“Incremental” Hashing

Reading:
• None in your text.

• Optional reading: ch 12 of File Structures: An Object-Oriented Approach with C++, by Folk et al.

A Problem

Performance degrades if the file becomes heavily loaded, i.e., if \( \frac{\text{actual number of recs}}{\text{num buckets x bucket size}} \) gets large.

To make things better, it may be worthwhile to increase the number of buckets (and reorganize the data).

This general idea is called incremental hashing.

Guess what? There are many ways to do it.

Incremental Hashing

General Approach

As records are inserted, if performance becomes too low, grow the file.

• i.e., “split” one bucket and disperse its records; some stay put and others go to a new bucket.

• This reduces overflow (collisions to full buckets) and hence reduces the # of file accesses during search.

As records are deleted, if space usage becomes too poor, shrink the file.

• i.e., merge two buckets into one.

• This reduces the total # of buckets, and hence reduces waste.

File growth and shrinkage is incremental, i.e.:

• It happens on the fly.
  We do it during insertions and deletions, if needed.

• It happens in small amounts.
  We split one bucket rather than rehashing the whole file.

Possible measures of performance include:

• load factor

• average # of disk accesses per search.
Method I: Linear Hashing

Method

- When performance becomes too poor, split bucket 0. (Yes, this is arbitrary.)
- Split it by doubling the mod factor and re-hashing its contents. E.g.,
  \[ h(k) = k \mod 3 \quad \text{becomes} \quad h(k) = k \mod 6. \]
- Next time, split bucket 1, then 2, etc.
- Keep a counter to remember which buckets have been split.
  Unsplit ones use the old hash function.
  Split ones use the new.

Merging is analogous but opposite.

<table>
<thead>
<tr>
<th># buckets</th>
<th>old</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T : 0 \ldots (T - 1) )</td>
<td>( h(k) = k \mod 3 )</td>
<td>( h(k) = k \mod 6 )</td>
</tr>
<tr>
<td>( T + 1 : 0 \ldots T )</td>
<td>( h(k) = k \mod T )</td>
<td>( h(k) = k \mod 2T )</td>
</tr>
</tbody>
</table>

Guarantee: Every element of bucket 0 will either stay put, or land in the new bucket \( T \).

More generally, if we split bucket \( b \), every record will either stay put, or land in the new bucket \( T + b \).

Let \( k \) be the record’s key.
If it was in bucket \( b \) originally, we know
\[ k \mod T = b. \]
So \( k \) must have been one of these:
\[ h \quad T + b 
2T + b 
3T + b 
4T + b 
5T + b \ldots \]

Questions

Will linear hashing work if we use open addressing to solve collisions?

Why split the “next” bucket? Why not the culprit, \( i.e., \) the one we inserted to when we passed the performance threshold?

Decision: What if the split fails, \( i.e., \) everything happens to stay put? We could split again.

What happens when we’ve split all the original buckets?
Method II: Extendible Hashing

Build a dynamic directory (in memory for speed) that copes with the varying load factor.

- Hash function takes you to a directory entry, rather than directly to a bucket.
- Because buckets are pointed to, needn’t be consecutive in the file. So can add and remove buckets as desired.
- Directory must grow and shrink with number of buckets.
- So # of places to hash to changes. Cope by using only the first so many bits of h(key); change this as necessary to change size of directory.
- If using \( d \) bits, directory size is \( 2^d \).
- So have capacity for \( 2^d \) buckets, but can start with fewer; even just one.

How to “grow” the file

When a bucket overflows:

- Split the one bucket in two.
- Half of the directory entries that pointed to the old bucket will still do so, and half will point to the new bucket.

Eventually, we may reach a point where we can’t split a bucket this way.

- This occurs when only one directory entry points to the bucket we want to split.
- Then we double the directory size, and re-organize.