

# Computer Science 331

## Hash Tables: Open Addressing

Mike Jacobson

Department of Computer Science  
University of Calgary

Lecture #19

## Open Addressing

In a hash table with *open addressing*, all elements are stored in the hash table itself.

For  $0 \leq i < m$ ,  $T[i]$  is either

- an element of the dictionary being stored,
- NIL, or
- DELETED (to be described shortly!)

**Note:** The textbook refers to DELETED as a “dummy value.”

## Outline

- 1 Open Addressing
- 2 Operations
  - Search
  - Insert
  - Delete
- 3 Collision Resolution
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
  - Analysis
- 4 Summary

## Example

	0	1	2	3	4	5	6	7
T:	NIL	25	2	NIL	12	NIL	14	22

- $U = \{1, 2, \dots, 200\}$
- $m = 8$
- $T$  : as shown above
- $h_0$  : Function such that

$$h_0 : \{1, 2, \dots, 200\} \rightarrow \{0, 1, \dots, 7\}$$

Eg.  $h_0(k) = k \bmod 8$  for  $k \in U$ .

$h_0$  used here for *first* try to place key in table.

## New Definition of a Hash Function

We may need to make more than one attempt to find a place to insert an element.

We'll use hash functions of the form

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

$h(k, i)$ : Location to choose to place key  $k$  on an  $i^{\text{th}}$  attempt to insert the key, if the locations examined on attempts  $0, 1, \dots, i-1$  were already full.

This location is not used if already occupied (i.e., if not NIL or DELETED)

Function  $h_0(k)$  from previous slide was:  $h(k, 0)$

## Probe Sequence

The sequence of addresses

$$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

is called the **probe sequence** for key  $k$

**Nonstandard Requirement:** some references, including *Introduction to Algorithms*, require that

$$\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$$

is always a *permutation* of the integers between 0 and  $m-1$ .

- This is highly desirable condition... but it is not satisfied by some of the hash functions that are frequently used.
- Hence, we will *not* assume it.

## Search Pseudocode

**Search(k)**

$i = 0$

**repeat**

$j = h(k, i)$

**if**  $T[j] == k$  **then**

**return**  $j$

**end if**

$i = i + 1$

**until or**

**return** Report that  $k$  is not found

## Example: Search for 9

	0	1	2	3	4	5	6	7
T:	NIL	25	2	NIL	12	NIL	14	22

$h$ : function such that

$$h : U \times \{0, 1, \dots, 7\} \rightarrow \{0, 1, \dots, 7\}$$

and  $h(k, i) = k + i \bmod 8$  for  $k \in U$  and  $0 \leq i \leq 7$ .

Probes when Searching for 9 :

- 
- 
-

## Insert Pseudocode: One Algorithm

```

Insert( $k$ )
   $i = 0$ 
  repeat
     $j = h(k, i)$ 
    if then
       $T[j] = k$ 
      return (key inserted at index  $j$ )
    else if  $T[j] == k$  then
      return (report that  $k$  is already used in the dictionary)
    else
       $i = i + 1$ 
    end if
  until
  return (report a hash table overflow (key not added))

```

## Example: Insert 1

	0	1	2	3	4	5	6	7
T:	NIL	25	2	NIL	12	NIL	14	22

Probe sequence:

- 
- 
- 

## Delete Pseudocode

```

Delete( $k$ )
   $i = 0$ 
  repeat
     $j = h(k, i)$ 
    if  $T[j] == k$  then
      return (reporting that  $k$  was deleted)
    end if
     $i = i + 1$ 
  until  $T[j] == \text{NIL}$  or  $i == m$ 
  return (report that  $k$  was not found)

```

**Question:** Why not set  $T[j] = \text{NIL}$ , above?

## Example: Delete 22

	0	1	2	3	4	5	6	7
T:	NIL	25	2	NIL	12	NIL	14	22

Probe sequence:

- 
- 

Insert 30?

- 
- 
-

## Complication

The “value” DELETED is never overwritten.

- once  $T[j]$  is marked DELETED it is not used to store an element of the dictionary!
- Eventually a hash table might report overflows on insertions, even if the the dictionary it stores is empty!

Unfortunately, cannot simply overwrite DELETED with NIL:

- can cause searches to fail when they should succeed because **insert** terminates when a NIL entry is reached

## Insert: Another Algorithm

### Exercise:

- Write another version of the “Insert” algorithm that allows “DELETED” to be overwritten with an input key  $k$
- *Don't Forget:* Make sure  $k$  can never be stored in two or more locations at the same time!

How to do this:

- 
- 

## Linear Probing

Let  $h(k) = h(k, 0)$

### Simple Form of Linear Probing:

$$h(k, i) = h(k) + i \bmod m \quad \text{for } i \geq 1$$

### Generalization:

$$h(k, i) = h(k) + ci \bmod m \quad \text{for } i \geq 1$$

for some *nonzero constant*  $c$  (not depending on  $k$  or  $i$ )

## Strengths and Weaknesses

### Strengths:

- If  $c = 1$  (or  $\gcd(c, m) = 1$ ) then the probe sequence *is* a permutation of  $0, 1, \dots, m - 1$
- This hash function is easy to compute: For  $i \geq 1$

$$h(k, i) = h(k, i - 1) + c \bmod m .$$

- If linear probing is used, you can delete from a hash table without using DELETED at all, but the algorithm is more complicated.

### Weakness:

- *Primary Clustering:*

## Quadratic Probing

Let  $h(k) = h(k, 0)$

**Simple form of Quadratic Probing:**

$$\begin{aligned} h(k, i) &= h(k) + i^2 \bmod m \\ &= h(k, i-1) + 2i - 1 \bmod m \quad \text{for } i \geq 1 \end{aligned}$$

**Generalization:**

$$h(k, i) = h(k) + c_0 i + c_1 i^2$$

for a constant  $c_0$  and a *nonzero* constant  $c_1$ .

## Double Hashing

Suppose  $h_0$  and  $h_1$  are both hash functions depending only on  $k$ , i.e.,

$$h_0, h_1 : U \rightarrow \{0, 1, \dots, m-1\}$$

and such that

$$h_1(k) \not\equiv 0 \pmod{m}$$

for every key  $k$ .

**Double Hashing:**

$$h(i, k) = (h_0(k) + i h_1(k)) \bmod m$$

Eg.  $h_0(k) = k \bmod m$ ,  $h_1(k) = 1 + (k \bmod m - 1)$

## Strengths and Weaknesses

**Strengths:**

- If  $\gcd(m, c) = 1$  and  $m \geq 3$  is prime then the probe sequence includes (slightly) more than half of  $0, 1, \dots, m-1$
- The hash function is easy to compute:

$$h(k, i) - h(k, i-1) = \alpha_0 + \alpha_1 i$$

for constants  $\alpha_0$  and  $\alpha_1$

**Weakness:**

- *Secondary Clustering:*

## Strengths and Weaknesses

**Strengths:**

- If  $m$  is prime and  $\gcd(h_1(k), m) = 1$  then the probe sequence for  $k$  is a permutation of  $0, 1, \dots, m-1$
- Analysis and experimental results both suggest extremely good expected performance

**Weakness:**

- A bit more complicated than linear (or quadratic) probing

## Summary

Deletions complicate things:

- Hash tables with chaining are often superior unless deletions are extremely rare (or do not happen at all)

Expected number of probes for searches is too high for these tables to be useful when  $\lambda$  is close to one, where

$$\lambda = \frac{\text{number of locations storing keys or DELETED}}{m}$$

Remaining slides show results concerning tables produced by inserting  $n$  keys  $k_1, k_2, \dots, k_n$  into an empty table (so  $\lambda = n/m$ )

## Analysis of Linear Probing (with $c = 1$ )

**Assumption:** Each of the  $m^n$  sequences

$$h_0(k_1), h_0(k_2), \dots, h_0(k_n)$$

of *initial* probes are assumed to be equally likely.

Expected number of probes is approximately

$$\begin{cases} \frac{1}{2} \left( 1 + \left( \frac{1}{1-\lambda} \right)^2 \right) & \text{unsuccessful search} \\ \frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right) & \text{successful search} \end{cases}$$

*Reference:* Knuth, Volume 3

## The Best We Can Hope For

**Uniform Hashing Assumption:** Each of the  $m!$  permutations is equally likely as a probe sequence for a key.

- In some sense, the best we can hope for
- Completely Unrealistic! Only  $m$  of these probe sequences are possible using linear or quadratic probing; only (approximately)  $m^2$  are possible with double hashing

Expected number of probes under this assumption: approximately

$$\begin{cases} \frac{1}{1-\lambda} & \text{(unsuccessful search)} \\ \frac{1}{\lambda} \ln \frac{1}{1-\lambda} & \text{(successful search)} \end{cases}$$

*References:* Textbook; Knuth, Volume 3

## Reference for Additional Results

*Knuth:* “Exhaustive tests show that double hashing with two independent hash functions  $h_0$  and  $h_1$  behaves essentially like uniform hashing, for all practical purposes.”

For additional details, and more results, see

Knuth, *The Art of Computer Programming*, Volume 3

# Summary

## Advantages of Open Addressing:

- does not have the storage overhead due to pointers (required for the linked lists in chaining)
- better cache utilization during probing if the entries are small
- good choice when entry sizes are small

## Advantages of Chaining:

- insensitive to clustering (only require good hash function)
- grows dynamically and fills up gracefully (chains all grow equally long on average), *unlike* open addressing
- good choice when entries are large and load factor can be high