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)

The Theory of Hashing

Direct Addressing

Imagine we had a very large dynamic set (such as all Penn students’ names) and we wanted quick lookups to see if a name existed. As you have seen, we could turn to a BST to store this information—the keys of the BST could be student names. This would give us $O(\log n)$ time lookups and insertions, which is quite fast!

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 do we do this? One way is to define our 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 $A$ the size of $U$. Then we can define a mapping $M$ where we assign every number from 1 to $\text{size}(U)$ to a string in $U$ and keep every name in the location $A[M(\text{name})]$.

![Figure 1: Here, every key in the universe is mapped to a different slot in the hash table $T$.](image)

This method would give us $O(1)$ lookups and inserts, but would take space $O(\text{size}(U))$ which can get very large indeed! For example, conservatively 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.
Hash Tables

Instead of having a function with one-to-one mappings, we can define a hash function \( h \) that maps all the keys in the universe to the \( m \) slots in our table. Because it is usually the case that \( m << \text{size}(U) \), there is a chance that two keys in the universe are mapped to the same slot in our table, which is known as a collision. Granted, if we knew exactly what data was going to be stored in the table beforehand, we could (somewhat painstakingly) find a perfect hash function that resulted in no collisions. But this is not usually the case. So, how do we choose a hash function? We could just set a predefined hash function and let it be, since it should perform decently well for any arbitrary dataset. What if, however, someone decided to hack our hash function? If this hash function was being used in some important piece of software and the hacker knew what keys collided in our hash function, they could devise a dataset that would slow down our code and bring our system to a halt. Thus, we avoid this worst-case behavior by choosing our hash function uniformly at random from a set of hash functions called a hash family.

The hash function \( h \) is defined under the simple uniform hashing assumption, which says that every key \( k \in U \) is equally likely to be mapped to any slot in the hash table \( T \) by \( h \). This ensures that each slot will get roughly the same number of collisions (\( \frac{n}{m} \), which is also called the load factor or \( \alpha \)).

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 (\( \frac{n}{m} \)). The number of elements examined in an unsuccessful search is \( \Theta(1 + \alpha) \) as well (\( \frac{n}{m} \)). 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.

Open Addressing

Preface: In this discussion, we will not consider deletions, as they can be quite tricky to implement with open addressing and there are many different ways of handling them.

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 resolving collisions with a linked list, we simply find another open spot in the hash table to store our value. So, when inserting, if our hash function says we should hash to index \( i \), and \( T[i] \) is already full, we simply jump to another index in

\[
E[X_i] = \sum_{j=1}^{n} \Pr[\text{element } j \text{ hashed to slot } i] = \sum_{j=1}^{n} \Pr[h(k_j) = i] \\
= \sum_{j=1}^{n} \frac{1}{m} \quad \text{(by Simple Uniform Hashing)} \\
= \frac{n}{m}
\]

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

\[
E[X] = \sum_{i=1}^{m} E[X_i] \\
= \sum_{i=1}^{m} \frac{n}{m} \\
= n.
\]
Figure 2: Here, there is a collision between $k_2$ and $k_5$ since they are mapped to the same slot.

$T$. If that is full, we repeat, but if not, we insert our element there. If we keep jumping to another index in the table, we can write the indices we jump to as a sequence $S = \{h_0(x), h_1(x), h_2(x), \ldots, h_{m-1}(x)\}$, where $h_i(x)$ represents the $i$th index we check. We also say $h_0(x) = h(x)$. Defining $h_i(x)$ also gives us some order as to how we choose what index to go to next—otherwise, we would just be choosing randomly.

We can write the pseudocode for search (which is very similar to insertion) as follows: This pseudocode illustrates how open addressing works: once we check an index, we should never check it again, almost like it doesn’t exist in the table anymore. This set has cardinality $m$, since the element we are looking for could be at any other location in the array. If we are searching and come upon a blank element, we know that the element we are searching for cannot exist because if it was inserted, it would have been found before we hit a blank element. The expected time for an unsuccessful search with this setup is $O(1/1-\alpha)$, which is $O(1)$ for values of $\alpha$ not near 1.

<table>
<thead>
<tr>
<th>function</th>
<th>SEARCH(x);</th>
</tr>
</thead>
<tbody>
<tr>
<td>index ← $h(x)$</td>
<td>▶ Finds $x$ in our hash table</td>
</tr>
<tr>
<td>for $i$ ← 1 to $m$ do</td>
<td></td>
</tr>
<tr>
<td>if $T[index] = x$ then</td>
<td></td>
</tr>
<tr>
<td>return true</td>
<td></td>
</tr>
<tr>
<td>else if $T[index] = \emptyset$ then</td>
<td></td>
</tr>
<tr>
<td>return false</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>index ← $P(index)$</td>
<td></td>
</tr>
<tr>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>return false</td>
<td></td>
</tr>
<tr>
<td>end function</td>
<td></td>
</tr>
</tbody>
</table>

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, insertion() and search() can 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 \( h_i(x) = (h(x) + i \times h'(x)) \mod m \)). This makes the probe sequence vary depending on the element, effectively eliminating clustering.

### Hash Maps in the Real World

The debate between chaining and open addressing does not have a clear-cut winner; there are merits to both solutions and both are widely used in the real world. For example, Java has chosen to implement hash tables with chaining, while Python chose to use open addressing with random probing.

Since we are working with Java, however, we will analyze Java’s implementation.

### Java HashMap

Java’s [HashMap constructor](#) allows the user to define an initial capacity and a load factor. HashMap is implemented with separate chaining. The default capacity is 16, and the default load factor is 0.75.

HashMap rehashes every time, so number of keys > load factor * capacity.

### Testing

We will vary initial capacity and load factor values and look at some performance results.

In our test program, we call `put` on a million random numbers, then we call `get` on the same million numbers. We run this test 100 times to eliminate the possibility of bad sets of numbers or CPU usage spikes affecting the result. The average times are then recorded (in milliseconds) for the `put` and `get` operations.

The following table varies initial capacity and uses the default load factor 0.75.

<table>
<thead>
<tr>
<th>Initial Capacity</th>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^2)</td>
<td>194.88</td>
<td>55.86</td>
</tr>
<tr>
<td>(2^3)</td>
<td>196.16</td>
<td>56.3</td>
</tr>
<tr>
<td>(2^6)</td>
<td>195.37</td>
<td>56.06</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>(2^{17})</td>
<td>193.12</td>
<td>56.15</td>
</tr>
<tr>
<td>(2^{18})</td>
<td>188.83</td>
<td>55.26</td>
</tr>
<tr>
<td>(2^{19})</td>
<td>180.37</td>
<td>56.29</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>142.55</td>
<td>54.31</td>
</tr>
<tr>
<td>(2^{21})</td>
<td>69.26</td>
<td>54.19</td>
</tr>
<tr>
<td>(2^{22})</td>
<td>64.58</td>
<td>48.94</td>
</tr>
<tr>
<td>(2^{23})</td>
<td>63.04</td>
<td>45.72</td>
</tr>
<tr>
<td>(2^{24})</td>
<td>63.17</td>
<td>44.89</td>
</tr>
</tbody>
</table>

We can see that `put` times are very stable until we get close to the 1 million mark, at which point they decrease dramatically, and stabilize after that. This is due to the Hashmap rehashing many times. For initial capacities of \(2^{21}\) or greater, no rehashing ever occurs. Setting initial capacity appropriately is very important to the running time of HashMaps.

You may have noticed that `get` times decreased after the \(2^{21}\) point as well. This is because the table is sparse so there are less collisions on average.

Next we look at varying load factor, with an initial capacity of 16 (default).

\(^2\)There is a good discussion of Python’s implementation [here](#).
We can see here why HashMap has a default load factor of 0.75. Load factors smaller than 0.75 offer insignificant get improvements, while causing lots of table space to be wasted. On the other hand, load factors greater than 0.75 do in fact cause a slow down of get operations.

**Worst Case Linear Time**

With the correct load factor and initial capacity, we can observe the expected constant time access of HashMaps. However, for any hash function, there exists a set of inputs for which performance degrades to linear time. The following table shows inserting into a HashMap a set of values that are known to collide.

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>Time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^1$</td>
<td>4</td>
</tr>
<tr>
<td>$2^2$</td>
<td>9</td>
</tr>
<tr>
<td>$2^3$</td>
<td>19</td>
</tr>
<tr>
<td>$2^4$</td>
<td>36</td>
</tr>
<tr>
<td>$2^5$</td>
<td>73</td>
</tr>
<tr>
<td>$2^6$</td>
<td>155</td>
</tr>
<tr>
<td>$2^7$</td>
<td>296</td>
</tr>
<tr>
<td>$2^8$</td>
<td>742</td>
</tr>
<tr>
<td>$2^9$</td>
<td>1617</td>
</tr>
</tbody>
</table>

As mentioned before, the way to circumvent this is to pick a random hash function every time a new HashMap is created. The latest version of Java implements this.

**Discussion**

**Chaining vs. Open Addressing**

Compare chaining and open addressing and discuss the pros and cons of each.

**Methods of chaining**

What effects would it have if, instead of chaining with linked lists, you used balanced binary search trees? What about if you used hash tables (in a sort of recursive fashion)?

**Methods of Open Addressing**

Compare linear probing and double hashing. What problems do each suffer from? Which is better?

**Problems**

**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.
Problem 2

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

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

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?

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?

Auxiliary proofs

**Theorem 1.** In a hash table where collisions are resolved by separate chaining with newly colliding elements added to the front of the chain, the average number of elements examined in a successful search is $\Theta(1 + \alpha)$, where $\alpha = n/m$ is the load factor, $n$ is the number of elements in the hash table, and $m$ is the number of buckets.

**Proof.** Let $k_1, k_2, \ldots, k_n$ be the elements of the hash table in order of insertion. Assume that for any particular search, any element is equally likely to be the element we are attempting to look up.

Let $Z$ be the random variable representing the number of elements examined in a successful search. If we compute $E[Z]$, we are done.

Define an indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j), \\ 0 & \text{otherwise}. \end{cases}$$

By simple uniform hashing, $Pr[h(k_i) = h(k_j)] = \frac{1}{m}$, so

$$E[X_{ij}] = Pr[X_{ij} = 1] = Pr[h(k_i) = h(k_j)] = \frac{1}{m}.$$

Recall that colliding elements are added to the front of a chain. An element $k_i$ only has $k_{i+1}, k_{i+2}, \ldots, k_n$ in front of it in the chain. Then the number of elements that are examined to find to find a particular $k_i$ (including considering $k_i$ itself) is simply

$$Z_{k_i} = 1 + \sum_{j=i+1}^{n} X_{ij}.$$
In expectation, we have

\[ E[Z_i] = E \left[ 1 + \sum_{j=i+1}^{n} X_{ij} \right] \]

\[ = 1 + \sum_{j=i+1}^{n} E[X_{ij}] \]

\[ = 1 + \sum_{j=i+1}^{n} \frac{1}{m} \]

\[ = 1 + \frac{n - i}{m} \]

Since each \( k_i \) is equally likely to be searched for, by the law of total expectation,

\[ E[Z] = \sum_{i=1}^{n} \frac{1}{n} E[Z | k_i] \]

\[ = \sum_{i=1}^{n} \frac{1}{n} \left( 1 + \frac{n - i}{m} \right) \]

\[ = \sum_{i=1}^{n} \left( \frac{1}{n} + \frac{1}{m} - \frac{i}{nm} \right) \]

\[ = 1 + \frac{n}{m} - \frac{1}{nm} \sum_{i=1}^{n} i \]

\[ = 1 + \frac{n}{m} - \frac{1}{nm} \cdot \frac{n(n + 1)}{2} \]

\[ = 1 + \frac{n}{m} - \frac{n + 1}{2m} \]

\[ = 1 + \frac{n - 1}{2m} \]

\[ = 1 + \frac{\alpha}{2m} - \frac{1}{2m} \]

\[ = 1 + \frac{\alpha}{2} - \frac{\alpha}{2nm} \]

\[ = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}, \]

so \( E[Z] = \Theta(1 + \alpha). \)

\[ \square \]

**Theorem 2.** In a hash table where collisions are resolved by separate chaining with newly colliding elements added to the front of the chain, the average number of elements examined in an unsuccessful search is \( \Theta(1 + \alpha) \), where \( \alpha = n/m \) is the load factor, \( n \) is the number of elements in the hash table, and \( m \) is the number of buckets.

**Proof.** Under the assumption of simple uniform hashing, any key \( k \) not already stored in the table is equally likely to hash to any of the \( m \) slots. The expected time to search unsuccessfully for a key \( k \) is the expected time to search to the end of list. The expected length of the chain is exactly equal to the load factor \( \alpha \), so the the total time required is \( \Theta(1 + \alpha) \).

\[ \square \]

**Theorem 3.** In a hash table where collisions are resolved by open addressing, the average number of elements examined in an unsuccessful search is \( O \left( \frac{1}{1-\alpha} \right) \).

**Proof.** We can show this by writing the expected number of probes as a recurrence on the two variables \( m \) and \( n \):

\[ E[T(m, n)] = 1 + \frac{n}{m} E[T(m - 1, n - 1)] \]
The $\frac{n}{m}$ term arises from the expected number of collisions.

We can prove using induction that $E[T(m, n)] \leq \frac{m}{m-n} = \frac{1}{1-\alpha}$.

Base Case: The base case arises at $T(m, 0)$, which means that the hash table is empty. Thus the first probe always finds an empty element, so $T(m, 0) = 1$.

Induction hypothesis: $E[T(m - 1, n - 1)] \leq \frac{m-1}{m-n}$.

Induction step:

$$E[T(m, n)] = 1 + \frac{n}{m} E[T(m - 1, n - 1)]$$

$$\leq 1 + \frac{n}{m} \cdot \frac{m-1}{m-n}$$

$$< 1 + \frac{n}{m} \cdot \frac{m}{m-n}$$

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

$$= \frac{m}{m-n} = \frac{1}{1-\alpha}$$

$\therefore E[T(m, n)] \leq \frac{1}{1-\alpha}$