Learning Goals

During this lab, you will:

- Motivate the need for hash tables
- Review the Simple Uniform Hashing Assumption
- Compare collision resolution strategies (chaining, open addressing, double hashing)

Hashing

Motivation

As you’ve undoubtedly already realized, many applications call for (large) dynamic sets that require support for the basic dictionary operations SEARCH(X), INSERT(X) and DELETE(X). As a concrete example, imagine that we had the set of all Penn students’ names and we wanted quick lookups to see if a name existed. We might turn to a linked list, but lookup is $O(n)$. A better choice might be a balanced BST with student names as keys. This would give us $O(\log n)$ time lookups and insertions, which is already pretty good!

But say we’re going to be doing a lot of repeated lookups, so we’d like lookups on the order of constant time. How could we achieve this?

The Theory of Hashing

Direct Addressing

The first intuitive idea is to leverage the constant time lookup property of arrays. We can define a universe $U$ of possible keys (which in this case is all possible strings up to size $l$, where $l$ is the length of the longest name) and keep an array $A$ of size $|U|$. Then we can define a one-to-one mapping $M$ where we assign every string in $U$ to a number from 1 to $|U|$ and keep every name $k$ in the location $A[M(k)]$.

This method would give us guaranteed $O(1)$ lookups and inserts, but would take space $\Theta(|U|)$ which can be impractically large. For example, assuming that the longest name at Penn is 20 characters long (including spaces), there would be $27^{20}$ strings in our universe. This is clearly inefficient because there is no way we are going to be storing that many names.

Thus, Direct Addressing could be a good option when $U$ is small. But as $U$ becomes very large, storing an array of size $|U|$ can be impractical due to memory limitations. Furthermore, often the set of actual keys $K$ stored is much less than $|U|$, as such the extra space allocated to $A$ would be wasted.

Hash Tables

Instead of having one-to-one mappings from keys to slots in our array (referred to more generally as “table”) of size $|U|$, we can define a deterministic hash function $h$ that maps each key to some slot in our table $T[0...m-1]$ of size $m$. Formally, the hash function $h$ is:

$$h : U \rightarrow \{0, ..., m-1\}$$
Figure 1: Here, every key in the universe is mapped to a different slot in the hash table $T$.

where usually $m << |U|$.

By initializing a table of size $m << |U|$ and defining the hash function $h$, we’ve managed to save a lot of space. However, this improvement (like many things in life), comes at a cost: insertions and lookups are now on average $O(1)$ — worst case for some particular insert or lookup may be $O(n)$ — instead of guaranted $O(1)$. Furthermore, by the pigeonhole principle, two keys in $U$ may now hash to the same slot, a situation known as a collision. Separate chaining and Open Addressing are two common ways of resolving collisions, which we’ll explore more in depth later.

N.B.: Our opening example used student names (which are strings) as keys. It should be noted that it is not difficult to convert any key to some non negative integer. As such, in the rest of our discussions we assume keys are non-negative integers.

A brief note on the hash function

Let’s take a brief moment to consider what might make a good hash function. Consider the hash function $h(k) = k \mod 2$. We quickly realize that this is a horrible hash function — regardless of how large our symbol table is, we will have repeated collisions at slots 0 and 1.

It intuitively follows that we seek a hash function that distributes the universe $U$ of keys as evenly as possible among the $m$ slots of our table. We can formalize this concept and give it a name:

**Simple Uniform Hashing Assumption:** Under this assumption, any key $k$ is equally likely to be mapped to any of the $m$ slots in our hash table $T$, independently of where any other key is hashed to. In other words, the probability of hashing some key $k$ not already present in the hash table into any arbitrary slot in $T$ is $\frac{1}{m}$.

A direct result of this assumption is that if we would like to store $n$ elements in a table of size $m$, each slot will roughly get the same number $\left(\frac{n}{m}\right)$ of collisions. This is also known as the **load factor**, commonly denoted as by $\alpha$, of the table. The simple uniform assumption, while potentially unrealistic, is used in the

\[1\] To be more rigorous, we can define a random variable $X$ denoting the total number of elements we have hashed into $T$ and
running time analysis of different collision resolution schemes, so make sure to keep it in mind.

The simple universal assumption, is afterall, just an assumption. In practice we rarely know the probability distribution from which the keys are drawn, and so we employ various heuristics when designing hash functions to achieve approximately an even distribution. See CLRS section 11.3 for a more in depth discussion of this topic.

Furthermore, here’s something cool. Suppose that in our implementation of HashTable we always use the same hash function for every instance of our table. This is quite dangerous because if an adversary got hold of our hash function, he or she could work out what keys collide in our hash function and thus devise a dataset that would slow down our code and bring our system to a halt. The attacker can actually exploit the fact that insertions and lookups for hashtables are worst case $O(n)$ — say when all elements hash to the same slot — to start a denial of service attack on your application.

Thus in practice we often introduce randomization techniques to alleviate this worst case behavior. We maintain a carefully selected set of hash functions called a hash family, and whenever a new instance of our table is initialized, we choose a hash function uniformly at random from this set of hash functions to use for that specific table. Since the adversary doesn’t know which hash function we’ll use before hand, he or she cannot engineer a specific dataset that takes advantage of any particular hash function. The latest version of Java does this automatically. Note that each hash function $h$ in the hash family is still deterministic — that is, for a particular key $k$ and hash function $h$, $h(k)$ always has to produce the same result. The randomization comes from which hash function we choose to use when we initialize a new Hash Table. Once a hash function is chosen uniformly at random for an instance, it is used for the duration of the instance. There is a precise notion as to how to carefully select this set of hash functions (universal family) to consistently give good average case behavior, in depth discussion of which can be found in CLRS 11.3.3.

Collision Resolution: Separate Chaining

The most obvious way to handle collisions is with a linked list. In other words, we chain all the keys that are hashed to slot $i$ in a linked list and store that list at slot $i$ in $T$. To find an element hashed to slot $i$, we would simply traverse the linked list until we found it (in the case of a successful search) or until we reached the end of the list (in the case of an unsuccessful search). Because of the simple uniform hashing assumption, the keys are roughly equally distributed amongst the $m$ linked lists in our hash table. With some math, we can show that on average, the number of elements examined in a successful search is $\Theta(1 + \alpha)$ where $\alpha$ is the load factor. The number of elements examined in an unsuccessful search is $\Theta(1 + \alpha)$ as well. If $\alpha$ is small, we can say that this is $O(1)$. However, we do not always know what $\alpha$ is at any given moment—thus we say that we can perform insertions and searches in expected $O(1)$ time.

$X_i$ is the number of elements that have been hashed to slot $i$.

$$E[X_i] = \sum_{j=1}^{n} \Pr[\text{element } j \text{ hashed to slot } i] = \sum_{j=1}^{n} \Pr[h(k_j) = i]$$

$$= \sum_{j=1}^{n} \frac{1}{m} \quad \text{(by Simple Uniform Hashing)}$$

$$= \frac{n}{m}$$

As a check, we can use Linearity of Expectation to solve for $E[X]$:

$$E[X] = \sum_{i=1}^{m} E[X_i]$$

$$= \sum_{i=1}^{m} \frac{n}{m}$$

$$= n.$$
Collision Resolution: Open Addressing

Preface: In this discussion, we will not consider deletions, as they can be quite tricky to implement with open addressing (See CLRS pg 271). As such, chaining is more commonly used when support for deletion of keys is necessary.

There are other ways besides chaining to handle collisions, and most of these strategies fall under the broad title of open addressing. In the technique of open addressing, instead of storing values that map to the same slot in a linked list, we find the “next” open spot in the same hash table to store our value. A consequence of this is that the hash table may become full, and the load factor \( \alpha = \frac{n}{m} \) will always be less than or equal to 1.

When inserting a new element, we successively examine, or probe, the hash table according to some rule until we find an open spot to put our new element. We can define our probe sequence as \( S = \{h_0(x), h_1(x), h_2(x), \ldots, h_{m-1}(x)\} \), where \( h_i(x) \) represents the \( i \)th index we examine. We also say \( h_0(x) = h(x) \). Defining \( h_i(x) \) gives us the pre specified rules as to what index to check next—otherwise, we would just be choosing randomly. Note that you can make up your own rules as to how \( h_i(x) \) is defined, but we need to make sure that the probe sequence is a permutation of \( \{0 \ldots m-1\} \) to ensure that eventually all slots are considered when inserting a new element as the table fills up. As we will see, there are different ways to specify our probe sequence, some yielding better results than others.

This pseudocode below for search (which is very close to insert) illustrates how open addressing works. Note that if an empty slot is found when searching for an element \( x \), we can conclude that element doesn’t exist (assuming no deletions), since otherwise it would’ve been placed into that slot during insertion. The expected time for an unsuccessful search with this setup is \( O\left(\frac{1}{1-\alpha}\right) \), which is \( O(1) \) for values of \( \alpha \) not near 1. The expected number of probes for a successful search is harder to derive, and is given by \( O\left(\frac{\alpha}{\ln(1-\alpha)}\right) \), which is also \( O(1) \) when \( \alpha \) is small (See CLRS pg 276 for derivation).

Probing Techniques

There are variety of ways to define our probing function \( h_i(x) \). In linear probing, we define our \( h_i(x) = (h(x) + i) \mod m \). This has us taking equal sized steps through the table. While quick when the load factor is small (due to caching effects), as the load factor becomes large, \texttt{insertion()} and \texttt{search()} can
function search(x):
    index ← h(x)  
    ▶ Finds x in our hash table
    for i ← 1 to m do
        if T[index] = x then
            return true  
            ▶ We found x!
        else if T[index] = ∅ then
            ▶ If encounter empty slot during probing, x doesn’t exist
            return false
        end if
        index ← h_i(index)  
        ▶ Compute the next index to “probe”
    end for
    return false  
    ▶ Repeat unless entire probe sequence is complete
end function

end up traversing the entire table. This is due to a problem called **primary clustering**. This is just what it sounds like. Because the difference between consecutive probes is linear, elements tend to form clusters. This is because once just a few elements are placed next to each other in the table, it becomes increasingly likely that an element will hash to one of the indices occupied by an element in that cluster. Then, because linear probing is used, that element will be placed near the cluster once it finally finds an open address, effectively increasing the size of the cluster. This can be mitigated by other forms of probing such as **quadratic probing**: \( h_i(x) = (h(x) + i^2) \mod m \). However, quadratic probing suffers from a milder yet similar problem called **secondary clustering**. An even more robust form of open addressing is **double hashing** (where we choose another hashing function \( h′(x) \) so \( h_i(x) = (h(x) + i \cdot h′(x)) \mod m \)). This makes the probe sequence vary depending on the element being inserted instead of the initial probe index, effectively eliminating clustering.

![Figure 3: Insertion of \( k = 14 \) with double hashing, with \( h(k) = k \mod 13 \) and \( h′(k) = 1 + (k \mod 11) \). Hash table is size 13 so \( m = 13 \). \( h(14) = 14 \mod 13 = 1 \) and \( h′(k) = 1 + (14 \mod 11) = 4 \). Hence procedure inserts 14 into slot 9 after probing slots 1 and 5 (recall \( i \) begins at 0).](image)

If at this point you’re feeling a little lost as to how probing works, don’t worry — the formalization makes it seem confusing but working out a few simple examples make linear and quadratic probing as well as double hashing much clearer. Remember, these probing techniques are no more than “rules”, which we are even free to make up ourselves, to specify the next index to check!
Problems

Balls and Bins

With \( n \) distinct balls in \( m \) distinct bins what is the probability that no bucket has more than 1 ball? You may assume that \( n \leq m \).

Solution

We will treat this as a counting problem. That is, we count the number of ways the event that no bucket has more than 1 ball, and divide by the total amount of ways to divide the \( n \) balls into \( m \) buckets. This technique works because each division of the \( n \) balls is equally likely. The numerator of our expression is \( mP_n = \frac{m!}{(m-n)!} \). The intuition behind this is that we have to pick exactly \( n \) bins out of our \( m \) bins if each bin that is filled has exactly one ball in it. Because our balls are distinct, we deal with permutations and not combinations. The denominator is immediately \( m^n \), because there are \( m \) choices for each of our \( n \) balls. This gives:

\[
\frac{m!}{(m-n)!} \frac{1}{m^n}
\]

Problem 1

You are given an array \( A \) containing distinct randomly assorted integers. Your goal is to find two elements in the array whose sum is \( k \) in \( O(n) \) expected time.

Solution

Begin by traversing \( A \) and putting all of the elements in a hash table. Then traverse \( A \) again, and for each element \( e \), check and see if \( k-e \) exists in the hash table (and make sure \( k-e \neq e \)). If it does, simply return \( e \) and \( k-e \). If you reach the end of the array and haven’t found a pair, return that no such pair exists.

Running time Analysis: It takes expected \( O(n) \) time to insert all of the elements of \( A \) into the hash table, \( O(n) \) time to traverse \( A \) again, and expected \( O(1) \) time for each of the \( n \) elements (yielding expected \( O(n) \) time) to search for its complement. Therefore our algorithm still runs in expected \( O(n) \) time.

Problem 2

Assume we have a hash table \( T \) of size 10 that uses linear probing and has hash function \( h(x) = x \text{ mod } 10 \). We insert 6 numbers into \( T \) and we get the below table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is one possible order that we could have inserted these elements to get this result?

How many probes would be required for inserting 13 in the table?
Solution

One possible order that the numbers could have been inserted in is 46, 34, 42, 23, 52, 33. For a sequence to be valid:

- 46 must be inserted before 33
- 42, 34, and 23 must be inserted before 33 and 52

It would take 6 probes to insert 13 into the table, as we would first try to insert it in index 3, fail, and then traverse the cluster until we reach index 8, at which point it would be inserted after we realize that \( T[8] \) is empty. Note that this is a prime demonstration of primary clustering.

Problem 3

How would you detect a cycle in a linked list of distinct elements in expected \( O(n) \) time?

Can you do it in constant space?

Solution

Our algorithm is as follows: Traverse the linked list, and keep a hash table of every element we’ve encountered. As we reach a new element in the linked list, check to see if it already exists in our hash table. If it does, we’ve found a cycle. If we reach the end of the linked list, then we know that no cycle exists.

Running time Analysis: It takes linear time to traverse the linked list and expected \( O(1) \) time for each lookup and insertion in the hash table, of which there could be \( 2n \). Thus, the algorithm is expected \( O(n) \) time.

To do it in constant space, we can begin with two pointers \( i \) and \( j \) and traverse the array. To traverse the array, we move the pointer \( j \) two spots along the linked list for every spot we move \( i \). If there is a cycle, we know that \( j \) will eventually intersect with \( i \) (think about why this is!). If there is no cycle, then \( j \) will reach the end of the linked list and we are done.

Running time Analysis: We can say this algorithm is \( O(n) \)—try to figure out why this is and see if you can come up with a more specific bound in the case of a cycle!

Problem 4

Design an algorithm that determines if two lowercase words are anagrams of each other in expected \( O(n) \) time. Note: a string \( A \) is an anagram of another string \( B \) if \( A \) is a permutation of \( B \).

Can you do it in worst case \( O(n) \) time?

Solution

Expected \( O(n) \) time solution: First of all, if \( A \) and \( B \) aren’t the same length, they cannot be permutations. Otherwise, use two hash tables where the keys are letters and the values are the counts of those letters. Iterate through \( A \) incrementing the count of a letter in \( A \)’s hash table every time that letter is seen in \( A \). Similarly, iterate through \( B \) incrementing the count of a letter in \( B \)’s hash table every time that letter is seen in \( B \). This effectively builds a map that maps from every letter in \( A \) to the number of times that letter appears in \( A \). It also does the same for \( B \). Then, iterate through the letters in \( A \)’s map, and see if their counts are the same as their counts in \( B \)’s map. If this passes, we know \( A \) and \( B \) are the same. Note that we didn’t have to iterate through the letters of \( B \) because we know \( A \) and \( B \) are the same length.

Worst Case \( O(n) \) time solution: Because there are only 26 lowercase letters in the alphabet, we can just use 2 arrays of size 26 each to store the counts. Then, it’s just the same algorithm as was used above except the keys above are just indices in the arrays and the values above are the values in the arrays at those indices. Because arrays support worst case \( O(1) \) time insertions and lookups, and we iterate through each string once, this implementation runs in worst case \( O(n) \) time.

Alternatively, you could just build the hash table/array the same way for string \( A \), then traverse string \( B \) and for each letter, decrement the corresponding counter in \( A \)’s map/array. If a counter ever becomes \(-1\)
or the letter did not exist in $A$, return false. Otherwise, return true (assuming we already checked if $A$ and $B$ are the same length).