CSCI 333
Searching

Chapter 9

1 and 6 November 2002

Notes with the dark blue background were prepared by the textbook author

Clifford A. Shaffer
Department of Computer Science
Virginia Tech

Search

Given: Distinct keys k_1, k_2, …, k_n and collection T of n records of the form
(k_1, I_1), (k_2, I_2), …, (k_n, I_n)
where I_j is the information associated with key k_j for 1 <= j <= n.

Search Problem: For key value K, locate the record (k_j, I_j) in T such that $k_j = K$.

Searching is a systematic method for locating the record(s) with key value $k_j = K$.

Successful vs. Unsuccessful

A successful search is one in which a record with key $k_j = K$ is found.

An unsuccessful search is one in which no record with $k_j = K$ is found (and presumably no such record exists).

Approaches to Search

1. Sequential and list methods (lists, tables, arrays).

2. Direct access by key value (hashing)

3. Tree indexing methods.

Searching Ordered Arrays

Sequential Search

Binary Search

Dictionary Search

Lists Ordered by Frequency

Order lists by (expected) frequency of occurrence.

- Perform sequential search

Cost to access first record: 1
Cost to access second record: 2

Expected search cost:

$$\overline{C}_n = 1p_1 + 2p_2 + \ldots + np_n.$$

Examples(1)

(1) All records have equal frequency.

$$\overline{C}_n = \sum_{i=1}^{n} i/n = (n+1)/2$$

Examples(2)

(2) Exponential Frequency

$$p_i = \begin{cases} 1/2^i & \text{if } 1 \le i \le n-1 \\ 1/2^{n-1} & \text{if } i = n \end{cases}$$

$$\overline{C}_n \approx \sum_{i=1}^{n} (i/2^i) \approx 2.$$

Zipf Distributions

Applications:

- Distribution for frequency of word usage in natural languages.
- Distribution for populations of cities, etc.

$$\overline{C}_n = \sum_{i=1}^{n} i/i\mathrm{H}_n = n/\mathrm{H}_n \approx n/\log_e n.$$

80/20 rule:

- 80% of accesses are to 20% of the records.
- For distributions following 80/20 rule,

$$\overline{C}_n \approx 0.122n.$$

Self-Organizing Lists

Self-organizing lists modify the order of records within the list based on the actual pattern of record accesses.

Self-organizing lists use a heuristic for deciding how to reorder the list. These heuristics are similar to the rules for managing buffer pools.

Heuristics

1. Order by actual historical frequency of access. (Similar to LFU buffer pool replacement strategy.)
2. When a record is found, swap it with the first record on list.
3. Move-to-Front: When a record is found, move it to the front of the list.
4. Transpose: When a record is found, swap it with the record ahead of it.

Text Compression Example

Application: Text Compression.

Keep a table of words already seen, organized via Move-to-Front heuristic.

- If a word not yet seen, send the word.
- Otherwise, send (current) index in the table.

The car on the left hit the car l left.
The car on 3 hit 3 5 l 5.

This is similar in spirit to Ziv-Lempel coding.

Searching in Sets

For dense sets (small range, high percentage of elements in set).

Can use logical bit operators.

Example: To find all primes that are odd numbers, compute:

0011010100010100 & 0101010101010101

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

Hashing (1)

Hashing: The process of mapping a key value to a position in a table.

A hash function maps key values to positions. It is denoted by ***h***.

A hash table is an array that holds the records. It is denoted by **HT**.

HT has *M* slots, indexed form 0 to *M*-1.

Hashing (2)

For any value *K* in the key range and some hash function ***h***, ***h***(*K*) = *i*, 0 <= *i* < *M*, such that key(**HT**[*i*]) = *K*.

Hashing is appropriate only for sets
Though you could add counts

Good for both in-memory and disk-based applications.

Answers the question "What record, if any, has key value K?"

Simple Examples

(1) Store the n records with keys in range 0 to *n*-1.
- Store the record with key *i* in slot *i*.
- Use hash function ***h***(*K*) = *K*.

(2) More reasonable example:
- Store about 1000 records with keys in range 0 to 16, 383.
- Impractical to keep a hash table with 16,384 slots.
- We must devise a hash function to map the key range to a smaller table.

Collisions (1)

Given: hash function h with keys k_1 and k_2. β is a slot in the hash table.

If $\boldsymbol{h}(k_1) = \beta = \boldsymbol{h}(k_2)$, then k_1 and k_2 have a collision at β under ***h***.

Search for the record with key *K*:
1. Compute the table location ***h***(*K*).
2. Starting with slot ***h***(*K*), locate the record containing key *K* using (if necessary) a collision resolution policy.

Collisions (2)

Collisions are inevitable in most applications.

- Example: 23 people are likely to share a birthday.

Hash Functions (1)

A hash function MUST return a value within the hash table range.

To be practical, a hash function SHOULD evenly distribute the records stored among the hash table slots.

Ideally, the hash function should distribute records with equal probability to all hash table slots. In practice, success depends on distribution of actual records stored.

Hash Functions (2)

If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.

If we have knowledge of the incoming distribution, use a distribution-dependent hash function.

Examples (1)

```
int h(int x) {
  return(x % 16);
}
```

This function is entirely dependent on the lower 4 bits of the key.

Mid-square method: Square the key value, take the middle *r* bits from the result for a hash table of 2^r slots.

Examples (2)

For strings: Sum the ASCII values of the letters and take results modulo *M*.

```
int h(char* x) {
  int i, sum;
  for (sum=0, i=0; x[i] != '\0'; i++)
    sum += (int) x[i];
  return(sum % M);
}
```

This is only good if the sum is large compared to *M*.

Examples (3)

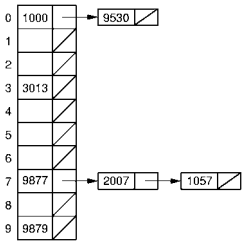
ELF Hash: From Executable and Linking Format (ELF), UNIX System V Release 4.

```
int ELFhash(char* key) {
  unsigned long h = 0;
  while(*key) {
    h = (h << 4) + *key++;
    unsigned long g = h & 0xF0000000L;
    if (g) h ^= g >> 24;
    h &= ~g;
  }
  return h % M;
}
```

Open Hashing

What to do when collisions occur?

Open hashing treats each hash table slot as a bin.



Bucket Hashing

Divide the hash table slots into buckets.

- Example: 8 slots/bucket.

Include an overflow bucket.

Records hash to the first slot of the bucket, and fill bucket. Go to overflow if necessary.

When searching, first check the proper bucket. Then check the overflow.

Similar to N-way set associate caches used in computer architecture

Closed Hashing

Closed hashing stores all records directly in the hash table.

Each record *i* has a home position $\boldsymbol{h}(k_i)$.

If another record occupies *i*'s home position, then another slot must be found to store *i*.

The new slot is found by a collision resolution policy.

Search must follow the same policy to find records not in their home slots.

Collision Resolution

During insertion, the goal of collision resolution is to find a free slot in the table.

Probe sequence: The series of slots visited during insert/search by following a collision resolution policy.

Let $\beta_0 = \boldsymbol{h}(K)$. Let $(\beta_0, \beta_1, \ldots)$ be the series of slots making up the probe sequence.

Space requirements

- Assume
  - N records
  - M elements in hash table
  - R is record size
  - P is pointer size
- Size of storage for
  - open hashing – records stored in table
  - open hashing – pointers stored in table
  - close hashing

Insertion

```
// Insert e into hash table HT
template <class Key, class Elem,
          class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::
hashInsert(const Elem& e) {
  int home;        // Home position for e
  int pos = home = h(getkey(e)); // Init
  for (int i=1;
       !(EEComp::eq(EMPTY, HT[pos])); i++) {
if (EEComp::eq(e, HT[pos]))
     return false; // Duplicate
  }
  HT[pos] = e;      // Insert e
  return true;
}
```

Search

```
// Search for the record with Key K
template <class Key, class Elem,
          class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::
hashSearch(const Key& K, Elem& e) const {
  int home;           // Home position for K
  int pos = home = h(K); // Initial posit
  for (int i = 1; !KEComp::eq(K, HT[pos]) &&
           !EEComp::eq(EMPTY, HT[pos]); i++)
    pos = (home + p(K, i)) % M; // Next
  if (KEComp::eq(K, HT[pos])) { // Found it
    e = HT[pos];
    return true;
  }
  else return false; // K not in hash table
}
```

Probe Function

Look carefully at the probe function p().

```
pos = (home + p(getkey(e), i)) % M;
```

Each time `p()` is called, it generates a value to be added to the home position to generate the new slot to be examined.

`p()` is a function both of the element's key value, and of the number of steps taken along the probe sequence.

- Not all probe functions use both parameters.

Linear Probing

Use the following probe function:

```
p(K, i) = i;
```

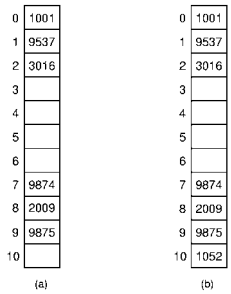
Linear probing simply goes to the next slot in the table.

- Past bottom, wrap around to the top.

To avoid infinite loop, one slot in the table must always be empty.

Linear Probing Example

Primary Clustering: Records tend to cluster in the table under linear probing since the probabilities for which slot to use next are not the same for all slots.



Improved Linear Probing

Instead of going to the next slot, skip by some constant *c*.

- Warning: Pick *M* and *c* carefully.

The probe sequence SHOULD cycle through all slots of the table.

- Pick *c* to be relatively prime to *M*.

There is still some clustering

- Ex: c=2, $\boldsymbol{h}(k_1) = 3$; $\boldsymbol{h}(k_2) = 5$.
- Probe sequences for k_1 and k_2 are linked together.

Pseudo-Random Probing(1)

The ideal probe function would select the next slot on the probe sequence at random.

An actual probe function cannot operate randomly. (Why?)

Pseudo-Random Probing(2)

- Select a (random) permutation of the numbers from 1 to M-1:

  $r_1, r_2, \ldots, r_{M-1}$
- All insertions and searches use the same permutation.

Example: Hash table size of M = 101
- r_1=2, r_2=5, r_3=32.
- $\boldsymbol{h}(k_1)$=30, $\boldsymbol{h}(k_2)$=28.
- Probe sequence for k_1:
- Probe sequence for k_2:

Quadratic Probing

Set the i'th value in the probe sequence as

$$\boldsymbol{h}(K, i) = i^2;$$

Example: M=101
- $\boldsymbol{h}(k_1)$=30, $\boldsymbol{h}(k_2)$ = 29.
- Probe sequence for k_1 is:
- Probe sequence for k_2 is:

Secondary Clustering

Pseudo-random probing eliminates primary clustering.

If two keys hash to the same slot, they follow the same probe sequence. This is called secondary clustering.

To avoid secondary clustering, need probe sequence to be a function of the original key value, not just the home position.

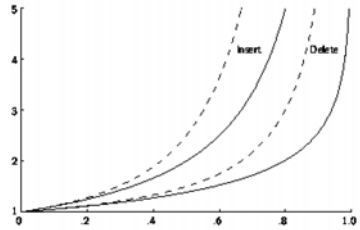
Double Hashing

Be sure that all probe sequence constants are relatively prime to M.
- This will be true if M is prime, or if M=2^m and the constants are odd.

Example: Hash table of size M=101
- $\boldsymbol{h}(k_1)$=30, $\boldsymbol{h}(k_2)$=28, $\boldsymbol{h}(k_3)$=30.
- $\boldsymbol{h}_2(k_1)$=2, $\boldsymbol{h}_2(k_2)$=5, $\boldsymbol{h}_2(k_3)$=5.
- Probe sequence for k_1 is:
- Probe sequence for k_2 is:
- Probe sequence for k_3 is:

Analysis of Closed Hashing

The load factor is α = N/M where N is the number of records currently in the table.



Deletion

Deleting a record must not hinder later searches.

We do not want to make positions in the hash table unusable because of deletion.

Tombstones (1)

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a tombstone.

A tombstone will not stop a search, but that slot can be used for future insertions.

Tombstones (2)

Unfortunately, tombstones add to the average path length.

Solutions:

1. Local reorganizations to try to shorten the average path length.
2. Periodically rehash the table (by order of most frequently accessed record).