Hashing

Biostatistics 615
Lecture 12
Abstraction
- Group data required by an algorithm into a `struct` (in C) or `list` (in R)
- Hide implementation details

Stacks
- Implemented as arrays
- Implemented as linked lists
Today

- Hashing Algorithms
- Fast way to organize data prior to searching
- Trade savings in computing time for additional memory use
Almost Trivia

- Short detour… Finding primes

- How do we find all prime numbers less than some number?
Eratosthenes Sieve

- List all numbers less than N
  - Ignore 0 and 1

- Find the smallest number in the list
  - Mark this number as prime
  - Remove all its multiples from the list

- Repeat previous step until list is empty
The Sieve in C

```c
void list_primes()
{
    int i, j, a[N];

    for (i = 2; i < N; i++)
        a[i] = 1;
    for (i = 2; i < N; i++)
        if (a[i])
            for (j = i * i; j < N; j += i)
                a[j] = 0;
    for (i = 2; i < N; i++)
        if (a[i]) printf("%4d ", i);
}
```
Notes on Prime Finding

- The algorithm is extremely fast
  - R version takes <1 sec to find all primes <1,000,000

- Performance can be improved by tweaking the inner loop
  - Can you suggest a way?

- Illustrates useful idea: check whether a number is in a list, by using it as an index into an array.
listPrimes <- function(N = 100)
{
    a <- seq(1, N)

    max = trunc(sqrt(N))

    a[seq(4, N, by = 2)] = 0

    for (i in 3:max)
        if (a[i] && (i * i <= N))
            a[seq(i * i, N, by = i * 2)] = 0

    return(a)
}
Idea

- If all items are integers within a short range...
  - ... speed up search operations
  - ... avoid having to sort data

- How?
Even better!

With this strategy…

- Adding an item to the collection takes constant time

- Searching through the collection takes constant time

- Independent of the number of objects in the collection!
Previous Search Strategies

- Place data into an array
  - $O(N)$

- Sort array containing data
  - $O(N \log N)$

- Search for items of interest
  - $\log N$ per search
Using Items as Array Indexes

- Place data into an array
  - $O(N)$

- Sort array containing data

- Search for items of interest
  - $O(1)$
Hashing

- Method for converting arbitrary items into array indexes
  - Items can always serve as array indexes…

- A different approach to searching
  - Not (primarily) based on comparisons
Time – Space Trade Off

- If memory were no issue…
  - Could allocate arbitrarily large array so that each unique item could be a unique index

- If computing time were no issue…
  - Could use linear search to identify matches

- Hashing balances these two extremes
Components of Hashing

- Hash Function
  - Generates table address for individual key

- Collision-Resolution Strategy
  - Deals with keys for which the Hash Function generates identical addresses
Desirable Hash Functions

Poor Hash Function:
Data is clustered

Good Hash Functions:
Data is Evenly Distributed
Hash Function #1

- Assume indexes are in range $[0, M - 1]$
- Items are floating point values between 0 .. 1
- Multiply by M and round
Hash Function #1

- In general, if items take ...
  - Minimum value $min$
  - Maximum value $max$

- Define hash function as

$$(\text{item} - \text{min}) / (\text{max} - \text{min}) \times M$$
Unfortunately ...

- If the items are not randomly distributed within their range...
- Hash function will generate a lot of collisions.
- Better strategies exist...
Hash Function #2

- For integers
- Ensure that table size $M$ is prime
- Define hash function as

\[ \text{item modulus } M \]

Note: The modulus operators are `%` (in C) and `%%` (in R)
Hash Function #2

- For floating point values
- First map item into 0 .. 1 range
  - As before ...
- Multiply result by large integer (say $2^K$) and truncate
- Take modulus $M$ (which should still be prime)
Two Simple Hash Functions

```c
int hash_int(int item, int M)
{
    return item % M;
}

int hash_double(double item, int M)
{
    return (int) ((item - min)/(max - min) * LARGE_NUMBER) % M;
}
```
A Hash Function for Strings

```c
int hash_string(char * s, int M) {
    int hash = 0, mult = 127, i;

    for (i = 0; s[i] != 0; i++)
        hash += (s[i] + hash * mult) % M;

    return result;
}
```

In C, characters can be treated as numbers. A string is an array of characters that terminates with the element zero.
Conflict Resolution: Separate Chaining

- What to do with items where the hash function returns the same value?

- One option is to make each entry in the hash table an array or list …
  - Each entry corresponds to a "chain of items"
Separate Chaining Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A S E R C H I N G X M P L</td>
<td>0 2 0 4 4 4 4 2 2 1 2 4 3 3</td>
</tr>
</tbody>
</table>

Chains:
- 0: E -> A
- 1: G
- 2: X -> N -> I -> S
- 3: L -> P
- 4: M -> H -> C -> R
For example ...

# Create a hash table as a list of vectors
table <- vector(M, mode = "list")

for (i in 1:M) table[[i]] <- c()

# Add an element to the table
h <- hash(item, M)
table[[h]] <- c(item, table[[h]])

# Check if an element is in the table
item %in% table[[hash(item, M)]]
Properties of Separate Chaining

- If the hashing function results in random indexes...

\[ \frac{N}{M} \]  
expected number of entries in each chain

\[ \binom{N}{k} \left( \frac{1}{M} \right)^k \left( 1 - \frac{1}{M} \right)^{N-k} \]  
number of entries follows Binomial distribution  
(well approximated with Poisson distribution)
Interesting Known Properties

- Number of entries is close to average
  \[ e^{-\alpha} \] probability of an empty slot
  \[ \sim 1.25\sqrt{M} \] number of items before first collision
  \[ \text{"the birthday problem"} \]
  \[ M\left(\sum_{i=1}^{M} \frac{1}{i}\right) \] number of items before all slots have one item
  \[ \text{"the coupon collector problem"} \]

- \( \alpha = \frac{N}{M} \) is the load factor…
Notes on Hashing

- Good performance when
  - Searching for elements
  - Inserting elements

- Ineffective when
  - Selecting elements based on rank
  - Sorting elements
Conflict Resolution 2: Linear Probing

- If we can guarantee that $M > N$
  - In this case, $\alpha < 1$

- Whenever there is a collision, search sequentially for the next empty slot
Linear Probing Example

Item Hash1

Table index

Table after inserting element 1
Table after inserting element 2

Table after inserting all elements
Cost Depends on Clustering...

- Consider two tables that are half full
  - In one, items occupy all the odd positions
  - In another, items occupy first M/2 positions

- Where do you expect searches to take longer?
/* Creating a hash table */
Item table[M];
for (i = 0; i < M; i++)
    table[M] = EMPTY;

/* Inserting an item */
h = hash(item, M);
while (table[h] != item && table[h] != EMPTY)
    h = (h + 1) % M;

/* Search successful if table[h] != EMPTY */
/* Otherwise, item could be inserted at table[h] */
### Number of Comparisons

<table>
<thead>
<tr>
<th>load factor ($\alpha$)</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Hit</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
<tr>
<td>Search Miss</td>
<td>2.5</td>
<td>5.0</td>
<td>8.5</td>
<td>50.5</td>
</tr>
</tbody>
</table>

\[
Cost(\text{Hit}) = \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \quad Cost(\text{Miss}) = \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)
\]
Notes on Linear Hashing

- Deleting elements is cumbersome
- Must rehash all other elements in cluster
- Or replace with "DELETED" element
  - Counted as mismatch in searches
  - Counted as empty slot for insert
Conflict Resolution 2: Double Hashing

- Similar to linear hashing
- Guards against clustering by using hash function to generate increment for sequential searches
### Double Hashing Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Hash1</th>
<th>Hash2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Table after inserting element 1</td>
<td>Table after inserting element 2</td>
</tr>
<tr>
<td></td>
<td>Table after inserting all elements</td>
<td>Table index</td>
</tr>
</tbody>
</table>
Double Hashing: C fragments

/* Searching for an item */
h = hash(item, M);
h2 = hash2(item, another_prime) + 1;
while (table[h] != item && table[h] != EMPTY)
    h = (h + h2) % M;

/* Search successful if table[h] != EMPTY */
/* Otherwise, item could be inserted at table[h] */
<table>
<thead>
<tr>
<th>load factor ($\alpha$)</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Hit</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>Search Miss</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>10</td>
</tr>
</tbody>
</table>

Cost(Hit) = \( \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \)  
Cost(Miss) = \( \frac{1}{1-\alpha} \)
Analysis of Double Hashing

- Performance similar to random hashing
  - Unique sequence of keys for each item

- Number of probes for a miss would be...

\[
1 + \frac{N}{M} + \left(\frac{N}{M}\right)^2 + \left(\frac{N}{M}\right)^3 \ldots = \frac{1}{1 - N/M} = \frac{1}{1 - \alpha}
\]
Analysis of Double Hashing

- Number of probes for a hit
  - The same as the cost of originally inserting the item
  - With N items, assume that each one is target with probability 1/N

\[
\frac{1}{N} \left( \frac{1}{1-1/M} + \frac{1}{1-2/M} + \frac{1}{1-3/M} + \ldots \right) = \\
\frac{1}{N} \left( \frac{M}{M-1} + \frac{M}{M-2} + \frac{M}{M-3} + \ldots \right)
\]
Further Notes on Hashing

- To ensure that search requires less than \( t \) comparisons on average
  - \( \alpha < (1 - 1/t) \) with double hashing
  - \( \alpha < (1 - 1/\sqrt{t}) \) with linear probes

- Dynamic hashing
  - Increase table size and rehash elements whenever \( \alpha \) exceed threshold (typically 0.5)
Cost Comparison

Cost of Searches with Double Hashing

Cost of Searches with Linear Probing
Summary

- Hashing
  - Chaining
  - Linear Probing
  - Double Hashing

- Cost of search that is independent of N
  - Fast searches that don't require sorting
  - Not very effective for rank-based selection
Further Reading

- Sedgewick, Chapter 14