: 635; 2543
i: 637; 2551
i: 644; 2579
i: 647; 2591
i: 661; 2647
i: 664; 2659
i: 665; 2663
i: 667; 2671
i: 670; 2683
i: 671; 2687
i: 674; 2699
i: 676; 2707
i: 677; 2711
i: 679; 2719
i: 682; 2731
i: 691; 2767
i: 697; 2791
i: 700; 2803
i: 704; 2819
i: 710; 2843
i: 712; 2851
i: 719; 2879
i: 721; 2887
i: 725; 2903
i: 731; 2927
i: 734; 2939
i: 740; 2963
i: 742; 2971
i: 749; 2999
i: 752; 3011
i: 754; 3019
i: 755; 3023
i: 766; 3067
i: 769; 3079
i: 770; 3083
i: 779; 3119
i: 790; 3163
i: 791; 3167
i: 796; 3187
i: 797; 3191
i: 800; 3203
i: 812; 3251
i: 814; 3259
i: 817; 3271
i: 824; 3299
i: 826; 3307
i: 829; 3319
i: 830; 3323
i: 832; 3331
i: 835; 3343
i: 836; 3347
i: 839; 3359
i: 842; 3371
i: 847; 3391
i: 851; 3407
i: 865; 3463
i: 866; 3467
i: 872; 3491
i: 874; 3499

i: 877; 3511	i: 905; 3623	i: 934; 3739	i: 976; 3907
i: 881; 3527	i: 907; 3631	i: 941; 3767	i: 977; 3911
i: 884; 3539	i: 910; 3643	i: 944; 3779	i: 979; 3919
i: 886; 3547	i: 914; 3659	i: 950; 3803	i: 980; 3923
i: 889; 3559	i: 917; 3671	i: 955; 3823	i: 982; 3931
i: 892; 3571	i: 922; 3691	i: 961; 3847	i: 985; 3943
i: 895; 3583	i: 929; 3719	i: 962; 3851	i: 986; 3947
i: 901; 3607	i: 931; 3727	i: 965; 3863	i: 991; 3967

Primes of form 4i + 3 for i: 9000…9999

i: 9001; 36007	i: 9160; 36643	i: 9310; 37243	i: 9469; 37879
i: 9002; 36011	i: 9167; 36671	i: 9326; 37307	i: 9476; 37907
i: 9016; 36067	i: 9170; 36683	i: 9334; 37339	i: 9487; 37951
i: 9020; 36083	i: 9172; 36691	i: 9340; 37363	i: 9490; 37963
i: 9026; 36107	i: 9184; 36739	i: 9344; 37379	i: 9491; 37967
i: 9032; 36131	i: 9191; 36767	i: 9355; 37423	i: 9496; 37987
i: 9037; 36151	i: 9194; 36779	i: 9361; 37447	i: 9497; 37991
i: 9046; 36187	i: 9196; 36787	i: 9365; 37463	i: 9502; 38011
i: 9047; 36191	i: 9197; 36791	i: 9370; 37483	i: 9509; 38039
i: 9062; 36251	i: 9211; 36847	i: 9376; 37507	i: 9511; 38047
i: 9065; 36263	i: 9217; 36871	i: 9377; 37511	i: 9520; 38083
i: 9074; 36299	i: 9221; 36887	i: 9386; 37547	i: 9529; 38119
i: 9076; 36307	i: 9224; 36899	i: 9391; 37567	i: 9541; 38167
i: 9079; 36319	i: 9229; 36919	i: 9392; 37571	i: 9545; 38183
i: 9085; 36343	i: 9230; 36923	i: 9394; 37579	i: 9554; 38219
i: 9095; 36383	i: 9232; 36931	i: 9397; 37591	i: 9557; 38231
i: 9112; 36451	i: 9235; 36943	i: 9401; 37607	i: 9559; 38239
i: 9116; 36467	i: 9236; 36947	i: 9404; 37619	i: 9571; 38287
i: 9119; 36479	i: 9244; 36979	i: 9410; 37643	i: 9574; 38299
i: 9130; 36523	i: 9250; 37003	i: 9415; 37663	i: 9575; 38303
i: 9131; 36527	i: 9254; 37019	i: 9422; 37691	i: 9581; 38327
i: 9137; 36551	i: 9259; 37039	i: 9424; 37699	i: 9587; 38351
i: 9139; 36559	i: 9271; 37087	i: 9436; 37747	i: 9592; 38371
i: 9140; 36563	i: 9280; 37123	i: 9445; 37783	i: 9607; 38431
i: 9142; 36571	i: 9284; 37139	i: 9449; 37799	i: 9611; 38447
i: 9145; 36583	i: 9289; 37159	i: 9452; 37811	i: 9614; 38459
i: 9146; 36587	i: 9292; 37171	i: 9457; 37831	i: 9635; 38543
i: 9149; 36599	i: 9299; 37199	i: 9461; 37847	i: 9641; 38567
i: 9151; 36607	i: 9305; 37223	i: 9467; 37871	i: 9650; 38603

i: 9652; 38611
i: 9659; 38639
i: 9662; 38651
i: 9667; 38671
i: 9674; 38699
i: 9676; 38707
i: 9677; 38711
i: 9680; 38723
i: 9686; 38747
i: 9691; 38767
i: 9695; 38783
i: 9697; 38791
i: 9700; 38803
i: 9709; 38839
i: 9712; 38851
i: 9716; 38867
i: 9722; 38891
i: 9725; 38903
i: 9730; 38923
i: 9739; 38959
i: 9742; 38971
i: 9754; 39019
i: 9755; 39023
i: 9760; 39043
i: 9761; 39047
i: 9769; 39079
i: 9775; 39103
i: 9776; 39107
i: 9779; 39119
i: 9784; 39139
i: 9790; 39163
i: 9797; 39191
i: 9799; 39199
i: 9806; 39227
i: 9809; 39239
i: 9812; 39251
i: 9830; 39323
i: 9835; 39343
i: 9839; 39359
i: 9841; 39367
i: 9842; 39371
i: 9845; 39383
i: 9854; 39419
i: 9859; 39439
i: 9860; 39443
i: 9862; 39451
i: 9874; 39499
i: 9875; 39503
i: 9877; 39511
i: 9887; 39551
i: 9890; 39563
i: 9901; 39607
i: 9904; 39619
i: 9905; 39623
i: 9907; 39631
i: 9914; 39659
i: 9916; 39667
i: 9917; 39671
i: 9919; 39679
i: 9925; 39703
i: 9929; 39719
i: 9931; 39727
i: 9944; 39779
i: 9947; 39791
i: 9949; 39799
i: 9956; 39827
i: 9959; 39839
i: 9961; 39847
i: 9965; 39863
i: 9970; 39883
i: 9971; 39887
i: 9992; 39971
i: 9994; 39979
i: 9995; 39983

There are infinitely many primes of this form – so not hard to find one of the right size for your data.

- Double hashing – uses a second hash function and we probe the following: f(i) = i*hash2(key) or if hash1(key) is in use, then try hash2(key), 2*hash2(key), 3*hash2(key), … etc.

Must make sure that hash2(key) never generates 0.
Must make sure that all indices can be probed: Weiss suggests a function of the form hash2(key) = R – (key % R) with R a prime smaller than table size. For his example, Weiss chooses R = 7 and shows results in Figure 5.18.

Other techniques:

Rehashing – applied when the table gets too full (load factor > some threshold value).

- Solution – double the tablesize (and pick the next prime for size) and then rehash (hash function modified for new table size) all values into the new table.
    - Rehashing is costly as it involves walking through entire table and rehashing all.
    - Rehashing can also be used with separate chaining, when hash table chains are getting too long, or table too full.

HashMap – is a Java collections framework implementation of a Map ADT. Makes a good choice for a solution that requires a hash table.

Hash tables with worst-case O(1) access

Perfect hashing – a *perfect hash function* is one that maps n iems to n unique indices; no collisions occur. So, how do you come up with a perfect hash function?

- If you know, in advance, all N keys, you could set up a chained hash table with N lists, of course that might result in a very big table and we might still get collsions.
- Research has shown that if the table size M is $\Omega(N^2)$ we can devise a plan that guarantees that the probability of collisions is at least ½ (more than half the time there is no collision). However, this makes for a very big table.
- One option is to store exactly N bins – of course, there may still be collisions, but since all keys are known, we can determine how many keys hash to each bin, and resolve collisions by storing an additional table for that bin, where the secondary table is of size k^2 if k elements map to that bin. A hash function for the bin is determined such that the k elements are hashed in a way that no collision occurs. Each bin may have its own hash function. It turns out that the space needed to make this plan work is linear!

Cuckoo hashing – for N items, we maintain 2 tables, each more than half empty, and we have two independent hash functions that can assign each item to a position in each table. An item is always stored in one of these two places.

From Wikipedia: The name derives from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table.

When a key hashes to a location already used; the key already there is moved to its correct position in table 2, and the new key gets its spot. A key always hashes to the spot in table 1, determined by the hash function for table 1. It is only moved into table 2, if another key pushes it out of its position.

It is possible that a cycle could occur and an element might be able to be placed in the table – example in book.

If each hash table has s load factor less than .5, cuckoo hashing works pretty well. When the table's load factor is .5 or higher, the probability of cycles increases.

Universal hashing –

Goals of good hash function:

1. Efficiently computable (in constant time)
2. Distributes its items uniformly across the table

Universal hash functions – allow us to choose the hash function randomly in a way that condition 2 is satisfied.

Let M represent the tablesize.

A family H of hash functions is universal, if for any $x \neq y$, the number of hash functions h in H for with $h(x) = h(y)$ is at most $|H|/M$.

One such H, where a & b are chosen randomly and p is a prime number bigger than the tablesize M:

$H = \{H_{a,b}(x) = ((ax + b) \bmod p) \bmod M)$, where $1 \leq a \leq p-1$, $0 \leq b \leq p-1\}$.