Computer Science 331

Hash Tables with Open Addressing

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #20

Outline

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."

Example

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

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

$$h_0 : \{1, 2, \ldots, 200\} \to \{0, 1, \ldots, 7\}$$

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

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

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

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

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

Initial Requirement:

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

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.
- We will disucss what happens in the general case later in these notes.

Search Pseudocode

The following algorithm either returns an integer `i` such that `T[i]` is equal to `k`, or throws a `notFoundException` (because `k` is not stored in the hash table).

```
int search (key k) {
  i = 0;
  do {
    j = h(k, i);
    if (T[j] == k) {
      return j;
    };
    i++;
  } while ((T[j] != nil) && (i < m));
  throw notFoundException;
}
```


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

$$h : U \times \{0, 1, \ldots, 7\} \to \{0, 1, \ldots, 7\}$$

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

The following algorithm either reports where the key k has been inserted or throws an appropriate exception

```
int insert (key k) {
  i = 0;
  while (i < m) {
    j = h(k, i);
    if (T[j] == nil) {
      T[j] = k; return j;
    } else {
      if (T[j] == k) { throw foundException; };
    };
    i++;
  };
  throw tableFullException;
}
```


Example: Insert 1

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

Probe sequence:

-
-
-

Delete Pseudocode

The next method either deletes k or returns an exception to indicate that k was not in the table.

```
void delete (key k) {
  i=0;
  do {
    j = h(k, i);
    if (T[j] == k) {
      T[j] = deleted; return;
    };
    i++;
  } while ((T[j] != nil) && (i < m));
  throw notFoundException;
}
```

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:

-
-

More General Probe Sequences

As previously noted it is not always true, in practice, that the sequence of addresses

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

is a permutation

Good News: In this more general situation, it is still true that

- the `search` algorithm will return an integer `i` such that `T[i]` is equal to `k` if the given key `k` is stored in the table
- the exceptions `FoundException` and `notFoundException` (used in the algorithms given previously) will still be thrown (precisely) when they are needed

Bad News: `tableFullException` might now be thrown even though there are still some `nil` entries in the table

Linear Probing

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

Simple Form of Linear Probing:

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

Generalization:

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

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 $0, 1, \ldots, m-1$
- 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:

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

Generalization:

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

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

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 $0, 1, \ldots, m-1$
- The hash function is easy to compute:

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

for constants α_0 and α_1

Weakness:

- *Secondary Clustering:*

Double Hashing

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

$$h_0, h_1 : U \to \{0, 1, \ldots, m-1\}$$

and such that

$$h_1(k) \not\equiv 0 \bmod m$$

for every key k.

Double Hashing:

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

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

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 $0, 1, \ldots, m-1$
- 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 λ 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 $k_1, k_2, \ldots, k_n$ into an empty table (so $\lambda = n/m$)

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

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

Assumption: Each of the m^n sequences

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

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

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