Computational Optimization
ISE 407

Lecture 16

Dr. Ted Ralphs

References for Today's Lecture

- References
  - CLRS Section 11.1, Chapter 12
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.
  - https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633#145633
  - https://sites.google.com/site/murmurhash/discussion

Symbol Tables and Dictionaries

- We have now discussed in detail the ADT for a list and have seen two implementations, as well as other data structures built on lists.
- We now consider a different kind of list data structure that supports different kinds of operations.
- A *symbol table* is a data structure for storing a list of items, each with a key and satellite data.
- This data structure supports the following basic operations.
  - Construct a symbol table.
  - Search for an item (or items) having a specified key.
  - Insert an item.
  - Remove a specified item.
  - Count the number of items.
  - Print the list of items.
- Symbol tables are also called *dictionaries* because of the obvious comparison with looking up entries in a dictionary.
- Note that the keys may not have an ordering.

Additional Operations on Symbol Tables

- If the items can be ordered, e.g., by `__lt__` and `__eq__`, we may support the following additional operations.
  - Sort the items (print them in sorted order).
  - Return the maximum or minimum item.
  - Select the k^{th} item.
  - Return the successor or predecessor of a given item.
- We may also want to be able to join two symbol tables into one.
- These operations may or may not be supported in various implementations.

Dictionary ADT

```
class Dictionary:
    def __init__()
    def getNumItems()
    def __contains__(key)
    def get(key)
    def append(key, value)
    def remove(key)
    def keys()
    def values()
    def select(k)
    def sort()
```

Naive Implementation

- Consider a list of items whose keys are small positive integers.
- Assuming no duplicate keys, we can implement such a symbol table using a list.

```
class Dictionary:
    def __init__(self, maxKey):
        self.list = [None for i in range(maxKey)]
        self.maxKey = maxKey
    def append(self, key, value):
        self.list[key] = value
    def remove(key):
        self.list[key] = None
    def __contains__(key)
        if self.list[key] = None:
            return False
        else:
            return True
```

Naive Implementation (cont.)

```
    def select(self, key):
        for i in self.list:
            if i != None:
                key--
                if key == 0:
                    return self.list[i]

    def sort(self):
        pass

    def getNumItems(self):
        count = 0
        for i in self.list
        if i != None:
             count++
        return count
```

Arbitrary Keys

- Note that with arrays, most operations are constant time.
- What if the keys are not integers or have arbitrary value?
- We could still use an array or a linear linked list to store the items.
- However, some of the operations would become inefficient.
  - If we keep the items in order, searching would be efficient (binary search), but inserting would be inefficient.
  - If we kept the items unordered, inserting would be efficient, but searching would be inefficient (sequential search).
- A *hash table* is a more efficient data structure for implementing symbol tables where the keys are an arbitrary data type.
- Hash tables are the subject of the next lecture.

Hash Tables

- We now consider data structure for storing a dictionary that support only the operations
  - insert,
  - delete, and
  - search.
- The most obvious data structures for storing dictionaries depend on using comparison and exchange to order the items.
- This limits the efficiency of certain operations.
- A *hash table* is a generalization of an array that takes advantage of our ability to access an arbitrary array element in constant time.
- Using hashing, we determine where to store an item in the table (and how to find it later) without using comparison.
- This allows us to perform all the basic operations very efficiently.

Addressing using Hashing

- Recall the array-based implementation of a dictionary from earlier.
- In this implementation, we allocated one memory location for each possible key.
- How can we extend this method to the case where the set U of possible keys is extremely large?
- Answer: Use *hashing*.
- A *hash function* is a function $h : U \to 0, \ldots, M-1$ that takes a key and converts it into an array index (called the *hash value*).
- Once we have a hash function, we can use the very efficient array-based implementation framework to store items in the table.
- Note that this implementation no longer allows sorting of the items.
- Questions:
  - What hash function should we use?
  - What do we do if two items result in the same hash value (a *collision*)?

Converting the Key to an Integer

- For simplicity, we can think of hash functions as functions that map large integers to smaller ones.
- Computing the hash value is then conceptually a two-step process: first convert the key into an integer and then compute the hash value.
- Since all data is ultimately stored in binary form, it is easy to interpret the contents of any memory address as an integer.
- We may also do a different kind of conversion in some cases.
- Example: Converting a string to an integer
  - To convert a standard 7-bit ASCII string, interpret it as an unsigned integer base 128.
  - The word `now` converts to

$$110 \cdot 128^2 + 111 \cdot 128^1 + 119 \cdot 128^0 = 1816567$$

  - The Python `ord` function can be used to convert characters to their ASCII codes.
  - Note that using this method can result in very large numbers!
- To convert floating point numbers to integers, we can simply multiply by a large number.

Choosing a Hash Function

- What makes a "good" hash function?
- A good hash function minimizes collisions and is easy to compute.
- For a "random" key, we would like the probability of each hash value to be "equally likely."
- To be more precise, we would like the output sequence to behave like a random sample from a discrete uniform random distribution.
- Importantly, this should be true whether the inputs are random or not!
- This ensures that the items are distributed evenly throughout the hash table.
- This is not easy to accomplish, since we have no idea ahead of time what the sequence of keys will be.
- Non-random input sequences typically cause the most problems.

Raw Bits

- An obvious class of hash functions is simply to consider some chunk of k bits of the key (e.g., the k most significant bits).
- How do we compute this hash function?
  - Assume x is a w-bit integer.
  - The index formed from the first k bits of x is the result of dividing by 2^{w-k} and rounding off, i.e., $h(x) = \lfloor x/2^{w-k} \rfloor$.
  - The index formed from the last k bits of x is the remainder after dividing by 2^k, i.e., $h(x) = x \mod 2^k$.
- Note that both of these hash functions must be used with a table of size 2^k.
- These hash functions are very fast to compute (why?).
- However, these are both notoriously bad hash functions, especially for strings (why?).

A Slight Improvement

- We can modify the approach a little to make the results behave a little more randomly.
  - Multiply the key by a number a between 0 and 1.
  - Multiply the fractional part of the answer by M and round off.
  - This can be written as $h(x) = \lfloor M(ax \mod 1) \rfloor$.
  - If $a = 1/2^k$ and $M = 2^l$, then the result is bits k through $k + l$ (very easy to compute).
- In practice, there are many values of a and M that work reasonably well.
- Taking $a = (\sqrt{5} - 1)/2$ (the *golden ratio*) seems to work well.

Modular Hashing

- Another variation on the theme of the previous slide is to take $h(x) = \lfloor ax \rfloor \mod M$.
- With this hash function, even $a = 1$ works well if we choose M properly.
- This is called *modular hashing* and is a very popular form of hashing.
- For reasons that will become clear, we typically choose M to be a large prime number, ideally not close to a power of two.
- In addition, we want the size of the table to be in a specified range.
- A challenge for this method is that computing a number satisfying all these requirements may be difficult.

Hashing Strings

- As mentioned previously, hashing strings can be problematic because a relatively small string can convert to huge integers.
- Example: The string "averylongkey" has 25 digits when converted to an integer!
- This is too large to be represented in most computers.
- With modular hash functions, we don't need to explicitly calculate the integer equivalent to obtain the hash value.
- We can calculate the result piece by piece using Horner's method.

```
def hash(str, M)
   h, a = 0, 128
   for c in str
      h = (a*h + ord(c)) % M;
   return h
```

Mixing It Up

- One way to make our hash function for strings appear more random is to "randomize" the value of the constant a.
- For this purpose, we build a simple pseudo-random number generator into the hash function.

```
def hash(str, M)
   h, a, b = 0, 31415, 27183
   for c in str:
      h = (a*h + ord(c)) % M
      a = a*b
   return h
```

- This idea can be extended to integers by multiplying each byte by a random coefficient in much the same fashion.
- One can prove that this method does produce an "ideal" hash function under certain theoretical assumptions.

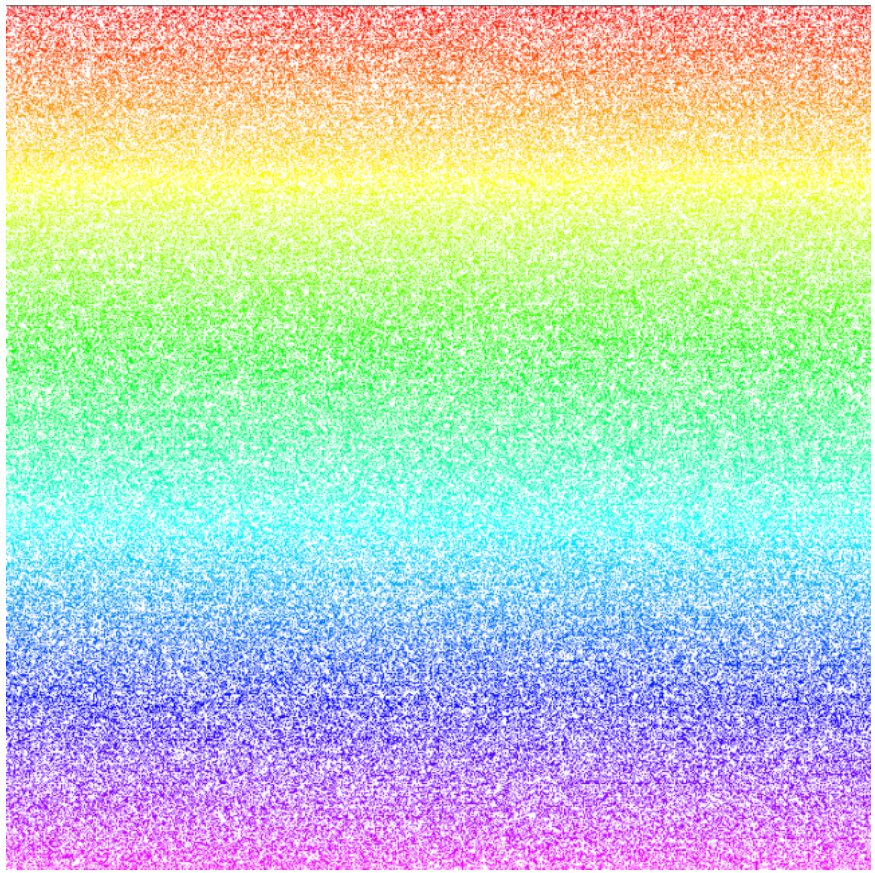
Evaluating Hash Functions

- There are several standard criteria used in evaluating hash functions.
  - Avalanche criteria: How many bits change in the output when one bit changes in the input.
  - χ^2 criteria: A statistical test for whether the distribution of hash values approximate iid samples from a discrete uniform distribution.
  - Speed: How efficiently the hash value can be computed.
- The χ^2 test uses Pearson's χ^2 with 1 degree of freedom.
- The test statistic is the sum of squared deviations

$$\frac{M/n^n}{\sum_{i=1}}(f_i - \frac{n}{M})^2$$

Visualizing Hash Function Distribution

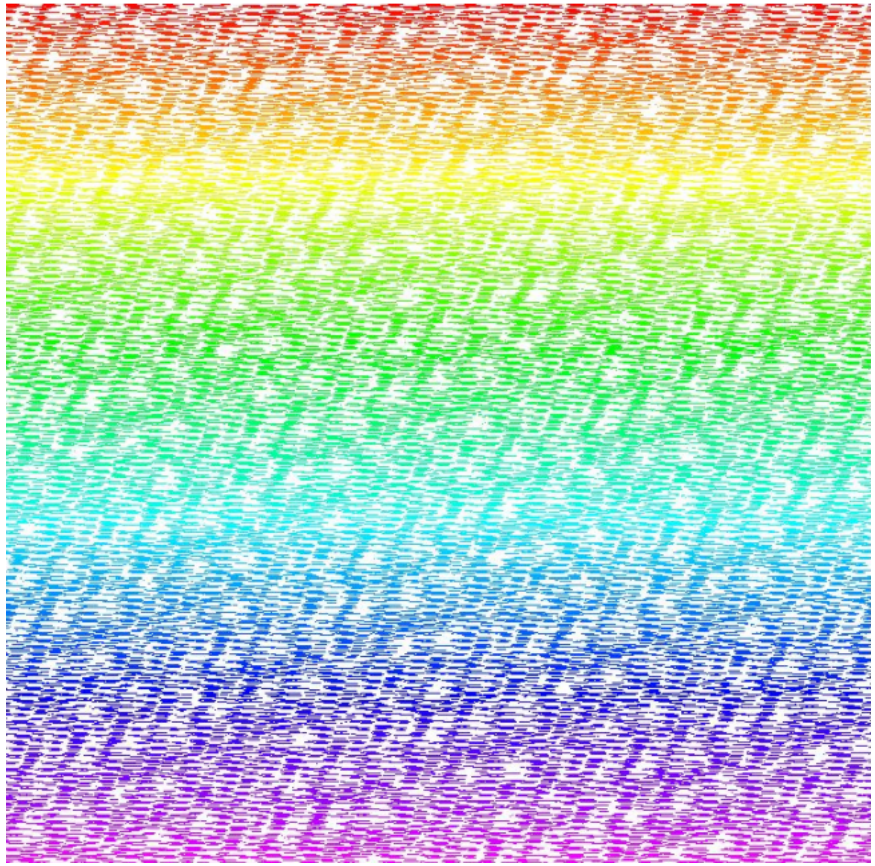
- It is possible to visualize the distribution of hash values for a given set of keys.
- Roughly, we want something that looks like this.



Source:
https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633#145633

Visualizing Hash Function Distribution

- Below is the distribution for the hash function SDBM when hashing numerical strings (think ZIP codes).
- Note the obvious pattern.



Source:
https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633#145633

Adding Sophistication: MurmurHash

Julia uses a variant of the MurmurHash Algorithm, which is designed to balanced performance on the χ^2 and avalanche tests.

```
function hash_64_64(n::UInt64)
    a::UInt64 = n
    a = ~a + a << 21
    a =  a ⊻ a >> 24
    a =  a + a << 3 + a << 8
    a =  a ⊻ a >> 14
    a =  a + a << 2 + a << 4
    a =  a ⊻ a >> 28
    a =  a + a << 31
    return a
end
```

Other Hash Functions

There are dozens of hash functions out there, but here are some of the more popular ones.

- DJB2
- FNV-1
- SDBM
- CRC32
- Murmur
- SuperFastHash

Source:
https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633#145633

Other Applications of Hashing

- Ensuring integrity of transferred files
- Encryption, Security, Crytptography
- Search
- File matching
- Syncing
- Blockchain (cryptocurrencies)
- Version Control (git)

The Rsync Algorithm

- Used to efficiently sync files over a network.
- Details are a bit involved, but the basic idea is this.
  - Break file B into blocks and apply a hash function to each block.
  - Send these signatures across the network.
  - Compute similar signatures for file A and only send the blocks for which the hash is different.
- The real algorithms uses two different hash functions and accounts for the possibility of not detecting differences when only using one hash function.

The `hashlib` Module

The `haslib` module contains implementations of many of the most commonly used hash functions.

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Nobody inspects")
>>> m.update(" the spammish repetition")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.digest_size
16
>>> m.block_size
64
```

Resolving Collisions

- There are two primary of methods of resolving collisions.
- Chaining: Form a linked list of all the elements that hash to the same value.
  - Easy to implement.
  - The table never "fills up" (better for extremely dynamic tables)
  - May use more memory overall.
  - Easy to insert and delete.
- Open Addressing: If the hashed address is already used, use a simple rule to systematically look for an alternate.
  - Very efficient if implemented correctly.
  - When the table is nearly full, basic operations become very expensive.
  - Deleting items can be very difficult, if not impossible.
  - Once the table fills up, no more items can be added until items are deleted or the table is reallocated (expensive).

Analysis of a Hash Table with Chaining

- Insertion is always constant time, as long as we don't check for duplication.
- Deletion is also constant time if the lists are doubly linked.
- Searching takes time proportional to the length of the list.
- How long the lists grow on average depends on two factors:
  - how well the hash function performs, and
  - the ratio of the number of items in the table to its size (called the *load factor*).

Length of the Linked Lists

- We will assume *simple uniform hashing*, i.e., that any given key is equally likely to hash to any address.
- Let the load factor be α.
- Under these assumptions, the average number of comparisons per search is $\Theta(1+\alpha)$, the average chain length plus the time to compute the hash value.
- If the table size is chosen to be proportional to the maximum number of elements, then this is just $O(1)$.
- This result is true for both search hits and misses.
- Note that we are still searching each list sequentially, so the net effect is to improve the performance of sequential search by a factor of M.
- If it is possible to order the keys, we could consider keeping the lists in order, or making them binary trees to further improve performance.

Related Results

- It can be shown that the probability that the maximum length of any lists is within a constant multiple of the load factor is very close to one.
- The probability that a given list has more than $t\alpha$ items on it is less than

$$\left(\frac{\alpha e}{t}\right) e^{-\alpha}$$

- In other words, if the load factor is 20, the probability of encountering a list with more than 40 items on it is $.0000016$.
- A related result tells that the number of empty lists is about $e^{-\alpha}$.
- Furthermore, the average number of items inserted before the first collision occurs is approximately $1.25\sqrt{M}$.
- This last result solves the classic *birthday problem*.
- We can also derive that the average number of items that must be inserted before every list has at least one item is approximately $M \ln M$.
- This result solves the classic *coupon collector's problem*.

Table Size with Chaining

- Choosing the size of the table is a perfect example of a *time-space* tradeoff.
- The bigger the table is, the more efficient it will be.
- On the other hand, bigger tables also mean more wasted space.
- When using chaining, we can afford to have a load factor greater than one.
- A load factor as high as 5 or 10 can work well if memory is limited.

Open Addressing

- In *open addressing*, all the elements are stored directly in the hash table.
- If an address is already being used, then we systematically move to another address in a predetermined sequence until we find an empty slot.
- Hence, we can think of the hash function as producing not just a single address, but a sequence of addresses $h(x, 0), h(x, 1), \ldots, h(x, M-1)$.
- Ideally, the sequence produced should include every address in the table.
- The effect is essentially the same as chaining except that we compute the pointers instead of storing them.
- The price we pay is that as the table fills up, the operations get more expensive.
- It is also much more difficult to delete items.

Linear Probing

- In *linear probing*, we simply try the addresses in sequence until an empty slot is found.
- In other words, if h' is an ordinary hash function, then the corresponding sequence for linear probing would be

$$h(x,i) = (h'(x) + i) \mod M, i = 0, \ldots, M-1.$$

- Items are inserted in the first empty slot with an address greater than or equal to the hashed address (wrapping around at the end of the table).
- To search, start at the hashed address and continue to search each succeeding address until encountering a match or an empty slot.
- Deleting is more difficult
  - We cannot just simply remove the item to be deleted.
  - One solution is to replace the item with a sentinel that doesn't match any key and can be replaced by another item later on.
  - Another solution is to rehash all items between the deleted item and the next empty space.

Analysis of Linear Probing

- The average cost of linear probing depends on how the items cluster together in the table.
- A *cluster* is a contiguous group of occupied memory addresses.
- Consider a table with half the memory locations filled.
  - If every other location is filled, then the number of comparisons per search is either 1 or 2, with an average of 1.5.
  - If the first half of the table is filled and the second half is empty, then the average search time is $1 + (\sum_{i=1}^{n} i)/(2n) \approx n/4$.
- Generalizing, we see that search time is approximately proportional to the sum of squares of the lengths of the clusters.

Further Analysis of Linear Probing

- The average cost for a search miss is

$$1 + (\sum_{i=1}^{l} t_i(t_i + 1))/(2M)$$

  where l is the number of clusters and t_i is the size of cluster i.

- This quantity can be approximated in the case of linear probing.
- On average, the time for a search hit is approximately

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

  and the time for a search miss is approximately

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

- These approximations lose their accuracy if α is close to 1, but we shouldn't allow this to happen anyway.

Clustering in Linear Probing

- We have just seen why large clusters are a problem in open addressing schemes.
- Linear probing is particularly susceptible to this problem.
- This is because an empty slot preceded by i full slots has an increased probability, $(i+1)/M$, of being filled.
- One way of combating this problem is to use *quadratic probing*, which means that

$$h(x,i) = (h'(x) + c_1 i + c_2 i^2) \mod M, i = 0, \ldots, M-1$$

- This alleviates the clustering problem by skipping slots.
- We can choose c_1 and c_2 such that this sequence generates all possible addresses.

Double Hashing

- An even better idea is to use *double hashing*.
- Under a double hashing scheme, we use two hash functions to generate the sequence as follows.

$$h(x,i) = (h_1(x) + ih_2(x)) \mod M, i = 0, \ldots, M-1$$

- The value of $h_2(x)$ must never be zero and should be relatively prime to M for the sequence to include all possible addresses.
- The easiest way to assure this is to choose M to be prime.
- Each pair $(h_1(x), h_2(x))$ results in a different sequence, yielding M^2 possible sequences, as opposed to M in linear and quadratic probing.
- This results in behavior that is very close to ideal.
- Unfortunately, we can't delete items by rehashing, as in linear probing.
- To delete, we must use a sentinel.

Analyzing Double Hashing

- When collisions are resolved by double hashing, the average time for search hits can be approximated by

$$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

and the time for search misses is approximately

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

- This is a big improvement over linear probing.
- Double hashing allows us to achieve the same performance with a much smaller table.

Worst Case Analysis

- So far, we have only looked at average performance over all possible inputs.
- Particular inputs may not exhibit the nice behavior seen on average.
- As with many algorithms, worst case behavior is easy to find.
- For any hash function, there is always a sequence of inserts that will lead to poor behavior.
- For both open addressing and chaining, a sequence of n inserts could require $\theta(n^2)$ steps.
- A common way to protect against worst-case behavior in algorithms is to *randomize* in case a certain common patern leads to the worst case.
- For this purpose, we can use the universal has functions described earlier.

Dynamic Hash Tables

- *Dynamic hash tables* attempt to overcome the limitations of open addressing when the number of table items is not known at the outset.
- When the table fills up beyond a certain threshold, we simply allocate a new array and rehash all the existing items.
- This operation is expensive, but it happens infrequently.
- Using a technique called *amortized analysis*, we can show that the average cost of each operation is still approximately constant.
- This may be a good option in some situations.

Python's Hash Table Implementation

- Python uses open adressing with a variant of double hashing.

```
self.mask = newsize - 1
perturb = key_hash
while True:
    i = (i << 2) + i + perturb + 1;
    entry = self.table[i & self.mask]
    if entry.key is None:
        return entry if free is None else free
    if entry.key is key or \
            (entry.hash == key_hash and key == entry.key):
        return entry
    elif entry.key is dummy and free is None:
        free = dummy
    perturb >>= PERTURB_SHIFT
```

- The table size is initially 8 and is increased by a factor of 4 whenver it fills up.
- Deletion is by sentinel.