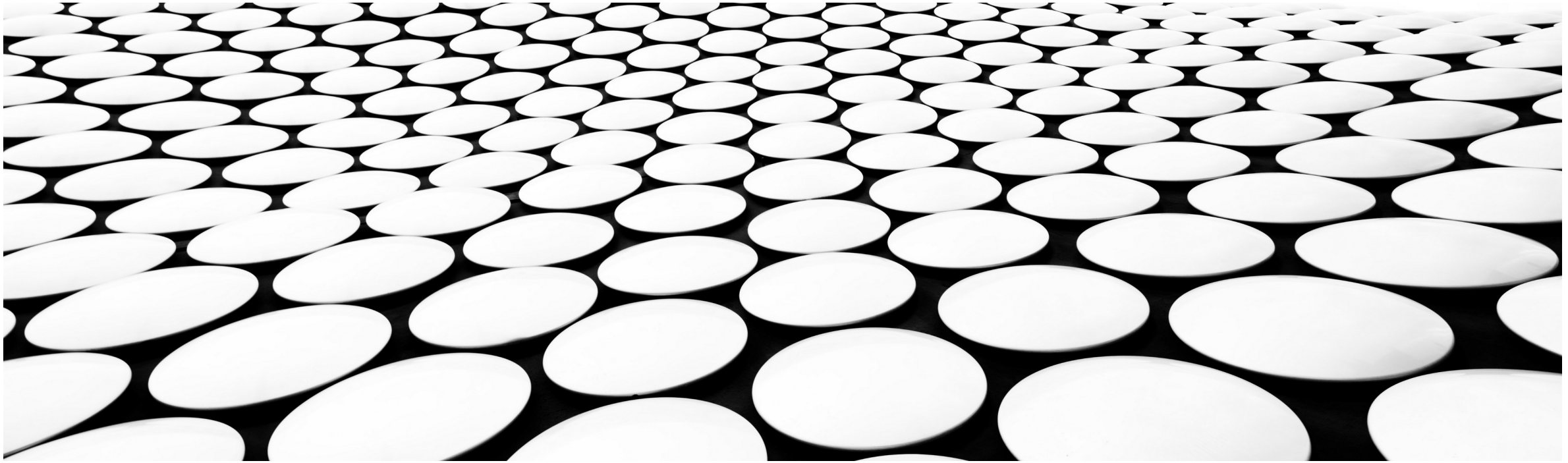
DSA 456 – WEEK 5

TABLES



TABLES

A Table is an ***unordered collection of records***. Each record consists of a key-value pair. Within the same table, keys are unique. That is only one record in the table may have a certain key. Values do not have to be unique.

Table operations

A table supports a subset of the following operators (though sometimes it may be combined in design)

- initialize - Table is initialized to an empty table
- isEmpty - tests if table is empty
- isFull - tests if table is full
- insert - add a new item with key:value to the table
- delete - given a key remove the record with a matching key
- update - given a key:value pair change the record in table with matching key to the value
- find - given a key find the record
- enumerate - process/list/count all items in the table

A SIMPLE IMPLEMENTATION

Like lists, a ***table*** is an abstract data type. That is, it describes what it is but not how to implement one. The underlying data structure needed to create a table can vary widely from arrays sorted by keys to hash tables and even trees.

A simple implementation is to simply use an array sorted by keys.

Insertion/Update

To add to this table, we must first find the spot where the item will go. This can be done by modifying a binary search algorithm. Once the location is found, if a record with the same key is not already in the table, we will need to shift every item over to make room for the new record. If a record with a matching key already exists, the old record can be replaced with the new.

The run time for performing the search is O(log n). If the record already exists, and we are just updating it, the run time would be O(log n). However, inserting a brand new record with a different key will require shifting on average 50% of the list, and thus we are looking at a run time of O(n) for that operation.

Remove

To remove a record, we can start with a binary search algorithm to find the record according to the key. If such a record exists, removal will involve shifting the records down.

The run time for search is O(log n). However to remove, we must shift all the records down. This process is O(n). Thus the remove operation is O(n)

Search

As the array is sorted by key, all searching can be done using a binary search. This has a run time of O(log n)

Drawbacks

This implementation clearly has a lot of drawbacks. While the search() function has an acceptable run time of $logn$, the other functions are generally slow. The only part that is fast is search. Ideally it would be faster than this.

HASH TABLES

A hash table uses the key of each record to determine the location in an array structure. To do this, the key is passed into a hash function which will then return a numeric value based on the key.

Hash Functions

A hash function must be designed so that given a certain key it will always return the same numeric value. Furthermore, the hash function will ideally distribute all possible keys in the keyspace uniformly over all possible locations.

For example suppose that we wanted to create a table for storing customer information at store. For the key, a customer's telephone number is used. The table can hold up to 10,000 records and thus valid indexes for an array of that size would be [0 - 9999]

Telephone numbers are 10 digits (###) ###-####

The first 3 of which is an area code.

Now, if your hash function was: use the first 4 digits of the phone number (area code + first digit of number) that hash function would not be very good because most people in the same area would have the same area code. Most people in the Toronto for example have area code of 416 or 647... so there would be very little variation in the records. However the last 4 digits of a phone number is much more likely to be different between users (though certainly not unique).

Generally speaking a good hash function should be:

- uniform (all indices are equally likely for the given set of possible keys)
- random (not predictable)

LOAD FACTOR

The load factor denoted by the symbol λ (pronounced lambda) measures the fullness of the hash table. It is calculated by the formula:

$$\lambda = \frac{\text{number of records in table}}{\text{number of locations}}$$

COLLISIONS- THE PIGEON HOLE PRINCIPLE

Suppose you had n mailboxes and m letters where m > n (more letters than mailboxes). If you were to place all the letters into the available mailboxes, there would be at least one mailbox with at least 2 letters in it. This is the pigeon hole principle.

This is effectively what the situation is with our hash function and keys. The number of mailboxes we have is n (capacity of array). The total number of possible keys is m. Typically, the total number of possible keys is bigger than the capacity of the array. (usually significantly bigger). Based on the pigeon hole principle, it means that we will have situations where at two keys will get hashed into the same hash index.

When two keys have the same hash index you have a ***collision***. Generally speaking, collisions are unavoidable. The key is to have a method of resolving them when they do happen. The rest of this chapter look at different ways to deal with collisions.

CHAINING

At every location (hash index) in your hash table store a linked list of items. You only use as many nodes as necessary. Using Chaining the Table will not overflow as you can have as long a chain as you need. However, You will still need to conduct a short linear search of the linked list but if your hash function uniformly distributes the items, the list should not be very long.

For chaining, the runtimes depends on the load factor (λ) The average length of each chain is λ . λ is the number of expected probes needed for either an insertion or an unsuccessful search. For a successful search it is $1 + \frac{\lambda}{2}$ probes.

While it is possible for $\lambda > 1$, it is generally not a great idea to be too much over. Your goal isn't to search long chains as that is very slow. The ability to have long chains is more of a safety feature... you should try to still have as short a chain as possible.

Chaining is a simple way of handling collisions. Instead of storing the key-value pair (k, v) into the array (with capacity m) directly, chaining creates an array of linked lists, initially all empty.For each operation involving key k

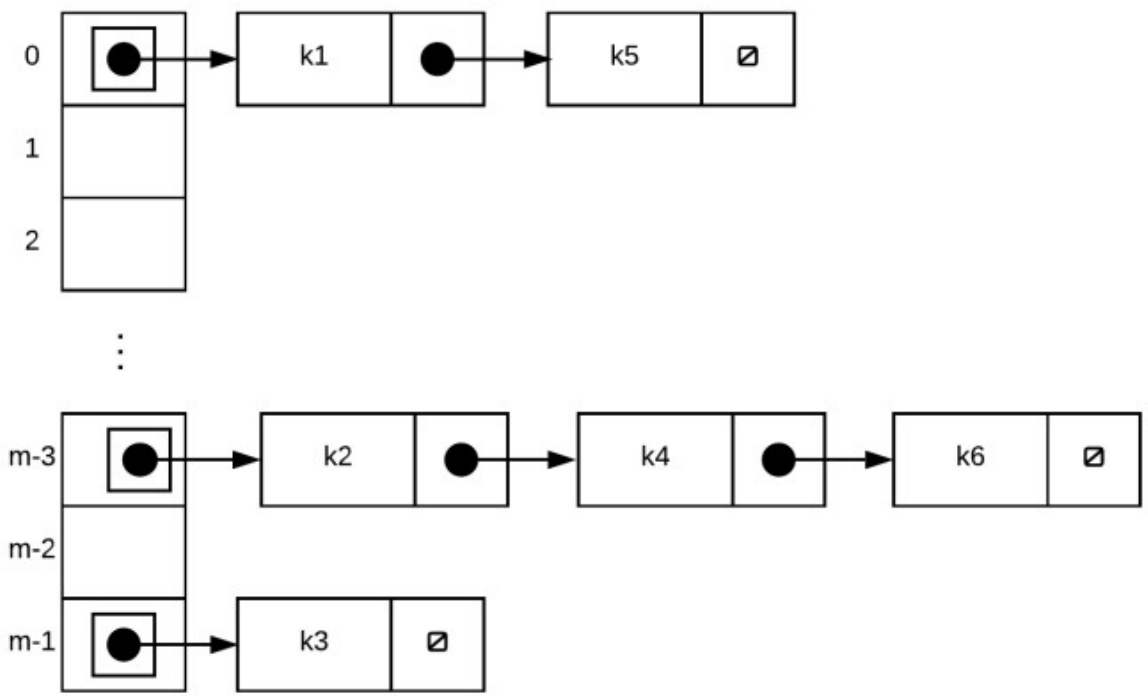
- calculate $i = hashIndex(k, m)$
- perform operation (insert/delete/search) on the linked list at array[i]

Example

Suppose we were to have 6 keys $(k1, k2, k3, k4, k5, k6)$. The hash function returns as follows for these keys:

- $hashIndex(k1, m) == 0$
- $hashIndex(k2, m) == m - 3$
- $hashIndex(k3, m) == m - 1$
- $hashIndex(k4, m) == m - 3$
- $hashIndex(k5, m) == 0$
- $hashIndex(k6, m) == m - 3$

A table created using chaining would store records as follows (note that only key's are shown in diagram for brevity)


0
1
2
k1
k5
m-3
m-2
m-1
k2
k4
k6
k3

WORST CASE RUN TIME

insert(k,v) - cost to find the correct linked list + cost to search for k within the linked list, + cost to add new node or modify existing if k is found

search(k) - cost to find the correct linked list + cost to search for k within the linked list

delete(k) - cost to find the correct linked list + cost to search for k within the linked list + cost to remove a node from list

In each of the above cases, cost of to find the correct linked list is $\theta(1)$ assuming that the cost of calculating hash is constant relative to number keys. We simply need to calculate the hash index, then go to that location

The cost to add a node, modify a node or delete a node (once node has been found) is $\theta(1)$ as that is the cost to remove/insert into linked list given pointers to appropriate nodes

The cost to search through linked list depends on number of nodes in the linked list. At worst, every single key hashes into exactly the same index. If that is the case, the run time would be $\theta(n)$

Thus, the worst case run time is $\theta(n)$. In reality of course, the performance is signficantly better than this and you typically don't encounter this worst case behaviour. Notice that the part that is slow is the search along the linked list. If our linked list is relatively short then the cost to search it would also not take very long.

AVERAGE CASE RUN TIME

We begin by making an assumption called Simple Uniform Hash Assumption (SUHA). This is the assumption that any key is equally likely to hash to any slot. The question then becomes how long are our linked lists? This largely depends on the load factor $\lambda = n/m$ where n is the number of items stored in the linked list and m is the number of slots. The average run time is $\theta(1 + \lambda)$

LINEAR PROBING

Chaining essentially makes use of a second dimension to handle collisions. Chaining is an example of a **closed addressing**. With closed addressing collision resolution methods use the hash function to specify the exact index of where the item is found. We may have multiple items at the index but you are looking at just that one index.

This is not the case for linear probing. Linear Probing only allows one item at each element. There is no second dimension to look. Linear probing is an example of open addressing. Open addressing collision resolution methods allow an item to be placed at a different spot other than what the hash function dictates. Aside from linear probing, other open addressing methods include quadratic probing and double hashing.

With hash tables where collision resolution is handled via open addressing, each record actually has a set of hash indexes where they can go. If the first location at the first hash index is occupied, it goes to the second, if that is occupied it goes to the third etc. The way this set of hash indexes is calculated depends on the probing method used (and in implementation we may not actually generate the full set but simply apply the probing algorithm to determine where the "next" spot should be).

Linear probing is the simplest method of defining "next" index for open address hash tables. Suppose hash(k) = i, then the next index is simply i+1, i+2, i+3, etc. You should also treat the entire table as if its round (front of array follows the back). Suppose that m represents the number of slots in the table, We can thus describe our probing sequence as:

$$\{hash(k), (hash(k)+1)\%m, (hash(k)+2)\%m, (hash(k)+3)\%m, \dots\}$$

METHOD 1

Insertion

The insertion algorithm is as follows:

- use hash function to find index for a record
- If that spot is already in use, we use next available spot in a "higher" index.
- Treat the hash table as if it is round, if you hit the end of the hash table, go back to the front

Each contiguous group of records (groups of record in adjacent indices without any empty spots) in the table is called a cluster.

Searching

The search algorithm is as follows:

- use hash function to find index of where an item should be.
- If it isn't there search records that records after that hash location (remember to treat table as cicular) until either it found, or until an empty record is found. If there is an empty spot in the table before record is found, it means that the the record is not there.
- NOTE: it is important not to search the whole array till you get back to the starting index. As soon as you see an empty spot, your search needs to stop. If you don't, your search will be incredibly slow

Removal

The removal algorithm is a bit trickier because after an object is removed, records in same cluster with a higher index than the removed object has to be adjusted. Otherwise the empty spot left by the removal will cause valid searches to fail.

The algorithm is as follows:

- find record and remove it making the spot empty
- For all records that follow it in the cluster, do the following:
  - determine the hash index of the record
  - determine if empty spot is between current location of record and the hash index.
  - move record to empty spot if it is, the record's location is now the empty spot.

EXAMPLE

Suppose we the following 7 keys and their associated hash indices. Let us then insert these 5 keys from k1 to k5 in that order.

Key	Hash Index
k1	8
k2	7
k3	9
k4	1
k5	8
k6	9
k7	8

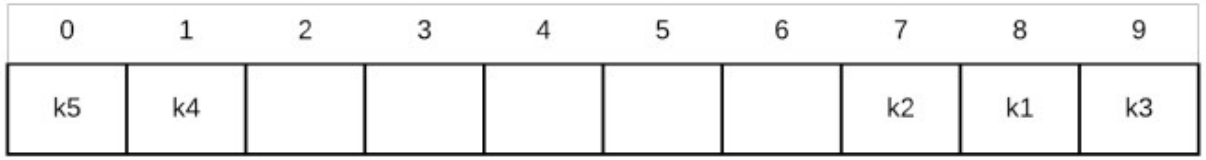
Insert keys k1 to k4

All four keys have the different hash indexes and thus, no collisions occur, they are simply placed in their hash position.

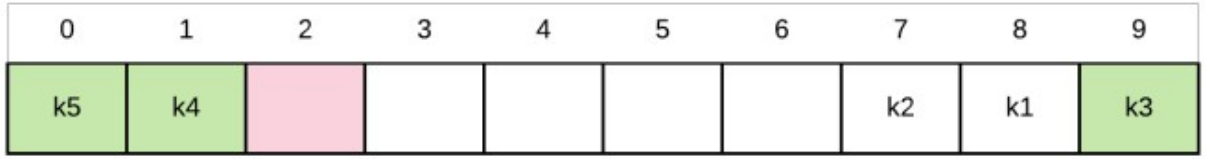
0	1	2	3	4	5	6	7	8	9
	k4						k2	k1	k3

Insert k5. probe sequence of k5 is $\{(8+0)\%10, (8+1)\%10, (8+2)\%10, (8+3)\%10, (8+4)\%10, (8+5)\%10, \dots\} = \{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we place k5 into index 0 because 8, 9 and 0 are all occupied

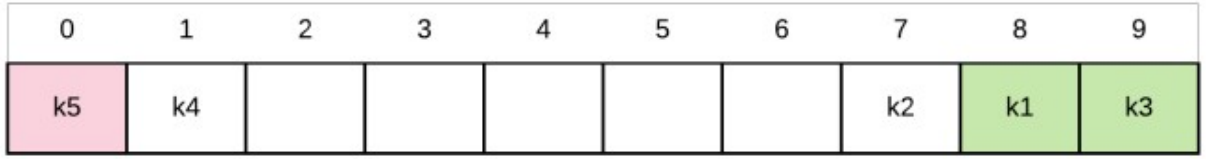
EXAMPLE CONT.



Suppose we then decided to do a ***search for k6***. k6 does not exist, so the question is when can we stop. k6's probe sequence is: $\{(9+0)\%10, (9+1)\%10, (9+2)\%10, (9+3)\%10, (9+4)\%10, (9+5)\%10, \dots\} = \{9, 0, 1, 2, 3, 4 \dots\}$. We begin looking at the first probe index 9. We proceed until we get to index 2. Since index 2 is empty, we can stop searching



Search for k5. If we were to search for something that is there, this is what would happen. Probe sequence for k5 is $\{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we would start search at 8, we would look at indices 8,9,0, and 1. At index 1 we find k5 so we stop



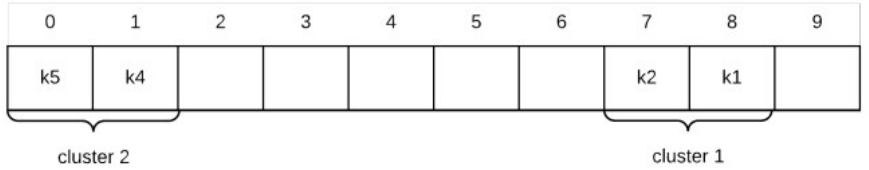
EXAMPLE CONT.

Now, lets remove a node.

A ***cluster*** is a group of records without any empty spots. Thus, any search begins with a hashindex within a cluster searches to the end of the cluster.

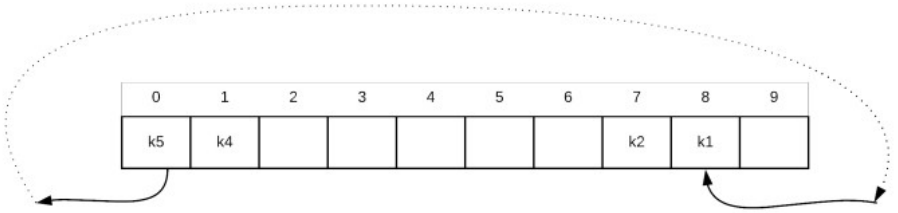
Currently there is one big cluster from index to 7 to index 1 inclusive.

Suppose we ***removed k3.*** If we did this, our one big cluster would be split into two smaller clusters. This is actually a good thing as search stops on first empty spot. So the only question really is whether each record in the group that follows the removed records are in the correct cluster (the groups before the removed record is always in the correct spot.

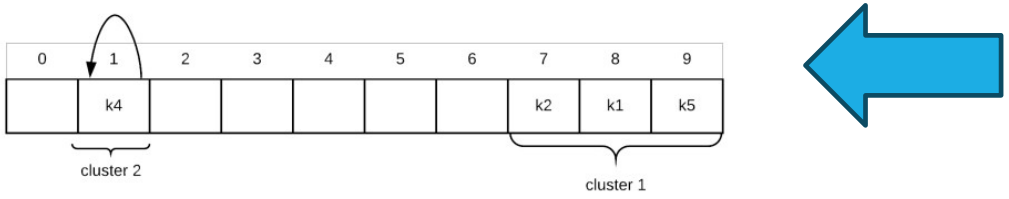


So we go through the remaining records in the cluster and use the hashindex of each key to determine if its in the correct cluster. If it is in the wrong cluster we move. it

Continuing with example, we look at k5. k5 should actually go in 8, so the record is in the wrong side of the empty spot, so what we do is move the record into the empty spot, make k5's spot the empty spot and continue.



We test k4 and its in the correct side of the empty spot so we leave it alone



We finish processing the records within the cluster so we are done.

METHOD 2: TOMBSTONING

Tombstoning is a method that is fairly easy to implement. It requires each element to not only store the record, but also a status of element. There are three statuses:

- Empty - nothing has ever been inserted into this spot
- Used/Occuppied - a record is stored here
- Deleted - something was here but it has been deleted. Note this is NOT exactly the same as empty. You will see why in a moment.

Insertion

- If it is not there, start looking for the first "open" spot. An open spot is the first probe index that is either deleted or empty.

Searching

The search algorithm is as follows:

- use hash function to find index of where an item should be.
- Check to see if the item's key matches that of key we are looking for
- If it isn't there search for key-value pairs in the "next" slot according to the probing method until either it found, or until an we hit an empty slot.
- NOTE: it is important **not** to search the whole array till you get back to the starting index. As soon as you see an empty slot, your search needs to stop. If you don't, your search will be incredibly slow for any item that doesn't exist.

  DANGER

Note that only empty slots stop searching not deleted slots

Removal

The removal algorithm is as follows:

- search for the record with matching key.
- If you find it, mark the spot as deleted

Suppose we the following 6 keys and their associated hash indices (these are picked so that collisions will definitely occur). Let us then insert these 5 keys from k1 to k5 in that order.

Key	Hash index
k1	8
k2	7
k3	9
k4	7
k5	8
k6	9

Insert k1 to k3 (no collisions, so they go to their hash indices)

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
							k2	k1	K3

Insert k4. probe sequence of k4 is $\{(7+0)\%10, (7+1)\%10, (7+2)\%10, (7+3)\%10, (7+4)\%10, (7+5)\%10, \dots\} = \{7, 8, 9, 0, 1, 2 \dots\}$. Thus, we place k4 into index 0 because 7, 8 and 9 are all occupied

0	1	2	3	4	5	6	7	8	9
Used	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
k4							k2	k1	K3

Insert k5. probe sequence of k5 is $\{(8+0)\%10, (8+1)\%10, (8+2)\%10, (8+3)\%10, (8+4)\%10, (8+5)\%10, \dots\} = \{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we place k5 into index 1 because 8, 9 and 0 are all occupied

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
k4	k5						k2	k1	K3

Suppose we then decided to do a search. First lets search for something that isn't there, k6. k6's probe sequence is: $\{(9+0)\%10, (9+1)\%10, (9+2)\%10, (9+3)\%10, (9+4)\%10, (9+5)\%10, \dots\} = \{9, 0, 1, 2, 3, 4 \dots\}$. We begin looking at the first probe index. We proceed until we get to index 2. Since index 2 is empty, we can stop searching

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
k4	k5						k2	k1	K3

If we were to search for something that is there (k5 for example), here is what we would do. Probe sequence for k5 is $\{8, 9, 0, 1, 2, 3 \dots\}$. Thus, we would start search at 8, we would look at indices 8,9,0, and 1. At index 1 we find k5 so we stop

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
k4	k5						k2	k1	K3

Suppose we delete k3. All we need to do is find it, and mark the spot as deleted

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Deleted
k4	k5						k2	k1	

When a spot is deleted, we still continue when we search... thus if we were to look for k5, we do not stop on deleted, we must keep going.

0	1	2	3	4	5	6	7	8	9
Used	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Deleted
k4	k5						k2	k1	

Quadratic Probing

With linear probing everytime two records get placed beside each other in adjacent slots, we create a higher probability that a third record will result in a collision (think of it as a target that got bigger). One way to avoid this is to use a different probing method so that records are placed further away instead of immediately next to the first spot. In quadratic probing, instead of using the next spot, we use a quadratic formula in the probing sequence. The general form of this algorithm for probe sequence i is: $hash(k) + c_1 i + c_2 i^2$. At it's simplest we can use $hash(k) + i^2$. Thus, we can use:
$\{hash(k), (hash(k) + 1)\%m, (hash(k) + 4)\%m, (hash(k) + 9)\%m, \dots\}$

Key	Hash index
k1	8
k2	7
k3	9
k4	7
k5	8
k6	9

Quadratic Probing Example

Insert k1 to k3 (no collisions, so they go to their hash indices)

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
							k2	k1	k3

Insert k4. probe sequence of k4 is $\{(7 + 0^2)\%10, (7 + 1^2)\%10, (7 + 2^2)\%10, (7 + 3^2)\%10, (7 + 4^2)\%10, (7 + 5^2)\%10, \dots\} = \{7, 8, 1, 6, 3, 2 \dots\}$. Thus, we place k4 into index 1 because 7 and 8 are both occupied

0	1	2	3	4	5	6	7	8	9
Empty	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
	k4						k2	k1	k3

Insert k5. probe sequence of k5 is $\{(8 + 0^2)\%10, (8 + 1^2)\%10, (8 + 2^2)\%10, (8 + 3^2)\%10, (8 + 4^2)\%10, (8 + 5^2)\%10, \dots\} = \{8, 9, 2, 7, 4, 3 \dots\}$. Thus, we place k5 into index 2 because 8 and 9 are both occupied

0	1	2	3	4	5	6	7	8	9
Empty	Used	Used	Empty	Empty	Empty	Empty	Used	Used	Used
	k4	k5					k2	k1	k3

Likewise searching involves probing along its quadratic probing sequence. Thus, searching for k6 involves the probe sequence $\{(9+0^2)\%10, (9+1^2)\%10, (9+2^2)\%10, (9+3^2)\%10, (9+4^2)\%10, (9+5^2)\%10, \dots\} = \{9, 0, 3, 8, 5, 4 \dots\}$. We search index 9, then index 0. We can stop at this point as index 0 is empty

0	1	2	3	4	5	6	7	8	9
Empty	Used	Used	Empty	Empty	Empty	Empty	Used	Used	Used
	k4	k5					k2	k1	k3

Double Hashing

Double hashing addresses the same problem as quadratic probing. Instead of using a quadratic sequence to determine the next empty spot, we have 2 different hash functions, $hash_1(k)$and $hash_2(k)$. We use the first hash function to determine its general position, then use the second to calculate an offset for probes. Thus the probe sequence is calculated as:
$\{hash_1(k), (hash_1(k) + hash_2(k))\%m, (hash_1(k) + 2hash_2(k))\%m, (hash_1(k) + 3hash_2(k))\%m, \dots\}$

Key	Hash1(key)	Hash2(key)
k1	8	6
k2	7	2
k3	9	5
k4	7	4
k5	8	3
k6	9	2

Double uses a second hash function to calculating a probing offset. Thus, the first hash function locates the record (initial probe index)... should there be a collision, the next probe sequence is a hash2(key) away.

Again we start off with hashing k1, k2 and k3 which do not have any collisions

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
							k2	k1	K3

Insert k4. probe sequence of k4 is $\{(7 + 0(4))\%10, (7 + 1(4))\%10, (7 + 2(4))\%10, (7 + 3(4))\%10, \dots\} = \{7, 1, 5, 9, \dots\}$. Thus, we place k4 into index 1 because 7 was occupied

0	1	2	3	4	5	6	7	8	9
Empty	Used	Empty	Empty	Empty	Empty	Empty	Used	Used	Used
	k4						k2	k1	k3

Insert k5. probe sequence of k5 is $\{(8 + 0(3))\%10, (8 + 1(3))\%10, (8 + 2(3))\%10, (8 + 3(3))\%10, \dots\} = \{8, 1, 4, 7, \dots\}$. Thus, we place k5 into index 4 because 8 and 1 were occupied

0	1	2	3	4	5	6	7	8	9
Empty	Used	Empty	Empty	Used	Empty	Empty	Used	Used	Used
	k4			k5			k2	k1	k3

END OF WEEK 5

- Please work on your quiz 2