References for Today’s Lecture

• Required reading
  – CLRS Chapter 11

• References
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

• 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
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.

• 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, \( (i + 1)/M \), of being filled.
- One way of combating this problem is to use **quadratic probing**, which means that

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

- 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.

• 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 average 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.
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

- 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.

- From here on, we will assume all keys are natural numbers.
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.

```c
int hash(char *v, int M)
{
    int h(0), a(128);
    for (; *v != 0; v++)
    {
        h = (a*h + *v) % M;
    }
    return h;
}
```
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.

• As we have done with previous algorithms, to protect against worst-case behavior, we need to randomize.
Universal Hashing

- A *universal hash function* is one in which the probability of a collision between any two keys is provably $\frac{1}{M}$.

- 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.

- These two methods amount to the same thing.

- The idea is to avoid worst-case behavior induced by non-random inputs, as in quicksort.

- 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.

```c
int hash(char *v, int M) {
    int h(0), a(31415), b(27183);
    for (; *v != 0; v++, a = a*b % (M-1))
        h = (a*h + *v) % M;
    return (h < 0) ? (h + M) : 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

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

is universal.
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

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

is universal.
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.