Hash Tables and Hash Functions

We have seen that with a balanced binary tree we can guarantee $O(\log n)$ worst-case time for insert, search and delete operations. Our challenge now is to try to improve on that ... if we can.

Suppose we have data objects that consist of pairs - each pair having a key value that is unique to the object, and some "satellite" data that may or may not be be unique. For example, the data might consist of "phone number, name" pairs in which each phone number must be unique (the key), but there might be two people both of whom are named "Cholmondeley Featheringstonehaugh Arbuthnot" (duplicate satellite data).

Assuming for the moment that the keys are integers, the simplest method to store the data is in an array that has an address for each possible key - this is called **direct addressing**. Then insert, search and delete all take $O(1)$ time – we can't beat that!

The problem of course is that the set of possible keys may be immense - with 10-digit phone numbers there are $10^{10}$ possible combinations, and most people have at most a couple of hundred phone numbers in their contact list. Even if you have every person in Canada in your contact list, creating an array with $10^{10}$ elements and only using 35,000,000 of them is not very practical.

However, the idea of using a simple array to store our data is very appealing and as we shall see, with a little care and attention we can get good **average-case** complexity for our three essential operations, even though we may not have optimal **worst-case** complexity.

Since we are introducing average-case complexity we should spend a moment looking at balanced binary trees in this way. In a complete binary tree (ie one in which there are no missing children - all the leaves are at the bottom level) - almost exactly half the vertices are at maximum distance from the root. This implies that the average insert/search/delete time is going to be close to $\log n$. We can make the same argument for Red-Black trees.

Since we contemplating using an array that is smaller than the set of all possible keys, we clearly need some way to map key values onto array addresses. For example if our array (which we will call T) has index values 0, 1, ..., 9 and our key value is 34, we need to decide which array address to use to store the data. We call this mapping a **hash function**. We call the array T a **hash table**.
For the rest of this discussion, we will use the following notation consistently:

**m**: the size of T. T has addresses 0, 1, ..., m-1

**n**: the number of data objects we need to be able to store in T. If this is not known precisely, we should at least be able to put an upper limit on it.

**h(k)**: a function that takes a key value as its argument and returns a value in the range [0 .. m-1]

We will spend some time later talking about how to choose h(k), but for now we will assume the keys are non-negative integers and we will use

\[ h(k) = k \mod m \]

as our hash function. So continuing the previous example, with m = 10 and k = 34, we get h(k) = 4

The problem is that since the number of possible keys exceeds m, we may get collisions - two or more keys in our set that hash to the same address. To deal with collisions we need to define a collision resolution method.

Note that due to collisions, we must store the key value as well as its satellite data - otherwise we cannot distinguish between the data associated with different keys that have the same value of h(k)

**Collision Resolution Methods**

The very bad method: If we are trying to insert a value \( k \) and the address \( h(k) \) is already occupied, we simply reject the new data. This has the advantage of making insert/search/delete all \( O(1) \) worst case - but it has the downside that we are frequently unable to successfully insert new values even though there may be a lot of empty space in T.

Another bad method: If we are trying to insert a value \( k \) and the address \( h(k) \) is already occupied, we overwrite \( h(k) \) with the new data. This has \( O(1) \) complexity for all operations, and we are always able to insert a new value. Alas, we are likely to lose a lot of data.

**Challenge**: come up with a situation where you can argue that one of the above methods is useful.
A good method: **Chaining**

In a chained hash table, T is not used to store the data directly. Each element of T is a pointer to the head of a linked list of objects that have all hashed to the same location in T. If no key values currently in the set have hash value = i, then T[i] is a null pointer. Each data pair is implemented as an object that contains the key value, the satellite data, and a pointer variable that will be used to connect to the next object in the list, if any.

Insert: It is always possible to insert a new key k into the hash table. We add the new object at the head of the list attached to its hash value address in T. This gives insertion O(1) complexity

```python
def insert(new_object):
    hash_val = h(new_object.key)
    new_object.next = T[hash_val]
    T[hash_val] = new_object
```

Search is also simple: we go to the hash value address in T and search through the list:

```python
def search(k):
    hash_val = h(k)
    temp = T[hash_val]
    while temp != null and temp.key != k:
        temp = temp.next
    if temp != null:
        return "found it"
    else:
        return "not found"
```

Delete is similar to search - we just need to fix up the pointers in the list

```python
def delete(k):
    hash_val = h(k):
    temp = T[hash_val]
    if temp == null:
        return
    elif temp.key == k:
        T[hash_val] = temp.next
    else:
        previous = temp
        current = temp.next
        while current != null and current.key != k:
            previous = current
            current = current.next
        if current != null:
            previous.next = current.next
```
Since the linked lists can be arbitrarily long, there is no upper limit to the number of values that can be stored in the hash table. But the longer the chains, the longer it will take to search and/or delete.

We will make the uniform hashing assumption: we assume that our hash function $h(k)$ maps key values uniformly onto addresses - that is, each key is equally likely to be hashed to each address. The validity of this assumption depends on a number of factors, including the distribution of the keys, the size of the table, and the hashing function itself - we will return to this issue later but for now we will just make the assumption. In effect, this means that approximately equal number of keys are mapped onto each address in $T$.

With this assumption, the expected number of data objects in each chain is $\frac{n}{m}$ (recall that $n$ is the number of values stored in the table, and $m$ is the size of the table). From this it is possible to show that the expected number of steps in a search (either successful or unsuccessful) is in $O(1 + \frac{n}{m})$ - full details of this proof are in the textbook. Since a delete operation requires only $O(1)$ operations after the object has been found, the expected time for both search and delete is in $O(1 + \frac{n}{m})$.

Now if we require that $n$ is in $O(m)$, we know $n \leq c \cdot m$ for some constant $c$. Then $\frac{n}{m} \leq c$, so $O(\frac{n}{m}) = O(c) = O(1)$, and the expected time for both search and delete is in $O(1)$.

The ratio $\frac{n}{m}$ is called the load factor of the hash table, and we often see the symbol $\alpha$ used for this. If $\alpha$ is high then collisions are more likely.

The downside of chaining is that indirect addressing (the pointers we use to link together the chains) is physically slower than direct addressing. The most popular alternative is to resolve a collision by finding an empty address in the table and storing the new data object there. This is called open addressing.
Open Addressing

We will look at three forms of open addressing: linear probing, quadratic probing, and double hashing.

For open addressing, we change the notation for our hash function to include a second parameter.

\[ h(k,i) \quad === \quad \text{the address to try to store key } k, \text{ when } i \text{ locations have previously been tried for this key} \]

We break \( h(k,i) \) into two independent functions, like this: \( h(k,i) = (h'(k) + f(k,i)) \mod m \)

where

- \( h'(k) \) is a hash function as we used the term before - any function that produces a value in the range \([0...m-1]\)
- \( f(k,i) \) is any function that produces integer values and satisfies the requirement that \( f(k,0) = 0 \)

Note that \( h(k,i) \) includes a final "mod m" to ensure that the value of \( h(k,i) \) is in the range \([0...m-1]\) even if \( f(k,i) \) returns a large value.

Open addressing looks like this: Given a key value \( k \), we first compute \( h(k,0) \). If there is a collision at that address, we compute \( h(k,1) \). If there is a collision there, we try \( h(k,2) \) ... and so on.

The search algorithm for a hash table using open addressing examines exactly the same sequence of addresses as the insert algorithm. The search is successful if the key is found, and unsuccessful if either an empty location is found, or all locations are examined without finding the desired key.

The sequence of addresses examined during any of the three essential operations is called the probe sequence for that key. It should be clear that the probe sequence is completely determined by the key value. Ideally, each probe sequence should contain each address in the hash table exactly once. These probe sequences would be permutations of the set \([0,...,m-1]\) ... so there are \( m! \) possible probe sequences. We will use this fact to compare the three forms of open addressing.
Linear Probing

The idea of linear probing is to resolve collisions by looking at the addresses sequentially following the first address tried. To achieve this we simply let \( f(k, i) = i \)

\[
h(k, i) = (h'(k) + i) \mod m
\]

When computing \( h(k, 0), h(k, 1), h(k, 2) \) etc, the \( h'(k) \) part never changes, so in implementation we compute this once and then just use it over and over.

Note that we need some way to establish that an address is empty - this is typically done by storing an illegal key value in each address where no key has yet been stored. For example if the legal keys are all positive integers, we can use 0 to signify "empty". Since this depends on the actual set of possible keys, I will just use “empty” in the pseudo-code versions of the algorithms.

**Linear_Probing_Insert(k):**

\[
i = 0 \\
v = h'(k) \\
a = v \\
\text{while (i < m) and (T[a] not "empty")} \\
\hspace{1cm} i += 1 \\
\hspace{1cm} a = (v + i) \mod m \\
\text{if T[a] is "empty"} \\
\hspace{1cm} T[a] = k \\
\text{else} \\
\hspace{1cm} \text{report "table full, insert failed"}
\]

**Linear_Probing_Search(k):**

\[
i = 0 \\
v = h'(k) \\
a = v \\
\text{while (i < m) and (T[a] not "empty") and (T[a] != k)} \\
\hspace{1cm} i += 1 \\
\hspace{1cm} a = (v + i) \mod m \\
\text{if T[a] == k} \\
\hspace{1cm} \text{report "found it"} \\
\text{else} \\
\hspace{1cm} \text{report "search failed"}
\]

Easy peasy, but what about deletion? The problem is that if we delete a key value by replacing it by our "empty" flag value, then a subsequent search might give an incorrect result due to hitting this empty spot and stopping, when it should have continued.
Example: Suppose the keys are integers in the range [1 \ldots 20], m = 10, and we decide to use

\[ h'(k) = \left\lfloor \frac{k^2}{7} \right\rfloor \]

(Side note: We discussed in class a reason why \(k^2\) is not a good hash function. Does dividing by 7 and rounding down to an integer make it any better?)

If our first key for insertion is \(k_1 = 13\), the probe sequence is 4, 5, \ldots, 9, 0, 1, 2, 3 and we place the data in \(T[4]\).

If our second key for insertion is \(k_2 = 10\), the probe sequence is again 4, 5, \ldots, 1, 2, 3 and we place the data in \(T[5]\).

Now suppose we delete \(k_1\) and place the “empty” flag in \(T[4]\).

Now, finally, suppose we search for \(k_2\). We look in \(T[4]\), see “empty”, and stop ... even though \(k_2\) is in the table.

We solve this by choosing another flag value to signify "deleted". Again, if the valid keys are all positive integers, we could use -1 as the "deleted" flag. This changes our insert algorithm a bit: \textit{insert} can insert into any address that is either "empty" or "deleted".

\begin{verbatim}
Linear_Probing_Insert(k):
    i = 0
    v = h'(k)
    a = v
    while (i < m) and (T[a] not "empty") and (T[a] not "deleted")
        i += 1
        a = (v + i) % m
    if (T[a] is "empty") or (T[a] is "deleted")
        T[a] = k
    else
        report "table full, insert failed"
\end{verbatim}

The \textit{search} algorithm does not change. Now we can write the \textit{delete} algorithm:
Linear_Probing_Delete(k):
  \[\text{i} = 0\]
  \[v = h'(k)\]
  \[a = v\]
  while \((i < m)\) and \((T[a] \text{ not } \text{"empty"})\) and \((T[a] \neq k)\)
    \[i += 1\]
    \[a = (v + i) \mod m\]
  if \(T[a] = k\)
    \[T[a] = \text{"deleted"}\]
  else
    report \text{"search failed - could not delete"}

Linear probing is quick and easy and it is guaranteed to find an empty address if there is one. Unfortunately it is subject to a phenomenon called **primary clustering** which can negatively affect the expected times for insertion, search and deletion. The problem is that if (for example) 4 consecutive addresses are filled and the next address is empty, the probability that the next address will be filled on the next insert is higher than it should be: any key that hashes to any of the 4 filled addresses will end up in the next one. Thus blocks of consecutive filled addresses tend to get larger and larger, and the number of probes needed to complete any of the three essential operations gets larger too. In the worst case we can end up with \(O(n)\) time for each of the essential operations.