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 information associated with key k_j for $1 \leq j \leq n$.

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 (or records) with key value $k_j = K$.

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

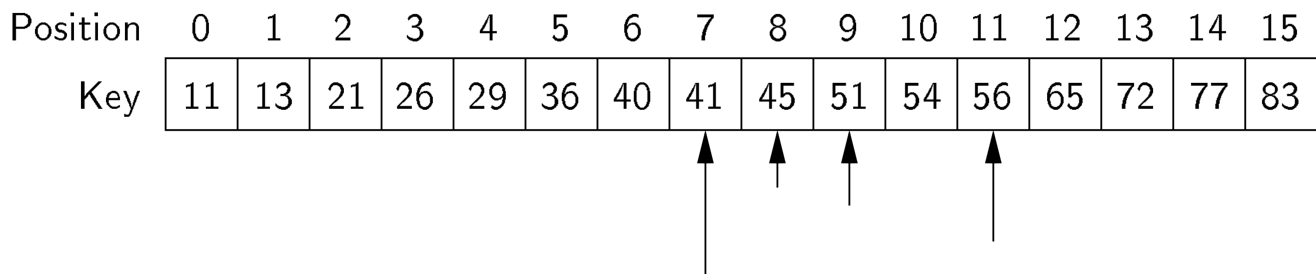
1. Sequential and list methods (lists, tables, arrays).
2. Direct access by key value (hashing).
3. Tree indexing methods.

Searching Ordered Arrays

Sequential Search

Binary Search

```
static int binary(int K, int[] array,
                  int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;   // l and r are beyond array bounds
  while (l+1 != r) { // Stop when l and r meet
    int i = (l+r)/2; // Look at middle of subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```



Dictionary Search

Lists Ordered by Frequency

Order lists by (expected) frequency of occurrance.

- Perform sequential search.

Cost to access first record: 1

Cost to access second record: 2

Expected search cost:

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

Example: all records have equal frequency

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

Example: Exponential frequency

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

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

Zipf Distributions

Applications:

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

Definition: Zipf frequency for item i in the distribution for n records as $1/i\mathcal{H}_n$.

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

80/20 rule: 80% of the accesses are to 20% of the records.

For distributions following the 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 access.

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

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

Example of Self-Organizing Tables

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 the (current) index in the table.

The car on the left hit the car I left.

The car on 3 left hit 3 5 I 5.

This is similar in spirit to Ziv-Lempel coding.

Searching in Sets

For dense sets (small range, many 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

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 T.

The hash table has M slots, indexed from 0 to $M - 1$.

For any value K in the key range and some hash function **h**,
$\mathbf{h}(K) = i, 0 \leq i < M$, such that $T[i]$.key() $= K$.

Hashing (continued)

Hashing is appropriate only for sets (no duplicates).

Good for both in-memory and disk based applications.

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

Example: Store the n records with keys in range 0 to $n-1$.

- Store the record with key i in slot i.
- Use hash function $\mathbf{h}(K) = K$.

Collisions

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.

Given: hash function **h** and keys k_1 and k_2.
β is a slot in the hash table.
If $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$, 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 $\mathbf{h}(K)$.
2. Starting with slot $\mathbf{h}(K)$, locate the record containing key K using (if necessary) a **collision resolution policy**.

Collisions are inevitable in most applications.

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

Hash Functions

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 the distribution of the actual records stored.

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 knowlege of the incoming distribution, use a distribution-dependant hash function.

Example Hash Functions

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

This function is entirely dependant 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.

Sum the ASCII values of the letters and take results modulo M.

```
static int h(String x, int M) {
  int i, sum;
  for (sum=0, i=0; i<x.length(); i++)
    sum += (int)x.charAt(i);
  return(sum % M);
}
```

ELF Hash

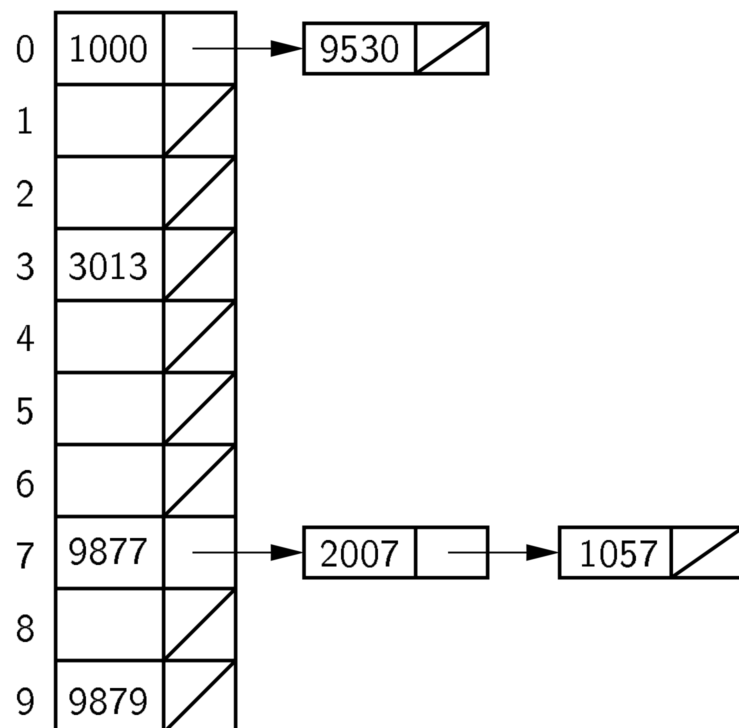
From Executable and Linking Format (ELF), UNIX System V Release 4.

```
static long ELFhash(String key, int M) {
  long h = 0;
  for (int i=0; i<key.length(); i++) {
    h = (h << 4) + (int) key.charAt(i);
    long g = h & 0xF0000000L;
    if (g != 0) 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.

Closed Hashing

Closed hashing stores all records directly in the hash table.

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

If i is to be inserted and another record already 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 $\beta_0 = \mathbf{h}(K)$. Let $(\beta_0, \beta_1, ...)$ be the series of slots making up the probe sequence.

```
void hashInsert(ELEM R) { // Insert R into hash table T
  int home;                    // Home position for R
  int pos = home = h(key(R));// Initial pos on sequence
  for (int i=1; key(T[pos]) != EMPTY; i++) {
    pos = (home + p(key(R), i)) % M; // Next slot
    if (key(T[pos]) == key(R)) ERROR; // No duplicates
  }
  T[pos] = R;                   // Insert R
}

ELEM hashSearch(KEY K) { // Search for record w/ key K
  int home;               // Home position for K
  int pos = home = h(K); // Initial pos on sequence
  for (int i = 1; (key(T[pos]) != K) &&
                  (key(T[pos]) != EMPTY); i++)
    pos = (home + p(K, i)) % M; // Next pos on sequence
  if (key(T[pos] == K)) return T[pos];  // Found it
  else return UNSUCCESSFUL;     // K not in hash table
}
```

Linear Probing

Use the probe function

```
int p(int K, int i) { return i; }
```

This is called **linear probing**.

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

If the bottom is reached, wrap around to the top.

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

Linear Probing Example

	(a)	(b)
0	1001	1001
1	9537	9537
2	3016	3016
3		
4		
5		
6		
7	9874	9874
8	2009	2009
9	9875	9875
10		1052

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.

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.

- Example: $c = 2$. $\mathbf{h}(k_1) = 3$. $\mathbf{h}(k_2) = 5$.
- The probe sequences for k_1 and k_2 are linked together.

Pseudo Random Probing

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:

- 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 of size $M = 101$

- $r_1 = 2, r_2 = 5, r_3 = 32.$
- $\mathbf{h}(k_1) = 30, \mathbf{h}(k_2) = 28.$
- Probe sequence for k_1 is:
- Probe sequence for k_2 is:

Quadratic Probing

Set the i'th value in the probe sequence as

$$(\mathbf{h}(K) + i^2) \bmod M.$$

Example: M = 101.

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

Double Hashing

Pseudo random probing eliminates primary clustering.

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

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

Double hashing:

$$\mathbf{p}(K, i) = i * \mathbf{h}_2(K) \text{ for } 0 \leq i \leq M - 1.$$

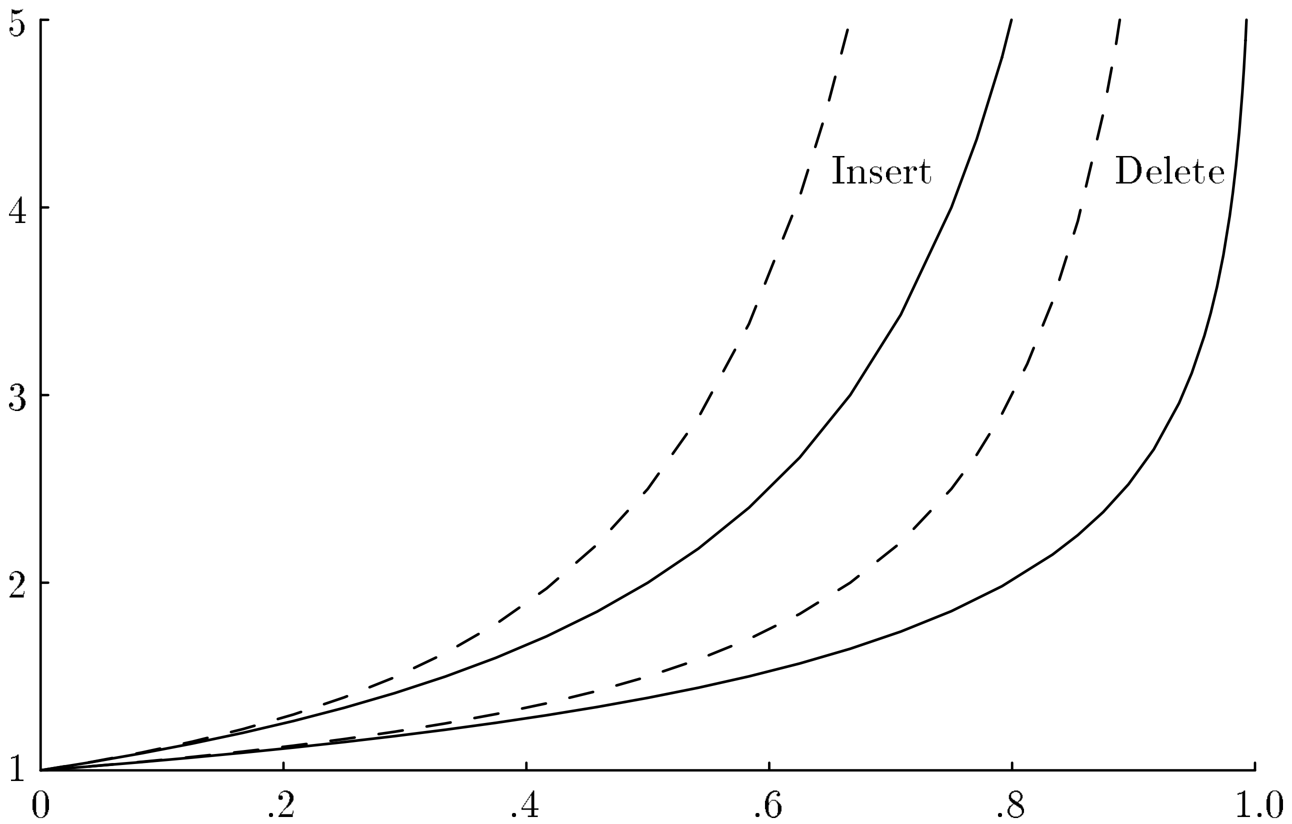
Be sure that all probe sequence constants are relatively prime to M.

Example: Hash table of size $M = 101$

- $\mathbf{h}(k_1) = 30, \mathbf{h}(k_2) = 28, \mathbf{h}(k_3) = 30.$
- $\mathbf{h}_2(k_1) = 2, \mathbf{h}_2(k_2) = 5, \mathbf{h}_2(k_3) = 5.$
- 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 $\alpha = N/M$ where N is the number of records currently in the table.



Deletion

1. Deleting a record must not hinder later searches.
2. We do not want to make positions in the hash table unusable because of deletion.

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.

Unfortunately, tombstones do 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).