References for Today’s Lecture

- References
  - CLRS Section 11.1, Chapter 12
Hash Tables

• We now consider data structure for storing a dictionary that support only the operations
  – insert,
  – delete, and
  – search.

• Most 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 extremely 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 $h : U \rightarrow 0, \ldots, M - 1$ 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*)?
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.”
- This assures that the items are distributed evenly throughout the hash table.
- This is not as easy to accomplish as it sounds!
- It depends on what the distribution of possible key values is.
- We may not know the distribution ahead of time.
Significant Bits

- Two obvious hash functions are to simply consider either the first (most significant) or last (least significant) $k$ bits of the key.

- 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?).
An Improved Hash Function

- The method of the previous slide can be made to work better simply by changing the size of the hash table.
- To hash a key $x$, take $x \mod M$, where $M$ is the size of the hash table.
- This is called **modular hashing** and is a very popular form of hashing.
- To avoid the problems discussed on the last slide and for reasons that will become clear later, it is best to choose $M$ to be prime.
- Choosing $M$ to be close to a power of two can also cause problems.
- In addition, we want the size of the table to be in a specified range.
- Computing a number satisfying all these requirements can be difficult.
- In practice, such numbers can be looked up in a table.
Other Simple Hash Functions

• Another approach to improving the method of significant bits is to consider the bits in the middle.

• How would we compute this hash function?
  – 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).

• The advantage of this method is that the value of $M$ is not as critical.

• In practice, there are many values of $A$ and $M$ that work well.

• Taking $A = (\sqrt{5} - 1)/2$ (the golden ratio) seems to work well.

• Another variation on the theme is to take $h(x) = \lfloor Ax \rfloor \mod M$. 
Converting the Key to an Integer

• To end up with a valid table address, we must convert the key into a natural number at some point.

• **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.
Hashing Strings

- As mentioned previously, hashing strings can be problematic because a relatively small string can convert to a huge integer.

- **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.

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

• A universal hash function is one in which the probability of a collision between any two keys is provably $1/M$.

• With chaining, one can prove that any sequence of $n$ inserts, deletes and searches (with $O(M)$ inserts) will take $O(n)$ steps.

• Implementing universal hash functions necessarily involves some randomization.

• Here are two approaches.
  – Choose the hash function randomly from a class selected to yield the desired property.
  – Randomize the hash function itself.

• These two methods amount to the same thing.

• For this to work, the randomization has to be independent of the keys.

• Generally, universal hashing isn’t worth the additional computation required, but we will look at two simple universal hash function.
A Universal Hash Function for Strings

• Consider our earlier modular hash function for strings.
• One way to randomize this hash function is to randomize the value of the constant $a$.
• We will use an inexpensive pseudo-random number generator for this purpose.
• Here is our new hash function.

```python
def hash(str, M):
    h, a, b = 0, 31415, 27183
    for c in str:
        h = (a*h + ord(c)) % M
        a = a+b % (M-1)
        if h < 0:
            return h + M
    else:
        return h
```

• This idea can be extended to integers by multiplying each byte by a random coefficient in much the same fashion.
A Universal Hash Function for Integers

- Another universal hash function is obtained as follows.
- Let $p$ be a large prime number such that every key value is between 0 and $p-1$.
- Then let $a$ and $b$ be integers smaller than $p$ with $a$ positive and $b$ nonnegative.
- If $a$ and $b$ are selected randomly, then the hash function

$$h_{a,b}(k) = (((ak + b) \mod p) \mod M) \quad (1)$$

is universal.
Python’s built-in hash function

This is the hash function used by Python in its implementation of dictionaries.

```python
def hash(s):
    if not s:
        return 0 # empty
    value = ord(s[0]) << 7
    for char in s:
        value = cmul(1000003, value) ^ ord(char)
    value = value ^ len(s)
    if value == -1:
        value = -2
    return value
```

The `hash` function is built into Python and can be called on any object.
Hashing tuples

Tuples are hashed recursively

def hash(key):
    value = 0x345678
    for item in key:
        value = c_mul(1000003, value) ^ hash(item)
    value = value ^ len(self)
    if value == -1:
        value = -2
    return value
Other Applications of Hashing

- Ensuring integrity of transferred files
- Cryptography
- Search
- File matching
- Syncing
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 `hashlib` module contains implementations of many of the most commonly used hash functions.

```python
>>> 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 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 $\alpha$.

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

• In other words, if the load factor is 20, the probability of encountering a list with more than 40 items on it is 0.000016.

• 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 $1 + \left( \sum_{i=1}^{n} i \right) / (2n) \approx n/4$.

- 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

\[
1 + \left( \sum_{i=1}^{l} t_i(t_i + 1) \right)/(2M)
\]

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 \( \alpha \) 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, \( \frac{i + 1}{M} \), of being filled.
- One way of combating this problem is to use quadratic probing, which means that

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

- 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 pattern leads to the worst case.

• For this purpose, we can use the universal hash functions described in Lecture 9.
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 addressing with a variant of double hashing.

```python
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 whenever it fills up.

- Deletion is by sentinel.