CSE 326 Lecture 13: Much ado about Hashing

- Today's munchies to munch on:
  - Review of Hashing
  - Collision Resolution by:
    - Separate Chaining
    - Open Addressing
      - Linear/Quadratic Probing
      - Double Hashing
    - Rehashing
    - Extendible Hashing
- Covered in Chapter 5 in the text

Review of Hashing: Integer Keys

- **Idea**: Store data record with its ***key*** in array slot:
  A[Hash(*key*)] where Hash is a hashing function.
- Integer Keys:
  - **Hash(*key*) = *key* mod *TableSize***
  - *TableSize* is size of the array (preferably a prime number)

Review of Hashing: String Keys

- **Idea**: Store data record with its ***key*** in array slot:
  A[Hash(*key*)] where Hash is a hashing function.
- String Keys: Treat characters as digits (e.g. use ASCII value)
  - **Hash(*key*) = StringInt(*key*) mod *TableSize***
  - Examples:
    - StringInt("abc") = $1*27^2 + 2*27^1 + 3 = 786$
    - StringInt("bca") = $2*27^2 + 3*27^1 + 1 = 1540$
    - StringInt("cab") = $3*27^2 + 1*27^1 + 2 = 2216$

Collisions

- What if two different keys hash to the same value?
  - E.g. *TableSize* = 17.
  - Keys 18 and 35 hash to same value:
    18 mod 17 = **1** and 35 mod 17 = **1**
- Cannot store both data records in the same slot in array!
- This is called a **"collision"**
  - How can we resolve collisions during hashing?

Collision Resolution

- Two different methods:
  - **Separate Chaining:** Use data structure (such as a *linked list*) to store multiple items that hash to the same slot
  - **Open addressing (or probing):** search for other slots using a second function and store item in first empty slot that is found

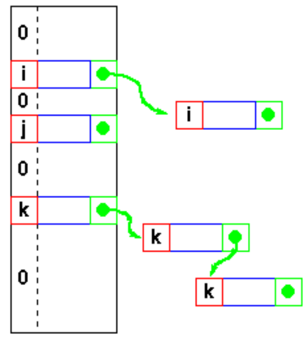
Collision Resolution

- Two different methods:
  - **Separate Chaining:** Use data structure (such as a *linked list*) to store multiple items that hash to the same slot
  - **Open addressing (or probing):** search for other slots using a second function and store item in first empty slot that is found

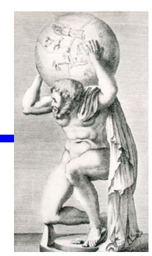
Chaining and probing???
Get me outta here!!

Separate Chaining

- Each hash table cell holds a pointer to a linked list of records with same hash value (i, j, k in figure)
- Collision: Insert item into linked list
- To Find an item: compute hash value, then do Find on linked list
- Can use List ADT for Find/Insert/Delete in linked list
- Can also use BSTs: O(log N) time instead of O(N). But lists are usually small – not worth the overhead of BSTs



Load Factor of a Hash Table



- Let N = number of items to be stored
- **Load factor λ = N/*TableSize***
- Suppose *TableSize* = 2 and number of items N = 10
  - ➪ $\lambda = 5$
- Suppose *TableSize* = 10 and number of items N = 2
  - ➪ $\lambda = 0.2$
- Average length of chained list = λ
- Average time for accessing an item = $O(1) + O(\lambda)$
  - ➪ Want λ to be close to 1 (i.e. *TableSize* $\approx$ N)
  - ➪ But chaining continues to work for $\lambda > 1$

Collision Resolution by Open Addressing

- Linked lists can take up a lot of space…
- Open addressing (or probing): When collision occurs, try alternative cells in the array until an empty cell is found
- Given an item X, try cells $h_0(X)$, $h_1(X)$, $h_2(X)$, …, $h_i(X)$
- **$h_i(X)$ = (Hash(X) + F(i)) mod *TableSize***
  - Define F(0) = 0
- F is the collision resolution function. **Three possibilities**:
  - Linear: **F(i) = i**
  - Quadratic: **$F(i) = i^2$**
  - Double Hashing: **$F(i) = i \cdot Hash_2(X)$**

Open Addressing I: **Linear Probing**

- **Main Idea**: When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $h_i(X)$ = (Hash(X) + i) mod *TableSize* (i = 0, 1, 2, …)
  - Compute hash value and increment until free cell is found
- **In-Class Example**: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 using:
  (a) separate chaining
  (b) linear probing

Load Factor Analysis of Linear Probing

- Recall: **Load factor λ = N/*TableSize***
- Fraction of empty cells = $1 - \lambda$
- Number of such cells we expect to probe = $1/(1- \lambda)$
- Can show that expected number of probes for:
  - Successful searches = $O(1+1/(1- \lambda))$
  - Insertions and unsuccessful searches = $O(1+1/(1- \lambda)^2)$
- Keep $\lambda \leq 0.5$ to keep number of probes small (between 1 and 5). (E.g. What happens when $\lambda = 0.99$)

Drawbacks of Linear Probing

- Works until array is full, but as number of items N approaches *TableSize* ($\lambda \approx 1$), access time approaches O(N)
- Very prone to cluster formation (as in our example)
  - If key hashes into a cluster, finding free cell involves going through the entire cluster
  - Inserting this key at the end of cluster *causes the cluster to grow:* future Inserts will be even more time consuming!
  - This type of clustering is called *Primary Clustering*
- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of distributing keys uniformly

Open Addressing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot – Increment by i^2 instead of i
- **$h_i(X) = (Hash(X) + i^2) \bmod$ *TableSize*** ($i = 0, 1, 2, \ldots$)
  - ➪ No primary clustering but secondary clustering possible
- Example 1: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10
- Example 2: Insert {1, 2, 5, 10, 17} with *TableSize* = 16
  - ➪ Note: 25 mod 16 = 9, 36 mod 16 = 4, 49 mod 16 = 1, etc.
- Theorem: If *TableSize* is prime and $\lambda < 0.5$, quadratic probing will always find an empty slot

Open Addressing III: Double Hashing

- Idea: Spread out the search for an empty slot by using a second hash function
  - ➪ No primary or secondary clustering
- **$h_i(X) = (Hash(X) + i \cdot Hash_2(X)) \bmod$ *TableSize***
  for $i = 0, 1, 2, \ldots$
- **E.g. $Hash_2(X) = R - (X \bmod R)$**
  - ➪ R is a prime smaller than *TableSize*
- Try this example: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 and R = 7
- No clustering but slower than quadratic probing due to $Hash_2$

The need to be lazy…

- Need to use *lazy deletion* if we use probing (why?)
  - Think about how Find(X) would work…
- Mark array slots as "Active/Not Active"
- If table gets too full ($\lambda \approx 1$) or if many deletions have occurred:
  - Running time for Find etc. gets too long, and
  - Inserts may fail!
  - What do we do?

Rehashing

- **Rehashing** – Allocate a larger hash table (of size 2**TableSize*) whenever λ exceeds a particular value
- **How does it work?**
  - Cannot just copy data from old table: Bigger table has a new hash function
  - Go through old hash table, ignoring items marked deleted
  - Recompute hash value for each non-deleted key and put the item in new position in new table
- Running time = O(N) but happens very infrequently

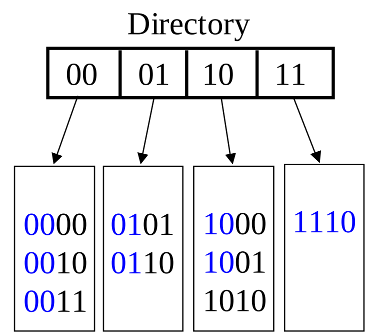
Extendible Hashing

- What if we have large amounts of data that can only be stored on disks and we want to find data in 1-2 disk accesses
- Could use B-trees but deciding which of many branches to go to takes time
- Extendible Hashing: Store item according to its bit pattern
  - **Hash(X) = first d_L bits of X**
  - Each leaf contains ≤ M data items with **d_L identical leading bits**
  - Root contains pointers to sorted data items in the leaves

Extendible Hashing: The details

- Extendible Hashing: Store data according to bit patterns
  - Root is known as the directory
  - M is the size of a disk block

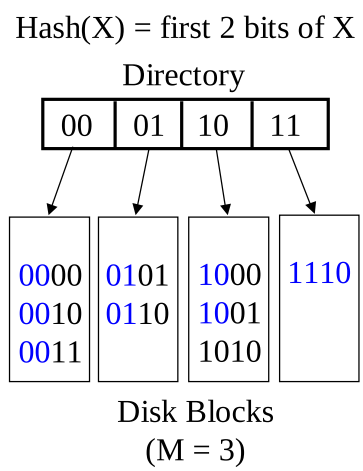
Hash(X) = first 2 bits of X



Disk Blocks
(M = 3)

Extendible Hashing: More details

- Extendible Hashing:
  - **Insert**: If leaf is full, split leaf and increase directory bits by one (e.g. 000, 001, 010, etc.)
  - To avoid collisions and too much splitting, would like bits to be nearly random
    - Hash keys to long integers and then look at leading bits



Applications of Hashing

- **In Compilers**: Used to keep track declared variables in source code – this hash table is known as the "Symbol Table."
- **In storing information associated with strings**
  - Example: Counting word frequencies in a text! (as in HW 3)
- **In Game playing programs**: Store the move for each position by hashing that position into a hash table
  - Called the Transposition table
- **In on-line spell checkers like this ome**
  - Entire dictionary stored in a hash table
  - Each word in text hashed – if not found, word is misspelled.

Next Class:
All sorts of Sorts

To Do:

Finish reading Chapter 5

Assignment # 3 (due Thu Feb 13)

Midterm on Wed Feb 12!