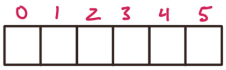
Hashing

Technology is used millions of times a day to quickly *access* specific items from a large set. This happens every time a password is accessed, or a word is looked up in a spell-checker, or when you search for the PIN number of a particular account holder. In most of these instances, we need to quickly *access* an element, and also occasionally *insert* or *delete* new items. For example, suppose we have a system that stores student IDs. We would likely need to look-up a student, and perhaps add or remove students. In this lecture, we look at data structures that can *efficiently* carry out exactly these operations:

- **Look-up**: Search of a specific item and return true if the item exists in the set
- **Insert**: a new item into the set
- **Delete**: find a current item and delete it from the set.

Basic data structures such as linked-lists and arrays can certainly store data. Let's begin by looking at how long it would take to use these primitive structures to hold our data and carry out our operations.

1 How good are arrays and linked-lists?

Arrays: 

Suppose the elements we need to store are held in an array. How would we then carry out a look-up or an insert/delete? If the array is completely unsorted, then locating a specific ID would mean searching through the entire array, which takes time $O(n)$. On the other hand, if we maintained a sorted array, then finding an element would be faster (using *Binary search* this takes time $O(\log n)$) but it would mean that each time we inserted or deleted an element we would have to do it in such a way that keeps the array sorted. The insert and delete costs of an array are both $O(n)$, since we may have to create new space in the array during an insert (and copy all elements over to a new larger array) and during a delete we have to shift all elements over to fill-in the newly made free-spot.

The following chart shows the cost of searching, inserting and deleting from an array in both the sorted and unsorted arrays:

Operation:	Sorted array	Unsorted array
Search:	$O(\log n)$	$O(n)$
Insert:	$O(n)$	$O(n)$
Delete:	$O(n)$	$O(n)$

Linked-Lists: 

One downfall of arrays is that they have a constrained size, whereas linked-lists can easily grow and shrink. If our linked-list is unsorted, then we can insert in constant time by simply adding the new element to the front of the list. However, a delete operation still requires $O(n)$ time since we would need to first search through the list in order to find the element to be deleted.

Let's summarize how linked-lists perform:

Operation:	Sorted list	Unsorted list
Search:	$O(n)$	$O(n)$
Insert:	$O(n)$	$O(1)$
Delete:	$O(n)$	$O(n)$

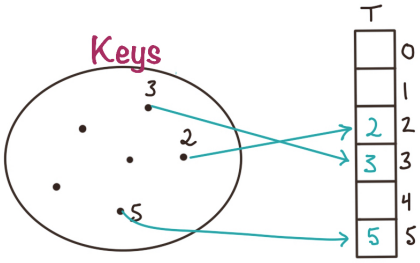
As we can see above, neither the linked-list nor the array implementation are particularly effective when searching for an element. The draw-back lies primarily in the fact that we cannot quickly locate the position of an element in an array. What if we organized the array so that each element was placed at a *specific position*? In this case, we would be able to *immediately* find our element in the array, and our search time would be reduced to $O(1)$. The next section looks at this improvement.

2 Direct Addressing

Suppose we knew from the get-go that we only need to store items that were in a specific range. We could set up the array so that elements are placed in **a specific position** - making our searching process quick and easy. An array used for this purpose is referred to as a **Direct-Address Table**, and it is used when we know that the elements we want to store are drawn from the set:

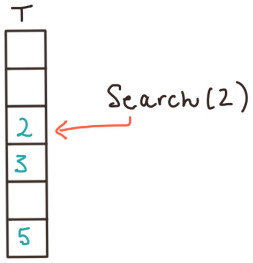
$$\{0, 1, 2, \ldots, m-1\}$$

We place each element in its corresponding position in the table. For example, element 3 is stored in position 3 of the array, and element 2 in position 2, etc.

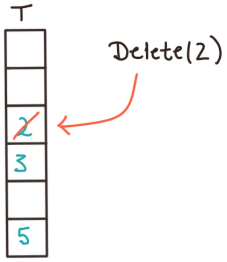


In general then, we **insert** element k at position $T(k)$ by simply setting $T(k) = k$. This takes **constant time**, which is a huge improvement over the performance we saw in the previous section. Similarly, we can search for an element in constant time and delete and element in **constant time** as follows:

Search k**:**. $ReturnT(k)$



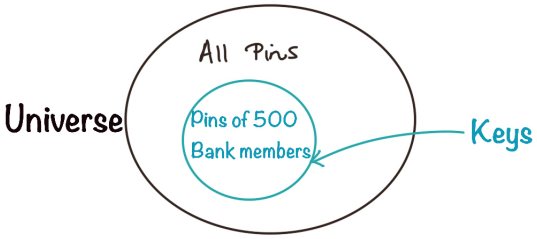
Delete k**:**. $T(k) = \emptyset$



The direct addressing technique performs all three operations in **constant time**. However its fundamental drawback is that the table T must be of size m. In certain instances this becomes a huge waste of space. Suppose we are storing the PIN numbers of bank cards, where each PIN number is exactly 6 digits. The largest such pin number would then be 999999. Does this mean we need a table of all possible numbers up to 999999? What if there were only 500 account holders at the bank? It seems that having a table with all numbers up to 999999 is not efficient when there are significantly less items to store. In the next section, we introduce **Hashing** whereby by reduce the size of the table that is needed.

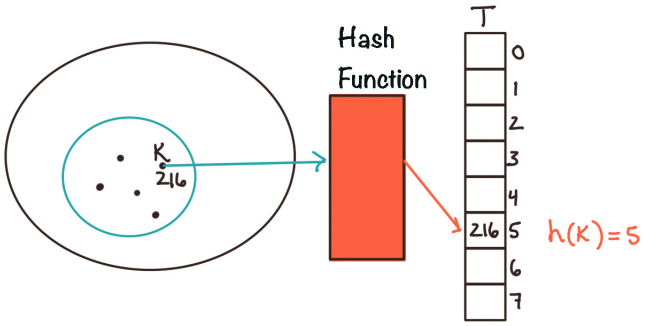
3 Hash Tables

Before getting started, let's formalize a few definitions related to Hash Tables. Typically we use the word **keys**, k, to refer to the specific items that need to be stored. The **Universe,** U is the word we use for all *possible* keys that could exist. From the banking example above, the *universe* of PIN numbers is all possible 6-digit pins. However, there are only 500 accounts at the bank, therefore the set of keys is exactly those 500 PIN numbers.



The main advantage of the hash table is that it takes advantage situations where there are fewer keys than elements in U. In this case, the table used in hashing is typically *much smaller* than the size of the universe.

In order to map the elements of U over to the table T, we need some kind of tool to let us know "*where to look*" so to speak. This is carried out by the Hash Function:



Suppose the array has size m. The hash function $h(k)$ takes *in* a key, k and *outputs* a position in the range $\{0, 1, 2, \ldots, m-1\}$

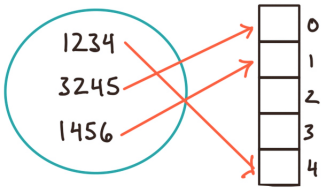
How does this effect our operations search/insert/delete?

- To **search** for a key k, we simply go to position $h(k)$ in the table. This takes time $O(1)$.
- To **delete** a key k, we go to position $h(k)$ in the table, and replace the value with $\emptyset$. This takes time $O(1)$.
- To **insert** a key k, we go to position $h(k)$ in the table, and set the value in T to k. This takes time $O(1)$.

Thus the three operations are all carried out in **constant time**, and this procedure uses much **less space** than direct addressing.

Example 1. *Determine the location in the hash table of the following keys:* 1234, 3245 *and* 1456 *under the hash function defined by* $h(k) = k \mod 5$.

Solution: Note that $h(1234) = 1234 \mod 5 = 4$, so we expect to find key 1234 in position 4 of the table. Similarly for the next two keys, $h(3245) = 0$ and $h(1456) = 1$. The hash table would then look like:



Example 2. . *Suppose we build a hash table to contain the last names of students in a class. The hash function takes the last name of the student and uses the first letter to map to a position in the table* T *in the range* $\{1, 2, \ldots, 26\}$. *Describe the hash table for the names McDonald, Jamieson, and Everest.*

Solution: The hash table has size 26. The names McDonald, Jamieson, and Everest are placed in position 13, 10 and 5 respectively.

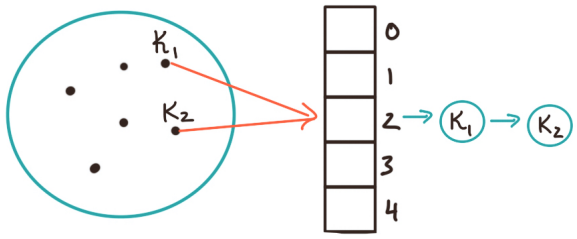
There are clearly several drawbacks with this method.

- **There may be significant wasted space**. In the above example regarding last names, there may be a significant amount of wasted space if for example no last names begin with V or Z or X.
- **There may be collisions**. In the above example, there may be several last names that start with the same letter - meaning they would be inserted into the same spot.

In the next sections, we look different options on how to resolve the problem of collisions.

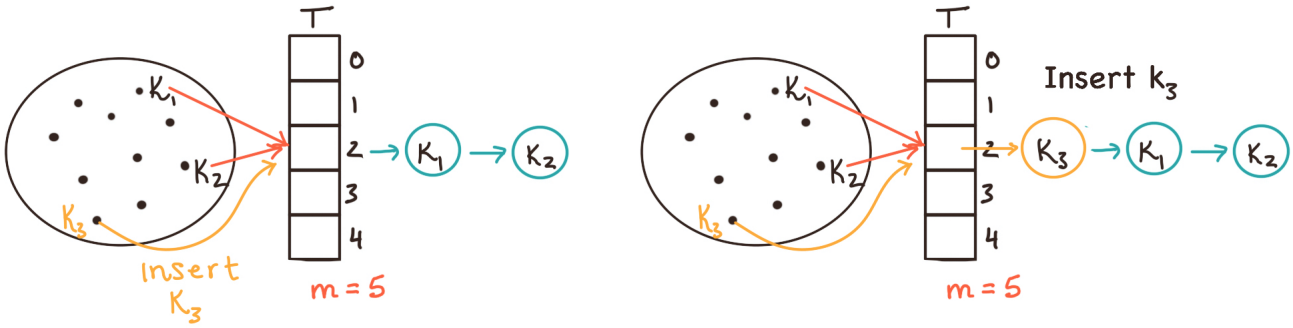
4 Hashing with Chaining

Suppose two keys are mapped to the same position in the hash table. A simple way to resolve the collisions is to allow *both* keys to occupy the same position in the table, by *linking* them together in a chain:

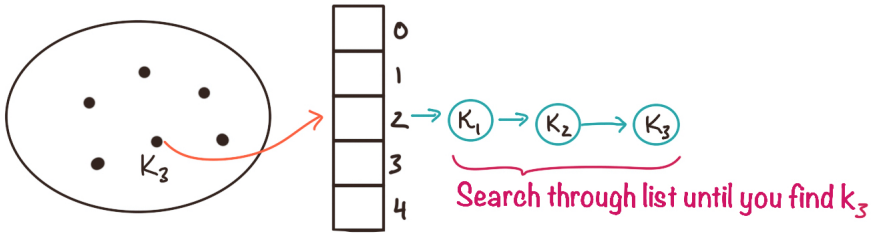


How would this chaining effect the cost of our operations? Although the chaining resolves the problem of collisions, it is possible that many keys get sent to the same spot in the table and we end up with long chains. The operations work as follows:

- To **insert** a key k, we locate position $h(k)$ in the table, and then simply attach the new element to the chain. This can be done either at the front of the linked-list, or at the back (if we store a pointer to the last element of the linked-list). This takes time $O(1)$



- To **search** for a key k, we locate position $h(k)$ in the table, and then **search** through the chain to locate our key. This could take time $O(n)$ in the worst-case, if the chain has length n.



- To **delete** a key k, we locate position $h(k)$ in the table, and then we need to **search** through the chain to locate our key and delete it. This could take time $O(n)$ if the chain has length n.

Unfortunately, as we see above, if there are long chains which contain almost all elements, the search and delete operations are now much slower.

How could we improve this problem? It seems reasonable to expect to able to *equally spread out* the n keys amongst the m slots in the table, thus avoiding chains of length n in one slot.

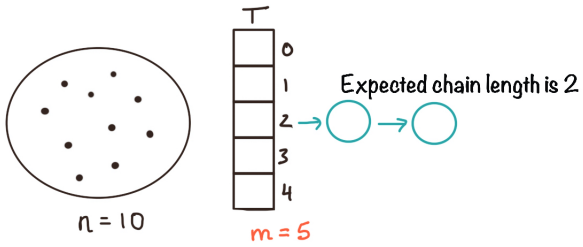
4.1 Uniform hashing

If we would like to "*spread out*" our n keys over the m spots of the table, then ideally we would like:

$$\frac{n}{m} \text{ keys per slot}$$

This value is called the **load factor** of the table, and is denoted α. If this were achievable, then each chain would have average length $\alpha = \frac{n}{m}$. For example, suppose we have $n = 10$ keys and a hash table of size $m = 5$. Then the load factor is $\alpha = \frac{10}{5} = 2$, which means the *average* length of each chain is 2. Ideally we would like each chain length to be as close to the average as possible. If this were achievable, then we would have fast search and delete runtimes because we avoid having long chains of length n.

How do we achieve this? What kind of hashing function will result in a load factor where the keys are spread out in the table? Such functions $h(k)$ are called **uniform hash functions**, because they are **equally likely** to hash key k to any of the m spots in the table. The expected length of a chain under uniform hashing is exactly $\alpha = \frac{n}{m}$.



What is the cost of hashing under this new improvement?

- To **insert** a key k, we locate position $h(k)$ in the table, and then simply attach the new element to the chain. This takes time $O(1)$
- To **search** for a key k, we locate position $h(k)$ in the table, (which takes time $O(1)$) and then **search** through the chain to locate our key. For the chain of length α, this takes time $O(\alpha)$ in the worst-case. Thus a search takes time
$$O(\alpha + 1)$$
- To **delete** a key k, we locate position $h(k)$ in the table, (which takes time $O(1)$) and then **search** through the chain to locate our key and delete it. For the chain of length α, this takes time $O(\alpha)$ in the worst-case. Thus a delete takes time
$$O(\alpha + 1)$$

What can we conclude from the above runtimes, and how can we compare this to the usual hashing with chaining? The results depend on the value of α. Suppose that α is **constant**. Then in fact our new hashing with chaining performs **all three operations in constant time** and **resolves collisions**. For example, for n keys, we could use a table of size $n/10$. Then the load factor would be $\alpha = n/(n/10) = 10$, which means the expected length of each chain is 10. The expected time for the search and delete operations would be $O(10 + 1) = O(1)$ in this case.

In the next section, we look at a final approach that resolves collisions and also minimizes the amount of wasted space in the table.

5 Hashing with Open Addressing

In hashing with open addressing, we do not build chains off the hash table. Instead, we resolve collisions in a rather interesting way. Suppose that you use the hash function to locate a specific slot in the table, and unfortunately that slot is full. In this new technique, we simply **probe** to a second location in the table. Again, if this spot is full, we probe to a third location. This process continues until we find a free spot and are able to insert our key:

The sequence that we follow while searching through the table is called a probe sequence. The probe sequence is simply some permutation of all the possible slots in the table. An important fact is the following:

Each key has its own specific probe sequence

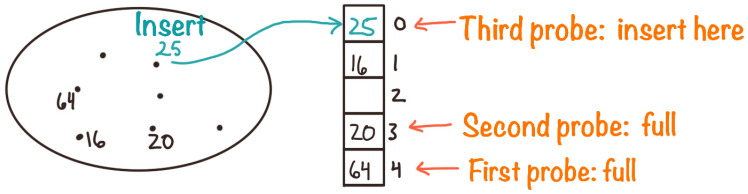
As an example, suppose that we have a hash table of size 5 and the particular key we wish to insert is 25. We have a corresponding *probe sequence* specific to key 25. For example,

$$h(25) = \{4, 3, 0, 2, 1\}$$

and we denote the individual terms of this sequences as:

$$h(25, 0) = 4, h(25, 1) = 3, h(25, 2) = 0, h(25, 3) = 2, h(25, 4) = 1$$

Thus in order to insert key 25, we would first try to insert at position 4, and if full, next look at position 3, etc, until we find a free spot.



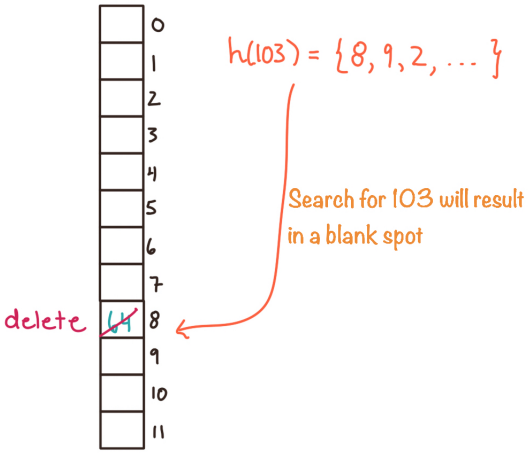
This probe sequence is denoted in general as $h(k, i)$ where k is the key and i is the "*probe attempt*".

The search process works in the same way. If we wish to search for key 25, we first look at position 4, then position 3, etc, until we either *find* key 25 or we located an empty spot.

5.1 Deletion:

The deletion operation in open addressing is slightly more complicated. What would happen if we simply searched for the key we wish to delete, and when we find it we delete it from the table? This is how we carried out deletion in the Direct-Addressing Table. In open addressing however, this causes a problem.

Suppose $h(64) = \{3, 2, 8, \ldots\}$ and when 64 was inserted, spots 3 and 2 were already filled. Key $k = 64$ would then get inserted in spot 8. Now suppose we delete key $k = 64$ from the table. By doing so, we leave an empty spot at position 8.
What if 8 was part of the probe sequence of another *key*? The example on the right shows the probe sequence for key 103, in the case where 103 was inserted at position 9. What happens if we carry out a search for key 103? We will look at spot 8, find an empty spot, and then assume that 103 is not in the table!



Suppose we try to fix this problem by filling this empty spot with some kind of fancy marker that means "*keep looking*". This solution quickly becomes inefficient and fills up the table with many blank markers. An example of this is included in the practice problems.

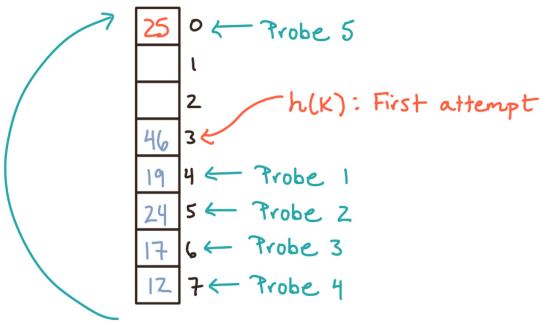
5.2 Types of probe sequences

There are several different types of probe sequences possible.

5.2.1 Linear probing:

Linear probing uses a usual hash function $h(k)$ in order to find the *first* possible spot in the hash table. After that it probes subsequent spots by simply increasing to the next position. When it reaches the end of the table, it bounces to the top of the table.

For example, suppose $h(25) = 3$. Then we would first try to insert 25 at position 3 and if full, we would next go to 4, then 5, then 6, etc, until we find a free spot. If we hit the end of the table first, we bounce up to the top.



One problem with this technique is that is tends to create clusters of keys in the table, separated by several free spots. This unfortunately slows down the search process, since every time we hit an occupied spot we must search through the whole cluster until we find a free spot.

5.2.2 Quadratic probe sequence:

A quadratic probe sequence is like linear probing in the sense that it uses a hash function $h(k)$ for the initial spot in the table. If there is a collision, it probes new spots using the sequence:

$$h(k, i) = h(k) + ai + bi^2 \mod m$$

where a and b are any constants. For example, suppose $h(k) = k \mod 5$, and suppose we pick $a = 1$ and $b = 1$. Then the probe sequence is given by $h(k, i) = k + i + i^2 \mod 5$. If we insert $k = 12$, then $h(12) = 2$, so the initial spot we try is 2. If there is a collision, the next spot would be $h(12, 1) = 12+1+1^2 \mod 5 = 4$. If this also results in a collision, next we try $h(12, 2) = 12 + 2 + 2^2 = 3$, etc.

5.2.3 Double hashing:

In double hashing, we use **two separate hash functions**: $h_1(k)$ and $h_2(k)$. The first function is used to find the first position in the table, and the second function is used to provide the **offset** by which we jump in the table to find the next free spot. For example, if $h_1(25) = 3$ and $h_2(25) = 2$, then we first try to insert at position 3 and if it's full, we jump next to position $3 + 2 = 5$, then to position $3 + 2 * 2 = 7$ etc until we find a free spot. As above, if we hit the end of the table, we bounce back to the top.

5.3 Analysis of Open-Addressing

In this last section, we analyze the performance of open-addressing. Recall from the previous section that if we have n **keys** and our **table size is** m then the load factor is $\alpha = \frac{n}{m}$ and it represents how "*full*" the table is. In open addressing, we cannot have more keys than fit in the table, so $n \leq m$ and thus $\alpha \leq 1$. The following theorem tells how how many probes are expected in our open-addressing solution. This theorem *assumes* that the probe sequence used is a uniform hash sequence, in other words that each probe sequence is equally likely.

Theorem 1. *For an open-address hash table where $\alpha < 1$, the expected number of probes during a search is at most*

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

The worst-case scenario for the number of probes corresponds to an **unsuccessful** search. If we have a **successful** search, we would have stopped the searching process sooner, and thus the expected number of probes is smaller in this case.

Example 3. *Using open-addressing uniform hashing, where the number of keys is half the size of the table, what is the expected number of probes in a search?*

Solution:. Since $m = 2n$, we have that $\alpha = \frac{1}{2}$. Thus the expected number of probes is

$$\frac{1}{1-\alpha} = \frac{1}{\frac{1}{2}} = 2$$