Lecture 16

- Hashing
- Hash table and hash function design
- Hash functions for integers and strings
- Collision resolution strategies: linear probing, double hashing, random hashing, separate chaining
- Hash table cost functions
- Map ADT

Reading: Weiss Ch. 5

Finding data fast

- You know that searching for a key in an unsorted array or a linked list has time cost O(N) average case
- You know that searching for a key in a sorted array or a balanced search tree or a randomized search tree or a skip list has time cost O(logN) average case
- Is it possible to do better?...
- Consider searching for a key `k` in an array `a`. If we somehow knew that, if key `k` is in the array, it would be at index `i`, we could do a find on `k` with one array access: `a[i]`
  - This would be O(1), ignoring any memory hierarchy effects
- Of course it may be unreasonable to just "know" the index for key `k`. But suppose we could *compute* the index for `k` in time O(1). Then we could still do a search with time cost O(1)!
- *Hashing* and *hash tables* are ways of trying to make this simple idea a reality
  - ... and it turns out that with good design it is definitely possible to get O(1) average case find, insert, and delete time costs, and good space costs too

Hashing

- Basic idea:
  - store key or key/data record pairs in an array, called a "hash table"
  - make the hash table size comparable to the number of actual keys that will be stored in it --- typically *much* smaller than the number of possible key values
  - given a key, compute the index of that key’s location in the array using a fast O(1) function, called a "hash function"
- Since the hash function is O(1), a hash table has the potential for very fast find performance (the best possible!), but...
- ... since the hash function is mapping from a large set (the set of all possible keys) to a smaller set (the set of hash table locations) there is the possiblity of *collisions*: two different keys wanting to be at the same table location

Probability of collisions

- Suppose you have a hash table with M slots or buckets, and you have N keys to randomly insert into it
- What is the probability that there will be a collision among these keys?
- You might think that as long as the table is less than half full, there is less than 50% chance of a collision, but this is not true
- The probability of at least one collision among N random independently inserted keys is

$$P_{N,M}(collision) = 1 - P_{N,M}(no\ collision)$$

$$= 1 - \prod_{i=1}^{N} P_{N,M}(i^{th} key\ no\ collision)$$

- The probability that the i^{th} key will not collide with any of the i-1 keys already in the table is just the probability that it will land in one of the M-i-1 available empty locations. If all locations are equally likely, then the probability of at least one collision when inserting N keys in a table of size M is:

$$P_{N,M}(collision) = 1 - (1) \cdot \frac{M-1}{M} \cdot \frac{M-2}{M} \cdot \ldots \cdot \frac{M-N+1}{M}$$

Average total number of collisions

- Another way to look at the situation is this...
- You are throwing N balls randomly into M containers
- The first ball lands in some container. The remaining N-1 balls each have probability 1/M of landing in the same container; so the average number of collisions with the first ball will be (N-1)/M
- The second ball lands in some container. The reminaing N-2 balls each have probability 1/M of landing in the same container; so the average number of total with the second ball will be (N-2)/M
- Etc... So the average *total* number of collisions is

$$\sum_{i=1}^{N-1} \frac{i}{M} = \frac{N(N-1)}{2M}$$

- And so the expected (average) number of collisions will be 1 when:

$$\frac{N(N-1)}{2M} = 1$$

which for large M, implies $N \approx \sqrt{2}\sqrt{M}$

Hashtable collisions and the "birthday paradox"

- Suppose there are 365 slots in the hash table: M=365
- What is the probability that there will be a collision when inserting N keys?
  - For N = 10, $\text{prob}_{N,M}(\text{collision}) = 12\%$
  - For N = 20, $\text{prob}_{N,M}(\text{collision}) = 41\%$
  - For N = 30, $\text{prob}_{N,M}(\text{collision}) = 71\%$
  - For N = 40, $\text{prob}_{N,M}(\text{collision}) = 89\%$
  - For N = 50, $\text{prob}_{N,M}(\text{collision}) = 97\%$
  - For N = 60, $\text{prob}_{N,M}(\text{collision}) = 99+\%$
- So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday
- And, on average one pair of people will share a birthday in a group of about $\sqrt{2 \cdot 365} \approx 27$ people
- In general: collisions are likely to happen, unless the hash table is quite sparsely filled
- So, if you want to use hashing, can't use perfect hashing because you don't know the keys in advance, and don't want to waste huge amounts of storage space, you have to have a strategy for dealing with collisions

The birthday collision "paradox"

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Jan																															
Feb																															
Mar																															
Apr																															
May																															
Jun																															
Jul																															
Aug																															
Sep																															
Oct																															
Nov																															
Dec																															

Making hashing work

- The main issues in implementing hashing are:
  - Deciding on the size of the hash table
  - Deciding on the hash function
  - Deciding on the collision resolution strategy

- With a good hashtable design, O(1) average-case insert and find operation time costs can be achieved, with O(1) space cost per key stored

- This makes hashtables a very useful and very commonly used data structure

Hash table size

- By "size" of the hash table we mean how many slots or buckets it has
- Choice of hash table size depends in part on choice of hash function, and collision resolution strategy
- But a good general “rule of thumb” is:
  - The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and
  - Size of hash table array should be a prime number
- So, let M = the next prime larger than 1.3 times the number of keys you will want to store in the table, and create the table as an array of length M
- (If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space)

Hash functions: desiderata

- A hash function should be very fast to compute
- A hash function should also distribute keys as uniformly as possible in the hash table, to avoid collisions as much as possible
  - For example, you don’t want a hash function that will map the set of keys to only a subset of the locations in the table!
  - Ideally: all the bits in the hash function should depend on each bit in the key
- A hash function should be consistent with the equality testing function
  - If two keys are equal, the hash function should map them to the same table location
  - Otherwise, the fundamental hash table operations will not work correctly
- A good choice of hash function can depend on the type of keys, the distribution of key insert requests, and the size of the table; and it is often difficult to design a good one
- (If you know the actual keys in advance, you can construct a "perfect" hash function, which never has any collisions; this has some uses, but obviously it’s more usual not to know the keys in advance)
- We will look at some commonly used hash functions for some different types of keys

Hash functions for integers: H(K) = K mod M

- For general integer keys and a table of size M, a prime number:
  - a good fast general purpose hash function   is H(K) = K mod M

- If table size is not a prime, this may not work well for some key distributions
  - for example, suppose table size is an even number; and keys happen to be all even, or all odd. Then only half the table locations will be hit by this hash function

- So, use prime-sized tables with H(K) = K mod M

Hash functions for integers: random functions

- A hash function tries to distribute keys "randomly" over table locations
- For typical integer keys K, with prime table size M, hash function K mod M usually does a good job of this
- But with any hash function, it is possible to have "bad" behavior, where most all keys the user happens to want to insert in the hash table hash to the same location
- To make this very unlikely to happen twice in a row, the hash function can be a pseudorandom number generator whose output depends on a "random" seed decided when the table is created, *and* on the key value
    - this is called "random hashing", which we will discuss in a moment
    - Random hashing is mostly of theoretical interest; it is not used much in practice because other, simpler hash functions give results that are as good

Hash functions for strings

- It is common to want to use string-valued keys in hash tables
- What is a good hash function for strings?
- The basic approach is to use the characters in the string to compute an integer, and then take the integer mod the size of the table
- How to compute an integer from a string?
  - You could just take the last two 16-bit chars of the string and form a 32-bit int
  - But then all strings ending in the same 2 chars would hash to the same location; this could be very bad
  - It would be better to have the hash function depend on *all* the chars in the string
- There is no recognized single "best" hash function for strings. Let's look at some possibile ones

String hash function #1

- This hash function adds up the integer values of the chars in the string (then need to take the result mod the size of the table):

```
int hash(std::string const & key) {
   int hashVal = 0, len = key.length();
   for(int i=0; i<len; i++) {
      hashVal += key[i];
   }
   return hashVal;
}
```

- This function is simple to compute, but it often doesn't work very well in practice:
- Suppose the keys are strings of 8 ASCII capital letters and spaces
- There are 27^8 possible keys; however, ASCII codes for these characters are in the range 65-95, and so the sums of 8 char values will be in the range 520 - 760
- In a large table (M>1000), only a small fraction of the slots would ever be mapped to by this hash function!   For a small table (M<100), it may be okay

String hash function #2

- A more effective approach is to compute a polynomial whose coefficients are the integer values of the chars in the String
- For example, for a String s with length n+1, we might compute a polynomial in *x*

$$H(s) = \sum_{i=0}^{n} \text{s.charAt}(i) \cdot x^i$$

  ... and take the result mod the size of the table.

- Horner's rule can be used to compute *H*(s) with n+1 iterations:

$$H_0(s) = \text{s.charAt}(n)$$

$$H_1(s) = x \cdot H_0(s) + \text{s.charAt}(n-1)$$

$$H_2(s) = x \cdot H_1(s) + \text{s.charAt}(n-2)$$

- ... etc. It is interesting to compare this to the basic form of a linear congruential random number generator; there is a relation between good hashing and apparent randomness:

$$R_0 = \text{seed}$$

$$R_1 = A \cdot R_0 + B$$

$$R_2 = A \cdot R_1 + B$$

String hash function #2: Java code

- java.lang.String’s hashCode() method uses that polynomial with *x*=31 (though it goes through the String’s chars in reverse order), and uses Horner’s rule to compute it:

```
class String implements java.io.Serializable, Comparable {
  /** The value is used for character storage. */
  private char value[];
  /** The offset is the first index of the storage that is used. */
  private int offset;
  /** The count is the number of characters in the String. */
  private int count;
  public int hashCode() {
      int h = 0;
      int off = offset;
      char val[] = value;
      int len = count;

      for (int i = 0; i < len; i++)
       h = 31*h + val[off++];

      return h;
  }
```

String hash function #3

- Here is a string hash function that uses Horner’s rule to compute a polynomial in *x*=16, using all the characters in the null-terminated string argument, plus doing some additional manipulations:

```
long hash(char* key) {
   long hashVal = 0;
   while (*key != '/0') {
      hashVal = (hashVal << 4) + *(key++);
      long g = hashval & 0xF0000000L;
      if (g != 0) hashVal ^= g >>> 24;
      hashVal &= ~g;
   }
   return hashVal;
}
```

- This function is used in some C++ object file linking code
- It apparently works well in practice, but its design is obscure!

Hash functions in other contexts

- A hash function for a hash table takes a key, and returns an index into the table
  - the set of possible keys is large, but the set of hash function values is just the size of the table
- Other uses for hash functions include password systems, message digest systems, digital signature systems
- For these applications to work as intended, the probability of collisions must be very low, and so you need a hash function with a very large set of possible values
- Password system: Given a user password, the operating system computes its hash, and compares it to the hash for that user stored in a file. (Don't want it to be easy to guess a password that hashes to that same value)
- Message digest system: Given an important message, compute its hash, and publish it separately from the message itself. A reader who wants to check the validity of the message also computes its hash using the same algorithm, and compares to the published hash. (Don't want it to be easy to forge a message and still get the same hash)
- Examples of popular hash function algorithms for these applications:
  - **md5**: 128 bit values (2^{128} possible values: try $\approx 2^{64}$ keys before finding a collision)
  - **sha-1**: 160 bit values (2^{160} possible values: try $\approx 2^{80}$ keys before finding a collision)

Using a hash function

- using the hash function H(K) = K mod M, insert these integer keys:

  701, 145, 217, 19

  in this table:

index:	0	1	2	3	4	5	6

- Now, what to do about collisions?

Collision resolution strategies

- Unless you are doing "perfect hashing" you have to have a collision resolution strategy, to deal with collisions in the table.
- The strategy has to permit find, insert, and delete operations that work correctly!
- Collision resolution strategies we will look at are:
    - Linear probing
    - Double hashing
    - Random hashing
    - Separate chaining

Linear probing: inserting a key

- When inserting a key K in a table of size M, with hash function H(K)

  1. Set indx = H(K)
  2. If table location indx already contains the key, no need to insert it. Done!
  3. Else if table location indx is empty, insert key there. Done!
  4. Else collision. Set indx = (indx + 1) mod M.
  5. If indx == H(K), table is full! (Throw an exception, or enlarge table.) Else go to 2.

- So, linear probing basically does a linear search for an empty slot when there is a collision

- Advantages: easy to implement; always finds a location if there is one; very good average-case performance when the table is not very full

- Disadvantages: “clusters” or "clumps" of keys form in adjacent slots in the table; when these clusters fill most of the array, performance deteriorates badly as the probe sequence performs what is essentially an exhaustive search of most of the array

Linear probing, an example

- M = 7, H(K) = K mod M
  insert these keys 701, 145, 217, 19, 13, 749

  in this table, using linear probing:

index:	0	1	2	3	4	5	6

Linear probing: searching for a key

- If keys are inserted in the table using linear probing, linear probing will find them!

- When searching for a key K in a table of size N, with hash function H(K) :

  1. Set indx = H(K)
  2. If table location indx contains the key, return FOUND.
  3. Else if table location indx is empty, return NOT FOUND.
  4. Else set indx = (indx + 1) mod M.
  5. If indx == H(K), return NOT FOUND. Else go to 2.

- Question: How to delete a key from a table that is using linear probing?
  - Could you do "lazy deletion", and just mark the deleted key's slot as empty? Why or why not?

Double hashing

- Linear probing collision resolution leads to clusters in the table, because if two keys collide, the next position probed will be the same for both of them.

- The idea of double hashing: Make the offset to the next position probed depend on the key value, so it can be different for different keys
  - Need to introduce a second hash function $H_2(K)$, which is used as the offset in the probe sequence (think of linear probing as double hashing with $H_2(K) == 1$)
  - For a hash table of size M, $H_2(K)$ should have values in the range 1 through M-1; if M is prime, one common choice is $H2(K) = 1 + ((K/M) \bmod (M-1))$

- The insert algorithm for double hashing is then:

```
1. Set indx = H(K); offset = H_2(K)
2. If table location indx already contains the key, no need to insert it.  Done!
3. Else if table location indx is empty, insert key there.  Done!
4. Else collision.  Set indx = (indx + offset) mod M.
5. If indx == H(K), table is full!  (Throw an exception, or enlarge table.)  Else go to 2.
```

- With prime table size, double hashing works very well in practice

Random hashing

- As with double hashing, random hashing avoids clustering by making the probe sequence depend on the key
- With random hashing, the probe sequence is generated by the output of a pseudorandom number generator seeded by the key (possibly together with another seed component that is the same for every key, but is different for different tables)
- The insert algorithm for random hashing is then:

```
1. Create RNG seeded with K.   Set indx = RNG.next() mod M.
2. If table location indx already contains the key, no need to insert it.  Done!
3. Else if table location indx is empty, insert key there.  Done!
4. Else collision.  Set indx = RNG.next() mod M.
5. If all M locations have been probed, give up.  Else, go to 2.
```

- Random hashing is easy to analyze, but because of the "expense" of random number generation, it is not often used; double hashing works about as well

Open addressing vs. separate chaining

- Linear probing, double and random hashing are appropriate if the keys are kept as entries in the hashtable itself...
  - doing that is called "open addressing"
  - it is also called "closed hashing"
- Another idea: Entries in the hashtable are just pointers to the head of a linked list ("chain"); elements of the linked list contain the keys...
  - this is called "separate chaining"
  - it is also called "open hashing"
- Collision resolution becomes easy with separate chaining: just insert a key in its linked list if it is not already there
  - (It is possible to use fancier data structures than linked lists for this; but linked lists work very well in the average case, as we will see)
- Let's look at analyzing time costs of these strategies

Analysis of open-addressing hashing

- A useful parameter when analyzing hash table Find or Insert performance is the *load factor*

  $$\alpha = N/M$$

  where M is the size of the table, and
  N is the number of keys that have been inserted in the table
- The load factor is a measure of how full the table is

- Given a load factor α, we would like to know the time costs, in the best, average, and worst case of
  - new-key insert and unsuccessful find (these are the same)
  - successful find

- The best case is O(1) and worst case is O(N) for all of these... so let's analyze the average case

- We will give results for random hashing and linear probing. In practice, double hashing is similar to random hashing

Average case unsuccessful find / insertion cost

- Assume a table with load factor α = N/M. Consider random hashing, so clustering is not a problem; each probe location is generated randomly, and independently
- With each probe, the probability of finding an empty location is (1-α). Finding an empty location stops the find or insertion
- This is a Bernoulli process, with probability of success (1-α). The expected first-order interarrival time of such a process is 1/(1-α). So:
- The average number of probes for insert or unsuccessful find with random hashing is

$$U_\alpha = \frac{1}{1-\alpha}$$

- With linear probing, probe locations are not independent; clusters form, which leads to long probe sequences when load factor is high. It can be shown that the average number of probes for insert or unsuccessful find with linear probing is approximately

$$U_\alpha \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

- These average case time costs are bad, bounded only by M, when α is close to 1; but are not too bad (4 and 8.5 respectively) when α is .75 or less, independent of M

Average case successful find cost

- Assume a table with load factor $\alpha = N/M$. Consider random hashing, so clustering is not a problem; each probe location is generated randomly, and independently
- For a key in the table, the number of probes required to successfully find it is equal to the number of probes taken when it was inserted in the table. The insertion of each new key increases the load factor, starting from 0 and going to α.
- Therefore, the average number of probes for successful find with random hashing is

$$S_\alpha = \frac{1}{N}\sum_{i=0}^{N-1}\frac{1}{1-i/M} \approx \frac{1}{\alpha}\int_0^\alpha \frac{1}{1-x}dx = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

- With linear probing, clusters form, which leads to longer probe sequences. It can be shown that the average number of probes for successful find with linear probing is

$$S_\alpha \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

- These average case time costs are bad, bounded only by M, when α is close to 1; but are good (1.8 and 2.5 respectively) when α is .75 or less, independent of M

Separate chaining: basic algorithms

- When inserting a key K in a table with hash function H(K)

  1. Set indx = H(K)
  2. Insert key in linked list headed at indx. (Search the list first to avoid duplicates.)

- When searching for a key K in a table with hash function H(K)

  1. Set indx = H(K)
  2. Search for key in linked list headed at indx, using linear search.

- When deleting a key K in a table with hash function H(K)

  1. Set indx = H(K)
  2. Delete key in linked list headed at indx

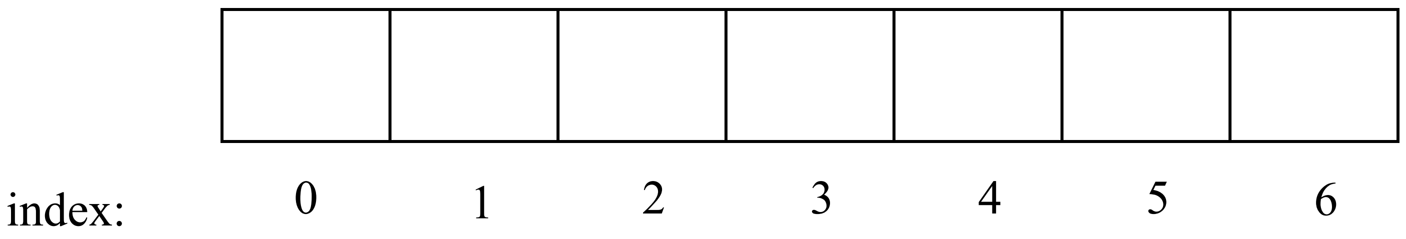
- Advantages: average case performance stays good as number of entries approaches and even exceeds M; delete is easier to implement than with open addressing
- Disadvantages: requires dynamic data, requires storage for pointers in addition to data, can have poor locality which causes poor caching performance

Separate chaining, an example

M = 7, H(K) = K mod M

insert these keys 701, 145, 217, 19, 13, 749

in this table, using separate chaining:



Analysis of separate-chaining hashing

- Keep in mind the *load factor* measure of how full the table is:

  $$\alpha = N/M$$

  where M is the size of the table, and
  N is the number of keys that have been inserted in the table

- With separate chaining, it is possible to have $\alpha > 1$

- Given a load factor α, we would like to know the time costs, in the best, average, and worst case of
  - new-key insert and unsuccessful find (these are the same)
  - successful find

- The best case is O(1) and worst case is O(N) for all of these... let's analyze the average case

Average case costs with separate chaining

- Assume a table with load factor $\alpha = N/M$
- There are N items total distributed over M linked lists (some of which may be empty), so the average number of items per linked list is:
- In unsuccessful find/insert, one of the linked lists in the table must be exhaustively searched, and the average length of a linked list in the table is α. So the average number of comparisons for insert or unsuccessful find with separate chaining is

$$U_\alpha = \alpha$$

- In successful find, the linked list containing the target key will be searched. There are on average (N-1)/M keys in that list besides the target key; on average half of them will be searched before finding the target. So the average number of comparisons for successful find with separate chaining is

$$S_\alpha = 1 + \frac{N-1}{2M} \approx 1 + \frac{\alpha}{2}$$

- These are less than 1 and 1.5 respectively, when $\alpha < 1$
- And these remain O(1), independent of N, even when α exceeds 1.

Map Abstract Data Type

- A data structure is intended to hold data
    - An insert operation inserts a data item into the structure; a find operation says whether a data item is in the structure; delete removes a data item; etc.
- A Map is a specialized kind of data structure:
    - A Map structure is intended to hold *pairs*: each pair consists of a key, together with some related data associated with that key
    - An insert operation inserts a key-data pair in the table; a find operation takes a key and returns the data in the key-data pair with that key; delete takes a key and removes the key-data pair with that key; etc.
- The Map data type is also known as “Table” or “Dictionary”
- Instances of this type are sometimes called “associative arrays” or “associative memories”

Map as ADT

- Domain:
  - a collection of pairs; each pair consists of a key, and some additional data

- Operations (typical):
  - Create a table (initially empty)
  - Insert a new key-data pair in the table; if a key-data pair with the same key is already there, update the data part of the pair
  - Find the key-data pair in the table corresponding to a given key; return the data
  - Delete the key-data pair corresponding to a given key
  - Enumerate (traverse) all key-data pairs in the table

Implementing the Map ADT

- A Map (also known as Table or Dictionary) can be implemented in various ways:
  - using a list, binary search tree, hashtable, etc., etc.

- In each case:
  - the implementing data structure has to be able to hold key-data pairs
  - the implementing data structure has to be able to do insert, find, and delete operations paying attention to the key

- The Java Collections Framework has **`TreeMap<K,V>`** (using a red-black tree) and **`HashMap<K,V>`** (using a separate chaining hashtable) implementations of the **`Map<K,V>`** interface

- The C++ STL has the **`std::map<Key,T>`** container (using a red-black tree). Current version of STL has no hashing container, though it has been proposed

Hashtables vs. balanced search trees

- Hashtables and balanced search trees can both be used in applications that need fast insert and find

- What are advantages and disadvantages of each?

  - Balanced search trees guarantee worst-case performance O(log N), which is quite good
  - A well-designed hash table has typical performance O(1), which is excellent; but worst-case is O(N), which is bad

  - Search trees require that keys be well-ordered: For any keys K1, K2, either K1<K2, K1==K2, or K1> K2
    - So, there needs to be an ordering function that can be applied to the keys
  - Hashtables only require that keys be testable for equality, and that a hash function be computable for them
    - So, there need to be an equality test function and a hash function that can be applied to the keys

Hashtables vs. balanced search trees, cont’d

- A search tree can easily be used to return keys close in value to a given key, or to return the smallest key in the tree, or to output the keys in sorted order
- A hashtable does not normally deal with ordering information efficiently

- In a balanced search tree, delete is as efficient as insert
- In a hashtable that uses open addressing, delete can be inefficient, and somewhat tricky to implement (easy with separate chaining though)

- Overall, balanced search trees are rather difficult to implement correctly
- Hash tables are relatively easy to implement, though they depend on a good hash function for good performance

Next time

- Simple vs. more sophisticated algorithm cost analysis
- The cost of accessing memory
- B-trees
- B-tree performance analysis
- B-tree find, insert, and delete operations
- B-tree example: a 2-3 tree

Reading: Weiss, Ch. 4 section 7