Hashing II:

The leftovers
Hash functions

• Choice of hash function can be important factor in reducing the likelihood of collisions

• Division hashing: key % CAPACITY
  – Certain table sizes are more conducive to collision avoidance with this method
  – A 1970 study suggests that a good size is a prime number of the form 4k+3
  – For example, 811 = 202 * 4 + 3)
Other hash functions

- **Mid-square hash**
  - multiply key by itself
  - use some middle digits of the result as hash value

- **Multiplicative hash**
  - multiply key by a floating-point constant that is less than one
  - use first few digits of fractional part of result as hash value
Insertion and linear probing

• During insertion process, collision may occur

• In case of collision, the insertion function moves forward through the array until a vacant spot is found; the process is known as linear probing
The perils of probing

- When many keys hash to the same index, elements start to group in clumps near that index; as the table grows, these clumps get larger.
- As the table approaches capacity, these clumps tend to merge into gigantic clusters; hence this process is known as clustering.
- Performance of insertion and search functions degrades when clustering occurs.
Double hashing

- A common technique used to reduce clustering is double hashing
- In double hashing, a second hash function is used to determine where to seek the next vacancy in the array when a collision occurs
- Rather than using linear probing, double hashing ensures that a few entries are skipped after each collision
Double hashing

• Step 1: hash key and check for collision
• Step 2: if collision occurs, run key through second hash function; check index result spaces beyond original
• Example:
  – 1st hash produces 206 - space at this index is taken, so run 2nd hash
  – 2nd hash produces 9, so check space at index 215 - if not vacant, go to 224, then 233, etc.
Considerations for second hash function

• Value added to index must not exceed valid range of array (0 .. CAPACITY-1)

• Can stay within range by using the following formula to determine next index (where hash2 is the second hash function):
  
  \[
  \text{index} = (\text{index} + \text{hash2(key)}) \mod \text{CAPACITY}
  \]
Considerations for second hash function

• Every array position must be examined - with double hashing, spots could be skipped, returning to start position before every available location has been probed

• To avoid this, make sure CAPACITY-1 is relatively prime with respect to value returned by hash2 (in other words, 2nd hash value and last array index should have no common factors)
Example values for double hashing

- Both CAPACITY and CAPACITY-2 should be primes -- e.g. 809 and 811
- First hash function:
  return (key % CAPACITY);
- Second hash function:
  return (1 + (key % (CAPACITY - 2)));
Modifications to dictionary class for double hashing

• Add a hash2 function as private member
• Change next_index to return
  \[(1 + \text{hash2}(\text{key}) \mod \text{CAPACITY})\]
Chained hashing

• Open-address hashing uses static arrays in which each element contains one entry; when the array is full, can’t add more entries
• Could use dynamic arrays, but would require resizing to new prime number and rehashing entire table
• Chained hashing is a more workable alternative
Chained hashing

• Chained hashing uses this approach:
  – each array element is a list which can hold several entries
  – all records that hash to a particular index are placed in the list at that index
  – a chained hash table can hold many more records than a simple hash table
Time analysis of hashing

• In the worst case, every key gets hashed to the same index -- this makes insertion, deletion and searching linear operations (O(N))

• Best case is the same as the linear search algorithm (O(1)), for the same reason

• Neither of these cases is particularly likely, however
Time analysis of hashing

• Average case is relatively complex, especially if deletions are allowed

• Three different formulas have been developed for the average number of elements that must be examined for a successful search -- each corresponds to a different version of hashing (open-address with linear probing, open-address with double hashing, and chained hashing)
Time analysis of hashing

- Each formula depends on the number of elements in the table
- the greater the number of items, the more collisions
- the more collisions, the longer the average search time
Time analysis of hashing

• A load factor (\( \alpha \)) is used in each formula --
  \( \alpha \) is the ratio of the number of occupied
  table locations to the size of the table:

  \[
  \alpha = \text{used} / \text{CAPACITY}
  \]
Time analysis of hashing

• For open-address hashing with linear probing, given the following conditions:
  – hash table is not full
  – no deletions

• Average number of table elements examined for a successful search is:
  \[ \frac{1}{2} \times (1 + \frac{1}{1 - \alpha}) \]
Time analysis of hashing

• Example of open-address hashing with linear probing:
  used = 525
  CAPACITY = 713
  \[ \alpha = \frac{525}{713}, \sim .75 \]

• therefore, average number of searches is:
  \[ \frac{1}{2} \times (1 + \frac{1}{1 - .75}) = 2.5 \]

• meaning about 3 table elements must be examined to complete a successful search
Time analysis of hashing

• Average search time for open addressing with double hashing, given:
  – table is not full
  – no deletions

• Formula:
  \[-\ln (1 - \alpha) / \alpha\]

• For same values as previous example, result is slightly less than 2
Time analysis of hashing

• For chained hashing, conditions for average search time are different:
  – each element of a table is the head pointer of a linked list
  – each list may have several items
  – $\alpha$ may be greater than one
  – formula remains valid even with deletions:
    \[ 1 + \alpha / 2 \]