Hash table collision resolution

Goldsmiths Computing

Motivation

We need (in general) the ability to insert an object whose reduced hash code is the same as that of an object already in the hash table.

Definition

A collision resolution strategy says what to do if a collision is found. Routines for insert, find and delete must all agree on the collision resolution strategy.

Separate chaining

aka: "Open hashing", "Closed addressing": each hash-table slot contains a *list* of values.

closed addressing: the location in the hash table is always the place implied by the hash code

open hashing: the object isn't directly stored in the hash table

function FIND(o,h)
  i ← HASH(o,h)
  return FIND(o,h.table[i])
end function

function INSERT(o,h)
  i ← HASH(o,h)
  h.table[i] ← CONS(o,h.table[i])
end function

function DELETE(o,h)
  i ← HASH(o,h)
  return DELETE(o,h.table[i])
end function

Open addressing

aka: "Closed hashing": if there's a collision, probe for a empty slot somewhere else

open addressing: find a different location in the hash table than that implied by the hash code

closed hashing: always directly store in the hash table

find

```
function FIND(o,h)
    i ← HASH(o,h); k ← 0
    repeat
        j ← PROBE(i,k,h); k ← k + 1
        if h.table[j] = o then
            return true
        end if
    until EMPTY?(h.table[j])
    return false
end function
```

Open addressing

aka: "Closed hashing": if there's a collision, probe for a empty slot somewhere else

open addressing
: find a different location in the hash table than that implied by the hash code

closed hashing
: always directly store in the hash table

insert

```
function INSERT(o,h)
    i ← HASH(o,h); k ← 0
    repeat
        j ← PROBE(i,k,h); k ← k + 1
    until FREE?(h.table[j])
    h.table[j] ← o
end function
```

Open addressing

Linear probing

If there's a collision, probe by looking at the next slot.

```
function PROBE(i,k,h)
    return (i + k) mod SIZE(h.table)
end function
```

- good locality of reference
- simple to implement

but

- similar hash codes lead to secondary collisions

Open addressing

Why FREE?, not EMPTY?, in INSERT?

Delete

function DELETE(o,h)
  i ← HASH(o,h); k ← 0
  repeat
    j ← PROBE(i,k,h); k ← k + 1
  until h.table[j] = o
  h.table[j] ← †
end function
function FREE?(value)
  return EMPTY?(value) ∨ value = †
end function

Open addressing

Quadratic probing

If there's a collision, probe by looking at slots square numbers away.

function PROBE(i,k,h)
  return (i - $(-1)^k \times k^2$) mod SIZE(h.table)
end function

- reduced primary clustering
- preserves some locality of reference
- SIZE(h.table) must be a prime number

Open addressing

Complexity of hash-table operations:

- no collisions: $\Theta(1)$
- everything collides: $\Theta(N)$
- usual case: somewhere in between

Improve the usual case:

- decrease probe length

Robin Hood linear probing

While inserting:

- if you find a value that is less far from its natural space
- steal that space and insert the value you've displaced instead

Extending and rehashing

Too many collisions?

1. make a bigger table
2. re-insert all the current contents
    - different reduction will mean different, fewer collisions

Work

1. Reading
   - CLRS, section 11.4
   - Drozdek, section 10.2
   - Pedro Celis, “Robin Hood Hashing”
2. Exercises

   CLRS 11.2-2, 11.4-1
3. Lab work (week of Monday 26th November)