Lecture 6: Searching
Search

Given: Distinct keys $k_1, k_2, \ldots, k_n$ and collection $L$ of $n$ records of the form

$$(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)$$

where $I_j$ is the 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 $L$ 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.
Average Cost for Sequential Search

How many comparisons does sequential search do on average?
We must know the probability of occurrence for each possible input.
Must $K$ be in $L$?
Let $k_i = i + 1$ be the number of comparisons when $X = L[i]$.

Let $k_n = n$ be the number of comparisons when $X$ is not in $L$.

Let $p_i$ be the probability that $X = L[i]$.

Let $p_n$ be the probability that $X$ is not in $L[i]$ for any $i$.

$$T(n) = np_n + \sum_{i=0}^{n-1} (i + 1)p_i.$$
Generalizing Average Cost

What happens to the equation if we assume all $p_i$'s are equal (except $p_n$)?

$$T(n) = p_n n + \sum_{i=0}^{n-1} (i + 1) p$$

$$= \frac{n + 1 + p_n (n - 1)}{2}$$

Depending on the value of $p_n$,

$$\frac{n+1}{2} < T(n) < n.$$
Searching Ordered Arrays

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than $K$. Why?

Observation: If we look at $L[5]$ and find that $K$ is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If $K > L[n]$, then we know in one test that $K$ is not in $L$.

• What is wrong here?
Jump Search

What is the right amount to jump?

Algorithm:
- Check every \( k \)'th element (\( L[k], L[2k], \ldots \)).
- If \( K \) is greater, then go on.
- If \( K \) is less, then use linear search on the \( k \) elements.

This is called Jump Search.
Analysis of Jump Search

If \( mk \leq n < (m+1)k \), then the total cost is at most \( m + k - 1 \) 3-way comparisons.

\[
T(n, k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1
\]

What should \( k \) be?

\[
\min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}
\]
Jump Search Analysis (cont)

Take the derivative and solve for $T'(x) = 0$ to find the minimum.

This is a minimum when $k = \sqrt{n}$

What is the worst case cost?

Roughly $2 \sqrt{n}$
Lessons

We want to balance the work done while selecting a sublist with the work done while searching another sublist.

In general, make sub-problems of equal effort.

This is an example of divide and conquer.

What if we extend this to three levels? .... ➔ binary 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: 
\[ \bar{C}_n = 1p_1 + 2p_2 + \ldots + np_n. \]
Examples(1)

(1) All records have equal frequency.

\[
\overline{C}_n = \sum_{i=1}^{n} \frac{i}{n} = \frac{(n+1)}{2}
\]

In the more general case we must consider the probability (labeled \(p_n\)) that the search key does not match any one in the list.

\[
\overline{C}_n = (1-p_n)\frac{n+1}{2} + p_n n \quad \frac{n+1}{2} \leq \overline{C}_n \leq n
\]
(2) Geometric Frequency

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

\[ \overline{C_n} \approx \sum_{i=1}^{n} \left( i / 2^i \right) \approx 2. \]
Zipf Distributions

Applications:

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

$$
\bar{C}_n = \sum_{i=1}^{n} \frac{i}{i H_n} = \frac{n}{H_n} \approx \frac{n}{\log_e n}.
$$

80/20 rule:

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

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

Order by actual historical frequency of access. (Similar to LFU buffer pool replacement strategy.)

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.
Text Compression Example

Application: Text Compression and transmission.

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 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, high percentage of elements in set).

Can use logical bit operators.

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

0011010100010100 & 0101010101010101

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Document processing: Signature files: a list of keywords, each associated with a bit vector indicating which documents contain the keyword.
Hashing

Hash: compute the position, rather than search for the position, of a key.

Hash function: a function used to compute a record’s position

Hash table: the data structure in which records are placed and retrieved on positions calculated by hash functions. A position in the table is called a slot, numbered from 0 to M-1.

In a hashing system, any key value K and some hash function h, h(K) is a slot in the table such that 0 <= h(K) < M, and we have the record of key K stored at position h(K).
Hashing

Key range is usually much larger than slot count (M).

Multiple keys can be mapped to the same slot.

Given a hash function $h$ and two keys $k_1$ and $k_2$, if $h(k_1) = h(k_2)$ then we say that $k_1$ and $k_2$ have a collision at slot $h(k_1)$ (or $h(k_2)$) under hash function $h$. $\Rightarrow$ a collision resolution mechanism is required.
Hash Functions

If we know nothing about keys’ distribution, a hash function should evenly distribute keys across the hash table.

Example hash functions:

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

2. mid-square: square a key and takes middle digits

3. What are good hash functions when key is a character string?
(3) What are good hash functions when key is a character string?

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

Another version:

```c
int sfold(char* key) {
    unsigned int *lkey = (unsigned int *)key;
    int intlength = strlen(key)/4;
    unsigned int sum = 0;
    for(int i=0; i<intlength; i++)
        sum += lkey[i];

    // Now deal with the extra chars at the end
    int extra = strlen(key) - intlength*4;
    char temp[4];
    lkey = (unsigned int *)temp;
    lkey[0] = 0;
    for(int i=0; i<extra; i++)
        temp[i] = key[intlength*4+i];
    sum += lkey[0];

    return sum % M;
}
```
Open Hashing

The collisions are stored outside of the table.

What should be considered to decide the table size?
Closed Hashing

All records are stored directly within the hash table.

Two versions of bucket hashing: home position of k is h(k)