Hash Tables

Chapter 20

CS 3358
Spring 2015

Jill Seaman

Sections 20.1, 20.2, 20.3, 20.4 (not 20.4.2), 20.5

What are hash tables?

- A Hash Table is used to implement a **set**, providing basic operations in constant time:
  - insert
  - remove (optional)
  - find (membership test)
  - makeEmpty (need not be constant time)
- It uses a function that maps an object in the set (a key) to its location in the table.
- The function is called a **hash function**.

Using a hash function

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers.

This hash function can be used to store and retrieve parts in an array.

Hash(partNum) = partNum % 100

Placing elements in the array

	values
[0]	Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

Hash(partNum) = partNum % 100

to place the element with part number 5502 in the array.

Placing elements in the array

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Next place part number 6702 in the array.

Hash(partNum) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to resolve the collision?

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

One way is by linear probing. This uses the following function

(HashValue + 1) % 100

repeatedly until an empty location is found for part number 6702.

Resolving the collision

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

(HashValue + 1) % 100

Collision resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Part 6702 can be placed at the location with index 4.

Collision resolved

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

Hashing concepts

- **Hash Table**: (usually an array) where objects are stored by according to their key
  - **key**: attribute of an object used for searching/ sorting
  - number of <u>valid</u> keys usually greater than number of slots in the table
  - number of keys <u>in use</u> usually much smaller than table size.
- **Hash function**: maps a key to a Table index
- **Collision**: when two separate keys hash to the same location

Hashing concepts

- **Collision resolution**: method for finding an open spot in the table for a key that has collided with another key already in the table.
- **Load Factor**: the fraction of the hash table that is full
  - may be given as a percentage: 50%
  - may be given as a fraction in the range from 0 to 1, as in: .5

Hash Function

- Goals:
  - computation should be fast
  - should minimize collisions (good distribution)
- Some issues:
  - should depend on ALL of the key (not just the last 2 digits or first 3 characters, which may not themselves be well distributed)

Hash Function

- Final step of hash function is usually:

  temp % size

  - temp is some intermediate result
  - size is the hash table size
  - ensures the value is a valid location in the table (0..size-1)
- Picking a value for size:
  - Bad choices:
    - a power of 2: then the result is only the lowest order bits of temp (not based on whole key)
    - a power of 10: result is only lowest order digits of decimal number
  - Good choices: prime numbers

Hash Function: string keys

- If the key is not a number, hash function must transform it to a number, to % by the size
- Method 1: Add up ascii values

```
int hash (string key, int tableSize) {
  int hashVal = 0;
  for (int i=0; i<key.length(); i++)
    hashVal = hashVal + key[i]; //implicit conversion
  return hashVal % tableSize;
}
```

  - different permutations of same chars have same hash value ("cat" and "act" have same value)
  - large tableSize and short key length do not distribute well:

If table size is 10,007 and keys are 8 characters long:
Since ascii values are between 0 and 127, There are 128^8 = 7.2×10^{16} different keys.
Hash produces values between 0 and 127*8 = 1016 all falling in first 1/10th of the table

Hash Function: string keys

- Method 2: Multiply each char by a power of 128

hash = (k[0]*128^3 + k[1]*128^2 + k[2]*128^1 + k[3]*128^0) %tableSize

This is equivalent to: hash = (((k[0]*128 + k[1])*128 + k[2])*128 + k[3]) %tableSize

  - now "cat" and "act" map to different values (most likely)
  - but now we get really big numbers (overflow)
  - we can take mod of intermediate results to reduce overflow:

```
int hash (string key, int tableSize) {
  int hashVal = 0;
  for (int i=0; i<key.length(); i++)
    hashVal = (hashVal * 128 + key[i]) % tableSize;
  return hashVal;
}
```

  - but mod is expensive . . .

Hash Function: string keys

  - could just allow overflow, take mod at the end
  - but repeated multiplying by 128 tends to shift early chars out to the left (potentially leading to more collisions)
- Method 3: Multiply each char by a power of 37

hash = (((k[0]*37 + k[1])*37 + k[2])*37 + k[3])*37

```
int hash (string key, int tableSize) {
  int hashVal = 0;
  for (int i=0; i<key.length(); i++)
    hashVal = hashVal * 37 + key[i];
  return hashVal % tableSize;
}
```

  - compromise. 37 is prime. Has good distribution.
  - "au" and "bP" map to the same value, but collisions are less common than in method 1.

Collision Resolution: 1. Linear Probing

- Insert: When there is a collision, search sequentially for the next available slot
- Find: if the key is not at the hashed location, keep searching sequentially for it.
  - if it reaches an empty slot, the key is not found
- Problem: if the the table is somewhat full, it may take a long time to find the open slot.
  - May not be O(1) any more
- Problem: Removing an element in the middle of a chain breaks the algorithm.

Linear Probing: Example

- Insert: 89, 18, 49, 58, 69, hash(k) = k mod 10

Probing function (attempt i): $h_i(K) = (hash(K) + \mathbf{i})$ % tablesize

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

49 is in 0 because 9 was full

58 is in 1 because 8, 9, 0 were full

69 is in 1 because 9, 0 were full

Linear Probing: delete problem

	values
[0]	Empty
[1]	4501
[2]	5502
[3]	7803
[4]	6702
.	.
.	.
.	.
[97]	Empty
[98]	2298
[99]	3699

Part 6702 was placed at the location with index 4, after colliding with 5502

Now remove 7803.

Now find 6702 (hash(6702)=2):
not at values[2]
values[3] is empty, so not found

Linear Probing: Lazy deletion

- Don't remove the deleted object, just mark as deleted
- During find, marked deletions don't stop the searching
- During insert, the spot may be reused
- If there are a lot of deletions, searching may still take a long time even if the table is mostly empty.

Linear Probing:
Primary Clustering

- Cluster: a large, sequential block of occupied slots in the table
- Any key that hashes into the cluster requires excessive attempts to resolve the collision
- If it's during an insert operation, the cluster gets bigger.
- If two clusters are separated by one slot, a single insertion will drastically degrade the future performance
- Primary clustering is a problem at high load factors (90%), not at 50% or less.

Collision Resolution:
2. Quadratic Probing

- An attempt to eliminate primary clustering
- If the hash function returns H, and H is occupied, try H+1, then H+4, then H+9, ...
  - for each attempt i, try $H+i^2$ next.

Probing function (attempt i): $h_i(K) = (hash(K) + i^2)$ % tablesize

- Is it guaranteed to find an empty slot if there is one (like linear probing)?
  - Yes IF: the table size is prime and the load is <= 50%

Quadratic Probing:
Example

- Insert: 89, 18, 49, 58, 69, hash(k) = k mod 10

Probing function (attempt i): $h_i(K) = (hash(K) + \mathbf{i^2})$ % tablesize

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

49 is in 0 because 9 was full

58 is in 2 because 8, 8+1=9 were full, (8+4)%10=2 wasn't

69 is in 3 because 9, (9+1)%10=0 were full, (9+4)%10=3 wasn't

Note: smaller clusters

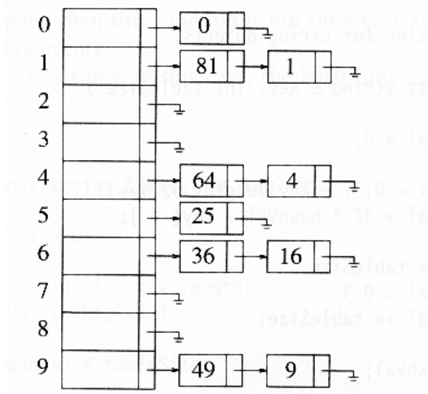
Quadratic probing:
expansion of table

- Since the table should be less than 50% full:
- Can the table be expanded if the load factor gets more than 50%?
- Yes.
  - Find the next prime number greater than 2*tableSize, resize to that.
  - Don't just copy all the elements (new tablesize => new hash function)
  - Scan old table for non-empties, and use insert function to add them to new table using new size.
- This is called **rehashing.**

Collision Resolution:
3. Separate chaining

- Use an array of linked lists for the hash table
- Each linked list contains all objects that hashed to that location
  - no collisions

Hash function is still:
h(K) = k % 10



Separate Chaining

- To insert a an object:
  - compute hash(k)
  - insert at front of list at that location (if empty, make first node)
- To find an object:
  - compute hash(k)
  - search the linked list there for the key of the object
- To delete an object:
  - compute hash(k)
  - search the linked list there for the key of the object
  - if found, remove it

Separate Chaining

- The load can be 1 or more
  - more than 1 node at each location, still O(1) inserts and finds
  - smaller loads do not improve performance
  - moderately larger loads do not hurt performance
- Disadvantages
  - Memory allocation could be expensive
  - too many nodes at one position can slow operations
- Advantages:
  - deletion is easy
  - don’t have to resize/rehash