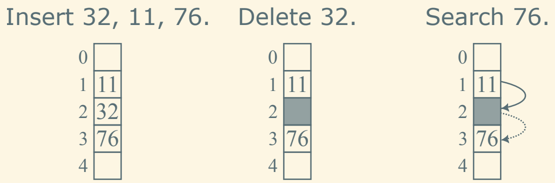
# Hash Tables: Open Addressing and Resizing

Michael T. Gastner (21 March 2023)



**Disclaimer:** These slides are based on and occasionally quote from ‘Introduction to Algorithms’ (3rd ed.) by Cormen et al. (2009), MIT Press.

## Review of Previous Lesson

### Hashing with Chaining

In the previous lesson, we learnt that hash tables are an abstract data type that stores a value $v$ for each key $k$ using a hash function $h$. The hash function returns a nonnegative integer $h(k)$, known as the hash value of $k$, which is equal to the index of the slot where $v$ is stored. Ideally, hash functions satisfy the criteria for simple uniform hashing (i.e. any random key is equally likely to hash into any of the slots, independently of where any other key has hashed to).

In almost all realistic settings, there are more possible keys than there are slots. Thus, some keys must share the same hash value; in this case, we say that the keys 'collide'. Collisions can be resolved with a technique known as chaining, whereby all the elements that hash to the same slot are placed in the same linked list. Slot $j$ contains a pointer to the head of the list of all stored elements that hash to $j$; if there are no such elements, slot $j$ contains NIL.

## Learning Objectives

By the End of This Lesson, You Should Be Able to ...

- Describe the difference between conflict resolution by chaining and open addressing.
- Define the concept of uniform hashing.
- State the general forms of hash functions used by linear probing, quadratic probing and double hashing.
- Apply linear probing, quadratic probing and double hashing to a sequence of insertions.
- Explain why linear probing, quadratic probing and double hashing violate the assumption of uniform hashing and why this violation can cause primary and secondary clustering.
- Explain why hash tables can be resized in $O(1)$ amortised time.

## Open Addressing
### Elements Are Stored Directly in the Hash Table

In **open addressing**, all elements occupy the hash table itself. That is, each table entry contains either an element or NIL.

When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

In contrast to chaining, no lists and no elements are stored outside the table.

Thus, in open addressing, the hash table can ‘fill up’ until no further insertions can be made. That is, the load factor $\alpha$ can never exceed 1.

## Probing

### Check a Sequence of Slots and Insert the Key in the First Free Slot

To perform insertion using open addressing, we **probe** the hash table until we find an empty slot in which to insert the key. Instead of being fixed in the order 0, 1, . . . , $m-1$ (which requires $\Theta(n)$ search time), the sequence of probes *depends on the key being inserted*.

To determine which slots to probe, we extend the hash function to include the probe number as a second input:

$$h : U \times \{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}.$$

The **probe sequence** $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$ must be a permutation of $\langle 0, 1, \dots, m-1 \rangle$ so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

## Inserting an Element When Using Open Addressing

### Pseudocode

The HASH-INSERT-WITHOUT-DELETED procedure takes as input a hash table $T$ and a key $k$. It either returns the slot number where it stores key $k$ or flags an error because the hash table is already full.

```
HASH-INSERT-WITHOUT-DELETED($T, k$)
1  $i = 0$
2  repeat
3      $j = h(k, i)$
4      if $T[j] ==$ NIL
5          $T[j] = k$
6          return $j$
7      else $i = i + 1$
8  until $i == m$
9  error "hash table overflow"
```

## Searching for an Element When Using Open Addressing

### Pseudocode

The algorithm for searching for key $k$ probes the same sequence of slots that the insertion algorithm examined when key $k$ was inserted.

```
HASH-SEARCH($T, k$)
1  $i = 0$
2  repeat
3      $j = h(k, i)$
4      if $T[j] == k$
5          return $j$
6      $i = i + 1$
7  until $T[j] ==$ NIL or $i == m$
8  return NIL
```

## Deleting an Element

### Slot Must Be Marked as DELETED Instead of NIL

Deletion from an open-address hash table is difficult. When we delete a key from slot $i$, we cannot simply mark that slot as empty by storing NIL in it. If we did, we might be unable to retrieve any key during whose insertion we had probed slot $i$ and found it occupied.

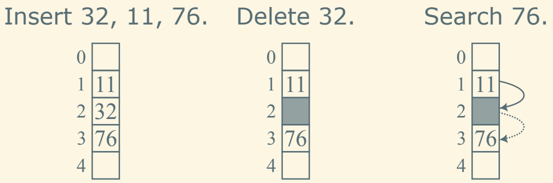
We can solve this problem by marking the slot as DELETED instead of NIL.

## Deleting an Element
### Example

In the figure below, the hash function is assumed to be $h(k, i) = (k + i) \bmod 5$.

If slot 2 is marked as NIL after deleting 32, $\textsc{Hash-Search}(T, 76)$ would return NIL, incorrectly indicating that key 76 is not in the hash table.

However, if slot 2 is marked as DELETED, $\textsc{Hash-Search}(T, 76)$ finds key 76 in slot 3.



# Open addressing with deletion

## Pseudocode

No changes are needed in HASH-SEARCH.

Here are the other two dictionary operations:

```
HASH-DELETE$(T, k)$
$i$ = HASH-SEARCH$(T, k)$
if $i \neq$ NIL
    $T[i]$ = DELETED
```

```
HASH-INSERT$(T, k)$
$i = 0$
repeat
    $j = h(k, i)$
    if $T[j]$ == NIL or $T[j]$ == DELETED
        $T[j] = k$
        return $j$
    else $i = i + 1$
until $i == m$
error "hash table overflow"
```

## Uniform Hashing

### Generalisation of Simple Uniform Hashing

Hash functions used for open addressing should ideally perform **uniform hashing**, defined by two criteria:

1. For every fixed probe number $i$, the hash function $h(k, i)$ should perform simple uniform hashing, meaning that the hash value $h(k, i)$ of any random key $k$:
   - is equally likely to hash into any slot $0, 1, \ldots, m-1$.
   - is independent of the hash value of any other key.
2. The probe sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ is:
   - equally likely to be any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1\rangle$.
   - independent of any probe sequence $\langle h(k', 0), h(k', 1), \ldots, h(k', m-1)\rangle$ with $k \neq k'$.

Violating either criterion can cause clustering of keys in certain parts of the hash table, which in turn can degrade the performance of the hash table.

## Uniform Hashing Is Difficult to Implement

### Other Approaches Are Needed

True uniform hashing is difficult to implement. Most hash functions that are used in practice do not generate all of the $m!$ possible permutations.

In this lesson, we discuss three methods that guarantee that the probe sequence $\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$ is a permutation of $\langle 0, 1, \dots, m-1\rangle$ for each key $k$:

- Linear probing, which generates at most $m$ distinct probe sequences
- Quadratic probing, which also generates at most $m$ distinct probe sequences
- Double hashing, which generates at most $m^2$ distinct probe sequences

## Linear Probing

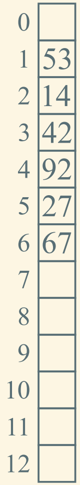
### Easy to Implement but Leads to Primary Clustering

Given an ordinary hash function $h' : U \to \{0, 1, \ldots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m-1$.

In the example on the right, keys were inserted in the sequence $\langle 42, 53, 14, 92, 27, 67 \rangle$, $h'(k) = k \bmod m$ and $m = 13$. The result exhibits **primary clustering**, whereby long sequences of occupied slots are created, leading to an increase in the average search time.



## Quadratic Probing

### Better than Linear Probing but Still Causes Secondary Clustering

**Quadratic probing** uses a hash function of the form

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

where $c_1$ and $c_2$ are positive integers.

In the example on the right, $c_1 = c_2 = 1$. As on the previous page, keys were inserted in the sequence $\langle 42, 53, 14, 92, 27, 67 \rangle$ and $h'(k) = k \bmod 13$.

If two keys have the same initial probe position (e.g. $53$, $14$, $92$ and $27$), then their probe sequences are the same. This phenomenon is called **secondary clustering**, which is milder than primary clustering but still undesirable.

| | |
|---|---|
| 0 | 92 |
| 1 | 53 |
| 2 | 67 |
| 3 | 42 |
| 4 | |
| 5 | |
| 6 | |
| 7 | 14 |
| 8 | 27 |
| 9 | |
| 10 | |
| 11 | |
| 12 | |

## Double Hashing

### Using Two Auxiliary Hash Functions

**Double hashing** uses a hash function of the form

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m,$$

where both $h_1$ and $h_2$ are auxiliary hash functions.

The value of $h_2(k)$ must be relatively prime to $m$ so that the entire hash table can be searched. This property can be established, for example, by letting $m$ be

- a power of 2 and designing $h_2$ so that it always produces an odd number.
- prime and letting

  $$h_1(k) = k \bmod m,$$
  $$h_2(k) = 1 + (k \bmod m'),$$

  where $m'$ is slightly less than $m'$ (e.g. $m-1$).

# Double Hashing

## No Clustering Despite Producing Only $m^2$ Distinct Probe Sequences

In the example on the right, $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 12)$. As before, keys were inserted in the sequence $\langle 42, 53, 14, 92, 27, 67 \rangle$. Note that we never had to probe any sequence beyond $i = 1$, which provides evidence that double hashing is neither subject to primary nor secondary clustering.

Although double hashing produces only $m^2$ out of the $m!$ possible probe sequences, its performance is practically as good as the ideal scheme of uniform hashing.

| | |
|---|---|
| 0 | |
| 1 | 53 |
| 2 | 67 |
| 3 | 42 |
| 4 | 14 |
| 5 | 27 |
| 6 | |
| 7 | |
| 8 | |
| 9 | |
| 10 | 92 |
| 11 | |
| 12 | |

## Running Time of Searching and Inserting Without Deleting

### Steep Increase as Load Factor Approaches 1 from below

One can show that the average time needed for searching an open-address hash table is

- $O\left(\frac{1}{1-\alpha}\right)$ if the search is unsuccessful.
- $O\left(\frac{1}{\alpha}\log\frac{1}{1-\alpha}\right)$ if the search is successful or an element is inserted,

where $\alpha = n/m$ is the load factor of the hash table, assuming that no entry has been deleted by the end of the program.

Deletions are more difficult to analyse because running times also depend on the number of DELETED entries. In general, it is better to resolve collisions by chaining than by open addressing when many entries must be deleted during the execution of a program.

## Running Time of Searching and Inserting without Deleting

### $O(1)$ if $\alpha$ Is Bounded From Above by a Constant $< 1$

As the hash fills up (i.e. $\alpha$ approaches 1 from below), both of the upper bounds $O\left(\frac{1}{1-\alpha}\right)$ and $O\left(\frac{1}{\alpha}\log\frac{1}{1-\alpha}\right)$ diverge. When $\alpha$ exceeds 1, open-addressing cannot be used for resolving collisions because there is insufficient space in the hash table.

However, if we can guarantee that $\alpha$ never exceeds a constant $< 1$, then searching and inserting only need $O(1)$ time.

To ensure that $\alpha$ remains small enough, we need a way to resize hash tables.

## Resizing Hash Tables

### Increase or Decrease Hash Table if Threshold Load Factors Are Reached

If the load factor $\alpha$ approaches 1 from below, the running time increases steeply when collisions are resolved by open addressing. When chaining is used instead of open addressing, the average running time increases more gently as a function of $\alpha$. Still, we only get performance guarantees if $\alpha$ does not exceed some constant $\alpha_+$.

Load factors near 0 are undesirable too because a fraction $1 - \alpha$ of the slots are unoccupied, which wastes space. Therefore, we want to keep $\alpha$ above some threshold $\alpha_-$.

Hence, we need strategies for increasing and decreasing hash tables.

## Resizing Hash Tables

### Choosing the Thresholds $\alpha_+$ and $\alpha_-$

From this point on, we assume that $\alpha_+ > 4\alpha_-$ so that we do not have to resize the hash table too frequently.

If collisions are resolved by open addressing, we must choose $\alpha_+ < 1$.

If collisions are resolved by chaining, $\alpha_+$ can be any positive number, but it is recommended to keep $\alpha_+ < 2$ to avoid having chains that are too long.

## Doubling the Size of a Hash Table When $\alpha > \alpha_+$

### Each Doubling Takes $O(m)$ Time

Suppose we double the number of slots in the hash table each time an insertion causes $\alpha$ to exceed $\alpha_+$. If the current table has $m/2$ slots and the next insertion increases $\alpha$ beyond $\alpha_+$, then we:

1. create a new hash table with $m$ slots.
2. generate a new hash function $h$ suitable for a table with $m$ slots.
3. rehash all $n \approx \alpha_+ m$ elements in the old hash table into the new hash table.
4. destroy the old hash table.

These steps require $O(m)$ time.

## Doubling the Size of a Hash Table When $\alpha > \alpha_+$

### Amortised $O(1)$ Running Time

Although an investment of $O(m)$ time is a substantial cost, we do not need to perform a doubling of the hash-table frequently.

In the worst case, we keep inserting one element after the next and never delete any. Then the next doubling occurs after $(\alpha_+ m)/2$ insertions. Thus, we must invest $O(m)$ time for doubling the number of slots only after $O(m)$ insertions.

Consequently, the running time of doubling divided by the number of insertions, also known as the 'amortised' running time of doubling, is $O(1)$.

## Halving the Size of a Hash Table When $\alpha < \alpha_-$

### Amortised $O(1)$ running time

With a similar argument, we can infer an upper bound for the amortised running time of halving the number of slots whenever a deletion causes the load factor to drop below $\alpha_-$.

If the table currently has $m$ slots, the time needed for halving is $O(m)$ because we must create a new hash table with $m/2$ slots, rehash $\alpha_- m$ elements and destroy the old hash table.

However, we only spend $O(m)$ time for halving after at least $\alpha_- m/2 = O(m)$ deletions. Thus, the amortised running time per deletion is only $O(1)$.

## Exercise
### Open Addressing

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88 and 59 into a hash table of length $m = 11$ using open addressing. Illustrate the result of inserting these keys using

(a) linear probing with the auxiliary hash function $h'(k) = k \bmod 11$.

(b) quadratic probing with $c_1 = 1$ and $c_2 = 3$ and the auxiliary hash function $h'(k) = k \bmod 11$.

(c) double hashing with $h_1(k) = k \bmod 11$ and $h_2(k) = 1 + (k \bmod 10)$.

## Solution to Exercise

### Open Addressing

| | (a) | (b) | (c) |
|---|---|---|---|
| 0 | 22 | 22 | 22 |
| 1 | 88 | | |
| 2 | | 88 | 59 |
| 3 | | 17 | 17 |
| 4 | 4 | 4 | 4 |
| 5 | 15 | | 15 |
| 6 | 28 | 28 | 28 |
| 7 | 17 | 59 | 88 |
| 8 | 59 | 15 | |
| 9 | 31 | 31 | 31 |
| 10 | 10 | 10 | 10 |

## Outlook and Conclusion

### Hash Tables

In this and the previous lesson, we discussed hash tables. We learnt that collisions, which occur when different keys are hashed to the same slot, can never be ruled out. There are two common techniques for resolving collisions: chaining and open addressing. They have fast average running times, but their worst-case running times can be slow.

In Section 11.5 of their book, Cormen et al. introduce ‘perfect hashing’, which gives worst-case performance guarantees when the set of keys is static (i.e. they are not inserted or deleted at run-time).

Among many other interesting hashing techniques, it is also worth mentioning ‘consistent hashing’, which ensures that, when a hash table is resized, only $n/m$ keys need to be remapped on average.