Collision Resolution: Closed Addressing (Separate Chaining)

Step 1

In the previous section, we discussed our first attempt at handling collisions using a technique called **Linear Probing**, where, if a key maps to a slot in our **Hash Table** that is already occupied by a different key, we simply slide over and try again; if that slot is also full, we slide over again; etc. On **average**, assuming we've designed our **Hash Table** intelligently, we experience **O(1)** find, insert, and remove operations with **Linear Probing**. However, in the **worst case**, we could theoretically have to iterate over all *N* elements to find a key of interest, making our three operations **O(*N*)**.

Also, we saw that keys tend to form "clumps" that make probing excessively slow, and these clumps are probabilistically favored to grow. We introduced **Double Hashing** and **Random Hashing**, which helped solve our issue of clump formation and helped speed things up a bit.

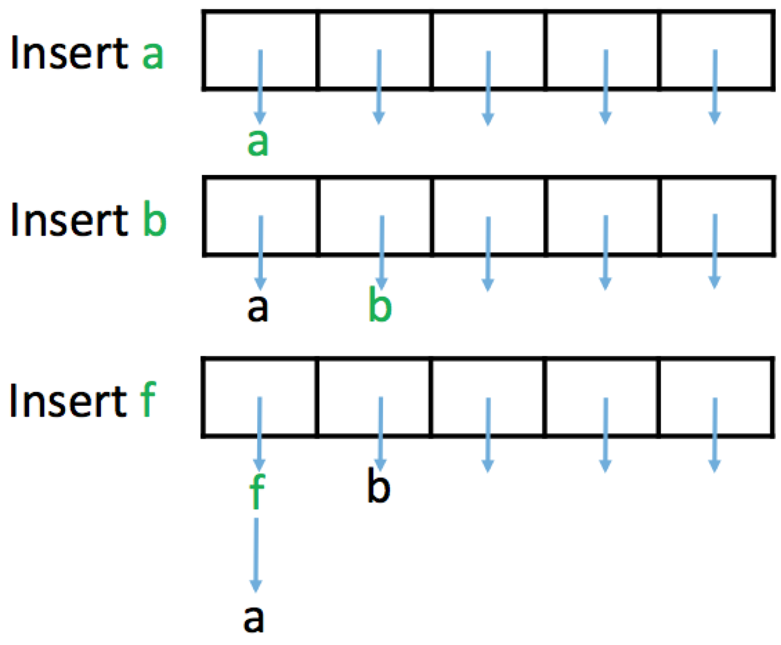
Recall that, if we try to insert a key and *don't* encounter a collision, the probability of subsequent collisions must increase, regardless of how we choose to handle collisions (because we have now filled a previously-unfilled slot of our **Hash Table**). If we try to insert a key and we *do* encounter a collision, with **Linear Probing**, we take a performance hit on two accounts: we have to linearly scan until we find an open slot for insertion (so we take a hit now), and because we've now filled a previously-unfilled slot of our **Hash Table**, we have yet again increased the probability of subsequent collisions (so we will take a hit in a future insertion).

Upon a collision, the current insertion will take a performance hit, no matter which collision resolution strategy we choose. In other words, the performance hit we take now is inevitable. However, in this section, we will discuss another collision resolution strategy **Separate Chaining**, in which collisions *do not* increase the probability of future collisions, so we avoid taking the additional performance hit in the future.

Step 2

The next collision resolution strategy that we will discuss is called **Separate Chaining**. The basic idea behind **Separate Chaining** is to keep pointers to Linked Lists as the keys in our **Hash Table**.

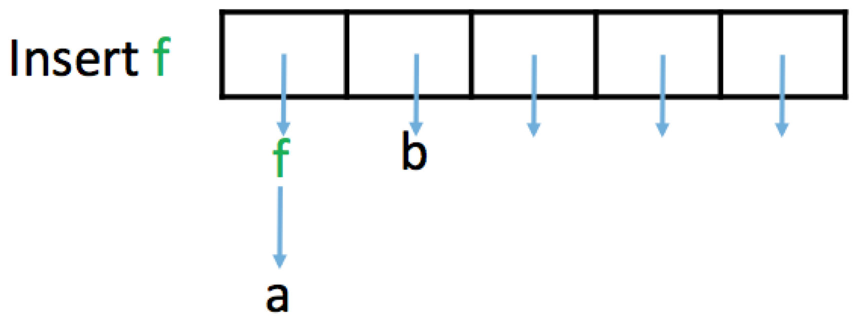
Here is an intuitive example using **Separate Chaining** in which the **hash function**, the same we used in the previous lesson, is defined as $H(\boldsymbol{k}) = ((\boldsymbol{k}+3)\ \%\ \boldsymbol{m})$, where ***k*** is the ASCII value of the key and ***m*** is the size of the backing array (we add 3 to ***k*** to have the letter **a** hash to index 0 for simplicity):



STOP and Think: Can the **Hash Table** above ever get full?

Step 3

EXERCISE BREAK: What is the probability of a collision for inserting a **new** arbitrary key in the **Hash Table** below (the same **Hash Table** *after* we inserted the key 'f' in the previous step)? Input your answer in decimal form and round to the nearest thousandth.



STOP and Think: Is the probability larger or smaller than it was when we used **Linear Probing**?

To solve this problem please visit https://stepik.org/lesson/31225/step/3

Step 4

How do we go about implementing this collision resolution strategy? Below is pseudocode to implement **Separate Chaining** when inserting a key `k` into a **Hash Table** of capacity `M` with a backing array called `arr`, using a **hash function** `H(k)`:

```
insert_SeperateChaining(k): // Insert k using Separate Chaining for collision resolution

    index = H(k)

    // check for duplicate insertions (not allowed) and perform insertion
    if Linked List in arr[index] does not contain k:
        insert k into Linked List at arr[index]
    // resize backing array
    if n/m > loadFactorThreshold:
        arr2 = new array with a size ~2 times the size of arr that is prime  // new backing array
        insert all elements from arr into arr2 using normal insert algorithm // rehash all
elements
        arr = arr2                                                           // replace arr with
arr2
```

Notice that the core of the **Separate Chaining** algorithm is defined by this line:

```
insert k into Linked List at arr[index]
```

As you might have noticed, we have been pretty vague as to which implementation of a Linked List (Singly-Linked, Doubly-Linked, etc.) to use because it is really up to the person implementing the **Hash Table** to decide exactly which Linked List implementation to choose based on his or her needs. Just like we have been discussing throughout this entire text, we can get vastly different time complexities based on which backing implementation we use and *how* we chose to use it.

A common implementation choice is to use a Singly-Linked List with just a head pointer. By choosing that implementation, we consequently add new elements to the *front* of our Linked List (i.e., reassign the head pointer and the next pointer of the new element) as opposed to the back of the Linked List (i.e., traverse the entire Linked List and then reassign the next pointer of the last element). Note that, if we do not check for duplicate insertions, inserting at the *front* of a singly-linked list produces a worst case *constant* time complexity and inserting at the *back* of a singly-linked list turns into a worst case *linear* time complexity.

By obeying the algorithm above, we are ensuring that our key is inserted strictly within the *single* index (or more formally, "address") to which it originally hashed. Consequently, we call this a **closed addressing** collision resolution strategy (i.e. the key *must* be located in the original address). Moreover, since we are now allowing multiple keys to be at a single index, we like to say that **Separate Chaining** is an **open hashing** collision resolution strategy (i.e., the keys do not necessarily need to be physically inside the Hash Table itself).

Note: You will most likely encounter people using the terms **open hashing** and **closed addressing** interchangeably since they arguably describe the same method of collision resolution.

Step 5

EXERCISE BREAK: Consider a **Hash Table** with 100 slots. Collisions are resolved using **Separate Chaining**. Assuming that there is an equal probability of mapping to any index of the **Hash Table**, what is the probability that the first 3 slots (index 0, index 1, and index 2) are unfilled after the first 3 insertions? Enter your answer as a decimal (i.e., not a percentage) and round to the nearest thousandth.

To solve this problem please visit https://stepik.org/lesson/31225/step/5

Step 6

We briefly mentioned in the previous step that, by not checking for duplicates in the **insert** method, we could get a worst-case *constant* time complexity. So how does checking for duplicates even work in the first place? We can check for duplicate insertions using two different methods:

1. Check for duplicates during the **insert** operation:
   - If we wanted to check for duplicates as we insert a key, we would calculate the index pertaining to the inserting key—using the key's hash function—and then *linearly scan* the *entire* linked list at the key's index to check if the same key is present. If the key is present, then we would abort the insert. Otherwise, we would proceed to insert the key. By checking for duplicates in the insert method, we would face a guaranteed worst-case *linear* time complexity as a result of the linear scan we are forced to do.
2. Check for duplicates during the **delete** operation:
   - If we wanted to check for duplicates inside the **delete** operation, we would not have to check for duplicates inside the insert operation (i.e., we would allow for duplicates to appear during insertion). Instead, once the delete operation is called on a key, we would calculate the index pertaining to the inserting key—using the key's hash function—and then *linearly scan* the *entire* linked list at the key's index to remove *all* instances of the key. This would result in a worst-case *linear* time complexity.

So why would we even choose to check for duplicates inside the **delete** operation to begin with? Notice that by doing so, we eliminate the need to check for duplicates inside the **insert** operation. Consequently, we are able to avoid having to linearly scan inside the **insert** method and therefore only have to do a single operation to insert a key at the front of a linked list. The **insert** operation now becomes a worst-case constant time complexity operation! Note that our **delete** operation originally (without having to check for duplicates) had a worst-case *linear* time complexity because worst case scenario, we would have had to traverse an entire linked list to find the element to delete. By checking for duplicates during **delete**, our worst-case time complexity actually still stays the same.

Step 7

EXERCISE BREAK: Consider a **Hash Table** in which collisions are resolved using **Separate Chaining**. Select the tightest worst-case time complexity for an insert function that uses a Doubly-Linked List (i.e., there are head and tail pointers, and you can traverse the list in either direction) and inserts keys to the end of the Linked Lists. This insert function does not allow duplicates.

To solve this problem please visit https://stepik.org/lesson/31225/step/7

Step 8

How would we go about implementing the remove and find functions for a **Hash Table** that uses **Separate Chaining**? The algorithm is actually quite simple: hash to the correct index of the **Hash Table**, and then search for the item in the respective Linked List. Note that this remove algorithm is *much* easier than the remove algorithm we had to use in **Linear Probing**.

It is also important to note that **Separate Chaining** does not necessarily *have* to use Linked Lists! A **Hash Table** can have slots that point to AVL Trees, Red-Black Trees, etc., in which case the worst-case time complexity to find an item would definitely be faster. However, the reason why we use Linked Lists is that we do not expect the worst-case time complexity of finding an item in a **Hash Table** slot to make a huge difference. Why? Because if it did, that would mean that the rest of our **Hash Table** is performing poorly and that would imply that we should probably investigate why that might be the case and fix it (perhaps the hash function isn't good or the load factor has become too high).

Step 9

So what are the advantages and disadvantages of **Separate Chaining**?

In general, the average-case performance is considered much better than **Linear Probing** and **Double Hashing** as the amount of keys approaches, and even *exceeds*, *M*, the capacity of the **Hash Table** (though this proof is beyond the scope of the text, you can find it here). This is because the probability of *future* collisions does not increase each time an inserting key faces a collision with the use of **Separate Chaining**: in the first Exercise Break of this lesson *and* the previous lesson we saw that inserting the key 'f' into a **Hash Table** that uses **Separate Chaining** kept the probability of a new collision at 0.4 as opposed to increasing it to 0.6 as with **Linear Probing**. It is also important to note that we could never have exceeded *M* in **Linear Probing** or **Double Hashing** (not that we would want to in the first place) without having to resize the backing array and reinsert the elements from scratch

A disadvantage for **Separate Chaining**, however, is that we are now dealing with a bunch of pointers. As a result, we lose some optimization in regards to memory for two reasons:

1. We require extra storage now for pointers (when storing primitive data types).
2. All the data in our **Hash Table** is no longer huddled near 1 memory location (since pointers can point to memory anywhere), and as a result, this poor locality causes poor cache performance (i.e., it often takes the computer longer to find data that isn't located near previously accessed data)

Step 10

As we have now seen, **Separate Chaining** is a **closed addressing** collision resolution strategy that takes advantage of other data structures (such as a linked list) to store multiple keys in one **Hash Table** slot. By storing *multiple* keys in *one* **Hash Table** slot, we ensure that the probability of *future* collisions does not increase each time an inserting key faces a collision—something that we saw happen in other **open addressing** methods—thereby giving us more desirable performance.

Nonetheless, we will continue to explore other collision resolution strategies in the next lesson, where we will look at a collision resolution strategy called **Cuckoo Hashing** that not only resolves collisions, but takes extra measures to avoid collisions in the first place.