Hashtable

December 02, 2019
If we want to store the numbers from 0 to 1,023, we can use an array of 1,024 elements, one for each possible value.

But consider these two cases:
- If we want to store only 100 numbers from a large domain, such as from 0 to 1,000,000, the above approach is wasteful of space.
- If we want to store 100 strings, how do we map a string to an array index?

Solution: Map the keys to array indices with a hash function.

The keys to be hashed can be integers, strings, or complex objects.
Hash Table

- A table of $m$ slots (or buckets)
- A universe $U$ of keys – the size of $U$ is much larger than $m$
- A hash function $h : U \rightarrow \{0, 1, \ldots, m - 1\}$
- $h(x)$ is expected to take $O(1)$ time
- A collision: two keys are hashed to the same slot
Hash Function

- **Division method:** $h(k) = k \mod m$
  - $m$ should not be a power of 2
  - Choose a prime not too close a power of 2

- **Multiplication method**
  - $A = (\sqrt{5} - 1)/2$
  - $h(k) = \lfloor m(kA \mod 1) \rfloor$
Hash Function for Strings

- Strings of fewer than 5 chars can be packed into an int and hashed
- Longer strings: it is important to let all parts of the string contribute to the result

```c
unsigned hash(char *key, unsigned m) {
    unsigned hashVal = 0;
    while (*key != '\0') {
        hashVal += *key;
        key++;
    }
    return hashVal % m;
}
```

- Advantage: Use all the available information, simple to calculate
- Disadvantage: Return same value for words like “bat” and “tab”; limited to values between 0 and 127 * strlen(key) % m
A Good Hash Function for Strings

- It is better to slide the contributions of characters over by multiplications by a prime, such as 31

```c
unsigned hash(char *key, unsigned m) {
    unsigned hashVal = 0;

    while (*key != '\0') {
        hashVal += 31 * hashVal + *key;
        key++;
    }

    return hashVal % m;
}
```

- This distinguishes between “bat” and “tab”
- To hash complex or large objects: hash the concatenated strings, add numbers to the hash, etc
A collision happens when two keys are hashed to the same bucket – having the same hash value.

What to do?

1. Separate chaining: Make the hash table an array of linked lists.
2. Open addressing: If the first spot is full, use another spot in the hash table.
   - Linear probing: Look in the next spot in the array.
   - Quadratic probing: Look in quadratically-determined places in the array.
Separate Chaining

- Each slot is a doubly linked list holding all keys that hash to that slot.
- When collision happens, insert the keys in the linked list.
- This is the simplest way to make a hash table that works decently.
- In some cases separate chaining may be a little slower than quadratic probing, because it causes memory references that hop around in memory – cache misses.
- If the table has \( m \) slots and stores \( n \) keys, the load factor is \( \alpha = \frac{n}{m} \).
- Assume a key is equally likely to hash into any of the \( m \) slots – simple uniform hashing.
- Let \( n_i, i = 0, 1, \ldots, m - 1 \) be the length of the linked list at slot \( i \).
- \( n = n_0 + n_1 + \cdots + n_{m-1} \)
- \( E[n_i] = \alpha \)
- Average-case: unsuccessful search takes \( \Theta(1 + \alpha) \) time.
- Average-case: successful search takes \( \Theta(1 + \alpha) \) time.
Chaining

$U$ (universe of keys)

$K$ (actual keys)

$T$

- $k_1$ → $k_4$
- $k_5$ → $k_2$ → $k_7$
- $k_8$ → $k_6$
- $k_3$
Open Addressing

- Store the keys directly in the table – no linked lists
- The table can fill up – we must keep $n \leq m$
- When there is a collision, probe the table for an empty slot
- Probe sequence: $h(k, 0), h(k, 1), \ldots, h(k, m - 1)$ is a permutation of $0, 1, \ldots, m - 1$
- Linear probing, quadratic probing, double hashing
Linear Probing

- If \( b = h(x) \) is already in use, try \( b + 1 \), then \( b + 2 \), etc., wrapping around to \( b = 0 \) if you reach \( m \): \( b = (b + 1) \mod m \)
- \( m \) distinct probe sequences
- As soon as you find an empty slot, take it
- On lookup, hash the key and check if it matches the one in the hash slot, if not, try the next (wrapping if necessary), etc., until you find the matching one or an empty slot
- Issue: primary clustering, an empty slot preceded by \( i \) occupied slots gets filled with probability \( (i + 1)/m \)
  - Stretches of the array get filled up and probes have to go farther and farther
  - You cannot delete items in a simple way by just removing them, because they have been used as stepping stones in other key’s probing sequences
Quadratic Probing

- \( h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \)
- \( c_1 \) and \( c_2 \) are positive constants
- \( m \) distinct probe sequences
- Issue: secondary clustering, when two keys start at the same initial slot, they probe the same sequence
- Secondary clustering is less problematic than primary clustering
Assume $m$ is prime

Instead of searching one by one through the array from the original slot, jump by bigger and bigger steps $b + 1, b + 4, b + 9, \ldots, b + i^2$

Wrap around to $b = 0$ if necessary

Because of jumping farther and farther, the clustering effect is lessened
Justification for Quadratic Probing

- We need to show that these sequences do not start cycling around some small set of values.
- Suppose they did. Let the table size $m$ be a prime:

$$b + i^2 = b + j^2 \quad \text{mod} \quad m$$

for some $i$ and $j$, different probe numbers

$$i^2 = j^2 \quad \text{mod} \quad m$$

$$i^2 - j^2 = 0 \quad \text{mod} \quad m$$

$$(i - j)(i + j) = 0 \quad \text{mod} \quad m$$

- Then $(i - j)(i + j)$ must be divisible by $m$, a prime.
- If we keep the hash table less than half full, then $i \leq m/2$ and $j \leq m/2$, and $(i - j)(i + j)$ cannot be divisible by $m$.
- Conclusion: as long as the hash table is less than half full, quadratic probing will succeed in $m/2$ tries.
General Guidelines for Hash Table

- Keep hash tables no more than half full
- Rehashing: every time the number of items doubles, double the hash table size to keep under the half-full rule
  - Use the technique of dynamic arrays
  - We can still maintain $O(1)$ lookup
- Use a prime not too close to $2^k - 1$ for the hash table size
Double Hashing

- \( h(k, i) = (h_1(k) + ih_2(k)) \mod m \)
- The value of \( h_2(k) \) must be relatively prime to \( m \) to produce a full-length probe sequence
- Let \( m \) be a power of 2 and choose \( h_2 \) so that it always produces an odd number
- Let \( m \) be a prime and choose \( h_2 \) so that it returns a number less than \( m \)
- \( \Theta(m^2) \) distinct probe sequences
Performance of Open Addressing

- Load factor $\alpha = \frac{n}{m} < 1$
- Unsuccessful search: the expected number of probes is at most $\frac{1}{1 - \alpha} = 1 + \alpha + \alpha^2 + \cdots$
- Successful search: at most $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$
We cannot just delete a key
This creates a hole in the probe sequence
We can mark the slot with a special key to denote that it used to hold a key
Search time will no longer depend on the load factor
Use open addressing when no deletion is needed
Use separate chaining when we need to delete
Rehash when there have been many deletions
Implement Separate Chaining

```c
struct nlist {
    struct nlist *next; // link to next word
    char *name;        // word
    char *defn;        // definition
};
```

- An array of pointers to structs
- Remember you can `typedef` `nlist`
Dynamic Array

- Initialize an array with one element
- Before inserting a new element (at the end, when the array is used as a stack), if the array is full
  - Allocate a new array of twice the length
  - Copy the existing elements to the new array
- Then proceed with insertion
- When the array is used as a hash table, we need to rehash all elements in the old array into the new array
- Great sources of segfaults
The cost of the $i$th insertion is

$$c_i = \begin{cases} 
  i & \text{if } i - 1 \text{ is a power of 2} \\
  1 & \text{otherwise}
\end{cases}$$

<table>
<thead>
<tr>
<th>$i$</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>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_i$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

The total cost of $n$ insertions is

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \leq n + 2n = 3n$$

The average cost of one insertion is 3