Computer Science 331

Hash Tables

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #23-24

Outline

1. Introduction
2. Hash Tables with Chaining
   - Overview
   - Cost Analysis
   - A Variation
3. Hash Tables with Open Addressing
   - Overview
   - Operations
   - Collision Resolution
   - Analysis
4. Summary
5. References

Common Situation

We wish to use a dictionary (or mapping), under the following circumstances:

- The "universe" of possible values for keys is extremely large.
- We have a much smaller bound on the (maximal) size of the dictionary we will need to support.
- The only dictionary operations we need are
  - initialization of an empty dictionary,
  - searches for items in the dictionary,
  - insertions of new items into the dictionary,
  - deletions of items from the dictionary.

What is a Hash Table?

A **hash table** is a generalization of an ordinary array.

Features:

- Array size is generally chosen to be comparable to (perhaps, a small *multiple* of) our bound on dictionary size
- Worst-case performance is generally poor
- However, the average-case performance is extremely good — better than that of the other implementations of a dictionary we have considered!

Notation and Definitions

U: **Universe**: The set of possible values for keys

m: **Table Size**: The size of the array used to build a hash table

T: The array that is used.

h: **Hash Function**: A function

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

used to map keys to array locations

Idea: try to store element x with key k in location $T[h(k)]$.

General Difficulty: Collisions

More terminology:

- A key k **hashes to** an array location ℓ if $h(k) = \ell$.
- A **collision** occurs if two keys k_1 and k_2 (used in the dictionary) hash to the same location, that is,

$$h(k_1) = h(k_2) \ .$$

Note: Collisions are unavoidable if the size of the dictionary is greater than the table size.

There are several different kinds of hash tables that use different ways to deal with collisions. We will study:

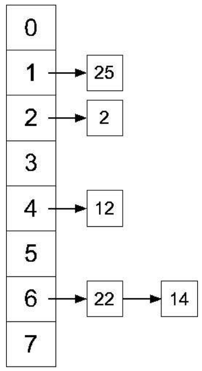
- chaining
- open addressing

Collision Resolution with Chaining

In a hash table with **chaining**:

- we put all the keys (used in the dictionary) that hash to the same location ℓ into a linked list.
- For $0 \le \ell < m$, $T[\ell]$ is a pointer/reference to the head of the linked list for location ℓ.
- *Abuse of Notation:* Sometimes $T[\ell]$ will be used as the name for the above linked list (instead of a pointer to it).

Example



U: $\{1, 2, \ldots, 200\}$

m: 8

T: As shown to the left.

h: Function such that

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

eg. $h(k) = k \bmod 8$ for $k \in U$.

Dictionary Operations

Search for an item with key k;

- Search for k in the linked list $T[h(k)]$.

Insertion of an item I with key k;

- Search for k in the linked list $T[h(k)]$.
- If the search was unsuccessful, insert I onto the **front** of this linked list.

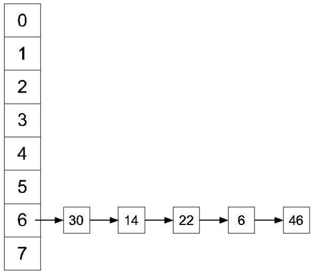
Deletion of an item with key k;

- Perform a deletion of an item with key k from the linked list $T[h(k)]$.

Worst-Case Analysis

Cost of an operation involving a key k is essentially the cost the same operation involving k, using the linked list $T[h(k)]$.

Problem:



It is possible for *all* dictionary items to be part of this linked list!

Average Case Analysis

We will consider the average cost of dictionary operations when a hash table is used to represent a dictionary with n elements.

The average cost of these operations depends on

- the likelihood of each kind of operation, and
- the *shape* of the hash table.

The shape of the hash table only depends on the *locations* to which keys are hashed — not on the values of the keys, themselves.

Simple Uniform Hashing

Assumption: **Simple Uniform Hashing:**

- Each key is hashed to location ℓ with the same probability, $\frac{1}{m}$, for $0 \leq \ell < m$.
- Furthermore, each key is hashed to a location *independently* of where any other key is hashed to.

That is: If $k_1, k_2, \dots, k_n$ are the keys in the dictionary then

$$h(k_1) = \ell_1 \quad \text{and} \quad h(k_2) = \ell_2 \quad \text{and} \quad \cdots \quad \text{and} \quad h(k_n) = \ell_n$$

with probability $(1/m)^n$, for *each* choice of locations $\ell_1, \ell_2, \dots, \ell_n$.

Load Factor

Load Factor of T: The average λ of the lengths of the linked lists (or "chains") $T[0], T[1], \ldots, T[m-1]$.

Claim:

$\lambda = n/m$ (hash table has m locations, dictionary has n elements).

Proof.

Suppose $T[i]$ has length n_i for $0 \leq i < m$.

- Then $\lambda = \frac{1}{m}(n_0 + n_1 + \cdots + n_{m-1})$ (by definition).
- However, since each key is hashed to exactly one location, and there are n keys, $n_0 + n_1 + \cdots + n_{m-1} = n$, so $\lambda = n/m$. □

Average Case Analysis: Summary

Expected Numbers of Comparisons Required:

Unsuccessful Search for a key k :

- Assumption: No additional assumptions required.
- Expected Cost: $\lambda = \frac{n}{m}$

Successful Search:

- Assumption: Search for each key with probability $\frac{1}{n}$
- Expected Cost: $1 + \frac{\lambda}{2} - \frac{\lambda}{2n} \in \Theta(1 + \lambda)$

Insertion of **New** Element: same as unsuccessful search

Deletion of Existing Element: same as successful search

A Variation

Suppose that the universe U is ordered, so that we can also ask whether $k_1 \leq k_2$ for any two keys k_1 and k_2.

In this case we could maintain the keys in each of our lists in *sorted order*.

- *Worst case* costs for operations are unchanged.
- *Expected cost for a successful search* using the usual assumptions is also unchanged
- However, the expected cost for an *unsuccessful* search is somewhat reduced — because we can use the list ordering to end an unsuccessful search a bit earlier.
- The overhead to maintain sorted order is insignificant, so this optimization is worthwhile.

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 $0 \le i \le 7$.

Probes when Searching for 9 :

- $i = 0 : h(k, 0) = k + 0 \bmod 8 = 1, \ T[1] = 25$
- $i = 1 : h(k, 1) = k + 1 \bmod 8 = 2, \ T[2] = 2$
- $i = 2 : h(k, 2) = k + 2 \bmod 8 = 3, \ T[3] = \text{NIL}$

Report that 9 is not in the hash table

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:

- $i = 0 : h(k, 0) = k + 0 \bmod 8 = 1, \ T[1] = 25$
- $i = 1 : h(k, 1) = k + 1 \bmod 8 = 2, \ T[2] = 2$
- $i = 2 : h(k, 2) = k + 2 \bmod 8 = 3, \ T[3] = \text{NIL}$

Set $T[3] = 1$

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:

- $i = 0 : h(k, 0) = k + 0 \bmod 8 = 6, \; T[6] = 14$
- $i = 1 : h(k, 1) = k + 1 \bmod 8 = 7, \; T[7] = 22$

Set $T[7] = \text{DEL}$

Insert 30?

- $i = 0 : h(k, 0) = k + 0 \bmod 8 = 6, \; T[6] = 14$
- $i = 1 : h(k, 0) = k + 1 \bmod 8 = 7, \; T[7] = \text{DEL}$
- $i = 2 : h(k, 0) = k + 2 \bmod 8 = 0, \; T[0] = \text{NIL}$

Set $T[0] = 30$

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:

- if an entry marked DELETED is encountered during probing, store it's index
- once NIL is reached, indicating that an entry with the given key is not currently in the list, overwrite the previously-detected DELETED entry

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 $i \geq 1$

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

Weakness:

- *Primary Clustering:* long runs of occupied slots can build up, causing long probe searches

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:* two keys with the same initial probe position have the same probe sequences

Double Hashing

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

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

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 $0, 1, \dots, m-1$
- 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 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 $k_1, k_2, \dots, k_n$ 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.

- 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

References

- *Introduction to Algorithms*, Chapter 11 — additional information about hash tables (including much of the material in these notes)
- *Introduction to Algorithms*, Appendix C — more information about useful concepts from probability and statistics
- **Data Structures: Abstraction and Design Using Java**, Chapter 7.3 and 7.4