## **Closed Hashing (Open Addressing)**

## **Probing**

* During an attempt to insert a new item into a **hash table**, if the hash function indicates a location in the hash table that is already occupied, a **collision** occurs.
* When a collision occurs, we attempt to **probe** a new possible index in which we can place the object.
* This process, called **probing**, involves **finding an empty/open**index to use if a collision occurs.
* The sequence of locations that you examine is called the **probe sequence**.
* Schemes that probe are said to use **open addressing**.

## **Open Addressing Schemes**

* The concern, of course, is that you must be able to not only **add** an item, but also be able to reproduce the probe sequence so that we can **remove** and **find** the items later in time.
* The difference among open-addressing schemes is the **technique used to probe for an empty location**.
  1. **Linear Probing**
  2. **Quadratic Probing**
  3. **Double Hashing**
* More formally,

cells *h*0(*x*), *h*1(*x*), *h*2(*x*), . . . are tried in succession,

where *hi*(*x*) = (*hash*(*x*) + *f*(*i*)) % *TableSize*, with *f*(0) = 0,

where the function *f* is the collision resolution strategy.

* We now look at three common collision resolution strategies.

### **Linear Probing**

**Description:**

* In this collision resolution technique, you **search the hash table** **sequentially**, **starting from the original hash location.**
* In simpler terms, you continue to look forward one index at a time for the next free index.

**Algorithm:**

* When a **bucket *i*** is **used**, the **next bucket** you will try is **bucket *i+1***
* The **search** can **wrap around** from the last array index to the **start of the array**.

Hash Function *h*(*x*) = *x* % *TableSize*

After ith collision *h*i(*x*) = (*x* + *f(i)*) % *TableSize*

*f* is a linear function of *i f(i)* = *i*

* This amounts to trying cells sequentially (with wraparound) in search of an empty cell.
* Let’s say you find a collision at table[h(x)].
  + If table[h(x)] is occupied, you check the hash table location at table[h(x)+1].
  + If table[h(x)+1] is occupied, you check the hash table location at table[h(x)+2].
  + If table[h(x)+2] is occupied, you check the hash table location at table[h(x)+3].
  + … and so on until you find an available location.

**Example:**

* The following diagram illustrates linear probing
  + we use the hash function *h*(*x*) = *x* % 101
  + the placement of {7597, 4567, 628, 2658} will map to *h*(*x*) = 22 which is table[22]

Table

Description automatically generated

**Implementing Contains**

* To implement **contains**, you need to follow the same probe sequence that **add** used until you
  + find the item you are searching for

or

* + reach an empty location, which indicates that the item is not present

or

* + or visit every table location

**Implementing Remove**

* Removals, however, complicate matters slightly.
* The remove operation itself is no problem. You merely find the desired item, as in getItem, and remove it from the hash table, making the location empty.
* But now what happens when getItem needs to locate an item?
* The new empty locations that remove created along a probe sequence could cause getItem to stop prematurely, incorrectly indicating a failure.
* You can solve this problem by placing a table location into one of three states:
  1. occupied (currently in use),
  2. empty (has not been used),

or

* 1. removed (was once occupied but is now available).
* You then modify the getItem operation to continue probing when it encounters a location in the removed state.
* Similarly, you modify add to insert into locations that are in either the empty or removed states.

**Problem: Primary Clustering**

* If a lot of hash codes end up mapping to the same index, then objects are grouped in consecutive locations called **clusters**.
* This phenomenon is called **primary clustering**.
* Any key that hashes into a cluster will require several attempts to resolve the collision and will then be added to the cluster.
* Due to primary clustering, it is no longer as simple to find an element’s value.
  + As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large if a cluster is large.
  + A value whose hash function evaluates to k might be stored at index k, but if something other than k is there, it might be at index k + 1, k + 2, etc.
* Clusters can get close to one another and, in fact, merge into a larger cluster.
  + Large clusters tend to get even larger.
  + Thus, one part of the hash table might be quite densely populated, even though another part has relatively few items.
* We must think about the original goals of this implementation. Searching for elements is supposed to be fast, and if we have to probe through a lot of elements to find anything, we’re losing the efficiency we sought after in the first place.

## **Quadratic Probing**

* You can virtually eliminate primary clusters simply by adjusting the linear probing scheme.
* Instead of probing consecutive table locations from the original hash location one after another, we square the distance we are from the table location each time we probe.

**Algorithm:**

* When a **bucket *i*** is **used**, the **next bucket** you will try is **bucket (*i+1)2***
* The **search** can **wrap around** from the end of the array to the **start of the array**.

Hash Function *h*(*x*) = *x* % *TableSize*

After ith collision *h*i(*x*) = (*x* + *f(i)*) % *TableSize*

*f* is a quadratic function of *i* *f(i)* = *i2*.

* Let’s say you find a collision at table[h(x)].
  + If table[h(x)] is occupied, you check the hash table location at table[h(x)+12].
  + If table[h(x)+12] is occupied, you check the hash table location at table[h(x)+22].
  + If table[h(x)+22] is occupied, you check the hash table location at table[h(x)+32].
  + … and so on until you find an available location.

**Example:**

* The following diagram illustrates quadratic probing
  + we use the hash function *h*(*x*) = *x* % 101
  + the placement of {7597, 4567, 628, 2658} will map to *h*(*x*) = 22 which is table[22]

Table

Description automatically generated

**Problem: Secondary Clustering**

* Unfortunately, when two items hash into the same location, quadratic probing uses the **same probe sequence** **for each item**.
* The resulting phenomenon—called secondary clustering—delays the resolution of the collision.
* However, secondary clustering is still preferrable to primary clustering.
* Secondary clustering is a slight theoretical blemish.
* Simulation results suggest that it generally causes less than an extra half probe per search.

## **Double Hashing**

* Double hashing is another open-addressing scheme that **drastically reduces clustering**.
* The double hashing technique virtually eliminates clustering but does so at the cost of computing an extra hash function.
* The probe sequences that both linear probing and quadratic probing use are **key independent**.
  + Linear probing probes the hash table sequentially regardless of the hash key.
  + Quadratic probing probes the hash table quadratically regardless of the hash key.
* In contrast, double hashing defines **key-dependent** probe sequences.
* In this scheme, the probe sequence is done in two steps

1. We generate a hash code using an initial hash function as we did previously *h1*(*key*).
2. In the case of a collision, we use a linear probe sequence in conjunction with a second hash function *h2*(*key*) to determine the size of the steps taken.

* Although you choose *h*1(*key*) as usual, you must follow these guidelines for *h*2(*key*):

*h2*(*key*) != 0

*h2*(*key*) != *h1*(*key*)

* There are many ways you could do h2, but we want to avoid these possible pitfalls:
  + Make sure the function never evaluates to zero.
  + Make sure all cells in the hash table can be probed.

A function such as *h*2(*x*) = *R* − (*x* mod *R*), with *R* a prime smaller than *TableSize,* will work well in many cases.

**Algorithm:**

* When a **bucket *i*** is **used**, the **next bucket** you will try is **bucket *( i · h2(x) )2***
* The **search** can **wrap around** from the end of the array to the **start of the array**.

Hash Function 1 *h1*(*x*) = *x* % *TableSize*

Hash Function 2 *h*2(*x*) = *R* − (*x* % *R*)

where *R* is a prime number smaller than *TableSize*

After ith collision *h*i(*x*) = (*x* + *f(i)*) % *TableSize*

*f* is a function of *i* and h2 *f*(*i*) = *i* **·** *h*2(*x*)

* Let’s say you find a collision at table[h1(x)].
  + If table[h1(x)] is occupied, you check the index at table[h(x) + 1**·**h2(x)].
  + If table[h(x) + 1**·**h2(x)] is occupied, you check the index at table[h(x) + 2**·**h2(x)].
  + If table[h(x) + 2**·**h2(x)] is occupied, you check the index at table[h(x) + 3**·**h2(x)].
  + … and so on until you find an available location.

## **Efficiency of Hashing**