Open Addressing Collision Handling

Hashing

The basic idea behind hashing is to take a field in a record, known as the key, and convert it through some fixed process to a numeric value, known as the **hash key**, which represents the position to either store or find an item in the table. The numeric value will be in the range of `0` to `n-1`, where `n` is the maximum number of slots in the table.

The fixed process to convert a key to a hash key is known as a **hash function**. This function will be used whenever access to the table is needed.

One common method of determining a hash key is the division method of hashing. The formula that is used is:

hash key = key % capacity the table

The division method is generally a reasonable strategy, unless the key happens to have some undesirable properties. For example, if the table size is 10 and all of the keys end in zero. In this case, the choice of hash function and table size needs to be carefully considered. The best table sizes are generally prime numbers.

No matter what the hash function, there is the possibility that two keys could resolve to the same hash key. This situation is known as a **collision**.

When this occurs, there are two straightforward solutions: chaining and open addressing.

Linear Probing

When using linear probing, the item will be stored in the next available slot in the table, assuming that the table is not already full. This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there. So the probe sequence is given by:

```
hash(key) + 1, hash(key) + 2, ...
```

or

```
hash(key) + i,
```

where `i` is the number of unsuccessful probes so far

If an empty slot is not found before reaching the point of collision, the table is full.

A problem with linear probing is that it is possible for blocks of data to form when collisions are resolved. This is known as **primary clustering**. This means that any key that hashes into the cluster will require several attempts to resolve the collision.

```
Linear Probing
[---,---,---,---,---,---,---,---,---,---,---]
[---,---,---, 14,---,---,---,---,---,---,---] Inserting 14  Probes: 1
[---,---,---, 14,---,---, 17,---,---,---,---] Inserting 17  Probes: 1
[---,---,---, 14, 25,---, 17,---,---,---,---] Inserting 25  Probes: 2
[---,---,---, 14, 25, 37, 17,---,---,---,---] Inserting 37  Probes: 2
[---, 34,---, 14, 25, 37, 17,---,---,---,---] Inserting 34  Probes: 1
[---, 34,---, 14, 25, 37, 17, 16,---,---,---] Inserting 16  Probes: 3
[---, 34,---, 14, 25, 37, 17, 16, 26,---,---] Inserting 26  Probes: 5
Inserted 7 values, 7 distinct in 139 ms
Average no. of probes: 2.142857142857143 Longest probe: 5
```

Quadratic Probing

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision, where `i` is the number of attempts to resolve the collision.

That is we look in slots:

```
hash(key) + 1, hash(key) + 4, hash(key) + 9
```

A major limitation of this scheme is that it there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This means that even if there are empty slots in the table, the probe sequence may not find them.

If the sign of the offset is alternated (e.g. +1, -4, +9, -16 etc.), and if the number of buckets is a prime number `p` congruent to 3 modulo 4 (i.e. one of 3, 7, 11, 19, 23, 31 and so on), a free slot will always be found as long as one exists.

So the probe sequence becomes:

```
hash(key) + 1, hashkey) - 4, hash(key) + 9, hash(key) - 16
```

and so on.

In either case, quadratic probing suffers from **secondary clustering** because elements that hash to the same hash key will always probe the same alternative cells.

```
Quadratic Probing
[---,---,---,---,---,---,---,---,---,---,---]
[---,---,---, 14,---,---,---,---,---,---,---] Inserting 14  Probes: 1
[---,---,---, 14,---,---, 17,---,---,---,---] Inserting 17  Probes: 1
[---,---,---, 14, 25,---, 17,---,---,---,---] Inserting 25  Probes: 3
[---,---,---, 14, 25, 37, 17,---,---,---,---] Inserting 37  Probes: 3
[---, 34,---, 14, 25, 37, 17,---,---,---,---] Inserting 34  Probes: 1
[ 16, 34,---, 14, 25, 37, 17,---,---,---,---] Inserting 16  Probes: 6
[ 16, 34, 26, 14, 25, 37, 17,---,---,---,---] Inserting 26  Probes: 5
Inserted 7 values, 7 distinct in 19 ms
Average no. of probes: 2.857142857142857 Longest probe: 6
```

Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions from the point of collision to insert.

Like linear probing, it uses one hash value as a starting point and then repeatedly steps forward a fixed interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering.

Here the probe sequence is given by:

```
hash(key) + (i * hash2(key))
```

There are a couple of requirements for the second function; it must never evaluate to 0 (it will never move from the home slot) and it must ensure that all cells in the table can be probed.

A popular second hash function is: `hash2(key) = R - ( key % R )` where `R` is a prime number that is smaller than the size of the table.

The second hash function in my implementation is: `1 + (key % (capacity - 1))`
This gives a potentially fairly long stride which helps space elements out.

```
Double Hashing
[---,---,---,---,---,---,---,---,---,---,---]
[---,---,---, 14,---,---,---,---,---,---,---] Inserting 14  Probes: 1
[---,---,---, 14,---,---, 17,---,---,---,---] Inserting 17  Probes: 1
[---,---,---, 14,---,---, 17,---,---, 25,---] Inserting 25  Probes: 2
[---,---,---, 14, 37,---, 17,---,---, 25,---] Inserting 37  Probes: 1
[---, 34,---, 14, 37,---, 17,---,---, 25,---] Inserting 34  Probes: 1
[---, 34,---, 14, 37, 16, 17,---,---, 25,---] Inserting 16  Probes: 1
[ 26, 34,---, 14, 37, 16, 17,---,---, 25,---] Inserting 26  Probes: 2
Inserted 7 values, 7 distinct in 4 ms
Average no. of probes: 1.2857142857142858 Longest probe: 2
```

Rehashing
Once the hash table gets completely full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

The new size of the hash table should also be prime and existing keys will reinserted into the table based on the new table size.

This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is okay though since it shouldn't happen too often.

The question becomes when should the rehashing be applied? Some possible choices are: when the table becomes half full, once an insertion fails or once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size.

Deletion
The method of deletion depends on the method of insertion. In any of the cases, the same hash function(s) will be used to find the location of the element in the hash table to be deleted.

It is not possible to simply delete elements by marking their location as free - subsequent searches are likely to break. Instead, we leave deleted keys in the table with a marker or flag that they are not valid elements of the table. Now, subsequent searches will not terminate prematurely. When inserting a new key, it overwrites a deleted key.

However, for a large number of deletions and a relatively small number of new insertions, the table can become overloaded with deleted keys, increasing the search time.