CSE 373 Lecture 13: Hashing

✦ Today’s Topics:
  ➤ Collision Resolution
  ➤ Separate Chaining
  ➤ Open Addressing
    ● Linear/Quadratic Probing
    ● Double Hashing
  ➤ Rehashing
  ➤ Extendible Hashing
✦ Covered in Chapter 5 in the text

Review of Hashing

✦ Idea: Store data record in array slot A[i] where i = Hash(key)
✦ If keys are integers, we can use the hash function:
  ➤ Hash(key) = key mod TableSize
  ➤ TableSize is size of the array (preferably a prime number)
✦ If keys are strings (in the form char *key), get integers by treating characters as digits in base 27 (using “a” = 1, “b” = 2, “c” = 3, “d” = 4 etc.)
  ➤ Hash(key) = StringInt(key) mod TableSize
  ➤ 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 and their Resolution

✦ A collision occurs when two different keys hash to the same value
  ➤ E.g. For TableSize = 17, keys 18 and 35 hash to the same value
  ➤ 18 mod 17 = 1 and 35 mod 17 = 1
✦ Cannot store both data records in the same slot in array?
✦ Two different methods for collision resolution:
  ➤ 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 by Separate Chaining

✦ Each hash table cell holds pointer to 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
Separate Chaining: In-Class Example

- Insert 10 random keys between 0 and 100 into a hash table with TableSize = 10

Load Factor of a Hash Table

- Let N = number of items to be stored
- Load factor \( \lambda = \frac{N}{TableSize} \)
- What is \( \lambda \) for our example?
- Suppose TableSize = 2 and number of items N = 10
  \( \Rightarrow \lambda = 5 \)
- Suppose TableSize = 10 and number of items N = 2
  \( \Rightarrow \lambda = 0.2 \)
- Average length of chained list = \( \lambda \)
- Average time for accessing an item = \( O(1) + O(\lambda) \)
  \( \Rightarrow \) Want \( \lambda \) to be close to 1 (i.e. TableSize = N)
  \( \Rightarrow \) 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 \( h_0(X), h_1(X), h_2(X), \ldots, h_l(X) \)
  - \( h_i(X) = (Hash(X) + F(i)) \mod TableSize \)
  - 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
  - \( h_i(X) = (Hash(X) + i) \mod TableSize \) \( (i = 0, 1, 2, \ldots) \)
  - Compute hash value and increment it until a free cell is found
- Example: Insert \{18, 19, 20, 29, 30, 31\} into empty hash table with TableSize = 10
Load Factor Analysis of Linear Probing

- Recall: Load factor $\lambda = \frac{N}{\text{TableSize}}$
- Fraction of empty cells = $1 - \lambda$
- Number of such cells we expect to probe = $\frac{1}{1 - \lambda}$
- Can show that expected number of probes for:
  - Successful searches = $O(1 + \frac{1}{1 - \lambda})$
  - Insertions and unsuccessful searches = $O(1 + \frac{1}{1 - \lambda^2})$
- 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 $\text{TableSize}$ ($\lambda \approx 1$), access time approaches $O(N)$
- Very prone to cluster formation (as in our example)
  - If a key hashes into a cluster, finding a 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 $i$
- $h_i(X) = (\text{Hash}(X) + i^2) \mod \text{TableSize}$ ($i = 0, 1, 2, \ldots$)
  - No primary clustering but secondary clustering possible
- Example 1: Insert {18, 19, 20, 29, 30, 31} into empty hash table with $\text{TableSize} = 10$
- Example 2: Insert {1, 2, 5, 10, 17} with $\text{TableSize} = 16$
  - Note: 25 mod 16 = 9, 36 mod 16 = 4, 49 mod 16 = 1, etc.
- Theorem: If $\text{TableSize}$ 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
- $h_i(X) = (\text{Hash}(X) + i \cdot \text{Hash}_2(X)) \mod \text{TableSize}$
  - for $i = 0, 1, 2, \ldots$
- E.g., $\text{Hash}_2(X) = R - (X \mod R)$
  - $R$ is a prime smaller than $\text{TableSize}$
- Try this example: Insert {18, 19, 20, 29, 31} into empty hash table with $\text{TableSize} = 10$ and $R = 7$
- No clustering but slower than quadratic probing due to $\text{Hash}_2$
Rehashing

✦ Need to use lazy deletion if we use probing (why?)
  ➤ Need to mark array slots as deleted after Delete
✦ If table gets too full (λ ≈ 1) or if many deletions have occurred, running time gets too long and Inserts may fail
✦ Solution: Rehashing – Build a bigger hash table (of size 2*TableSize) when λ exceeds a particular value
  ➤ 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 is O(N) but happens very infrequently

Extendible Hashing

✦ A method of hashing used when large amounts of data are stored on disks → can find data in 2 disk accesses
✦ Could use B-trees but deciding which of many children contains the data takes time
✦ Extendible Hashing: Store data according to bit patterns
  ➤ Root contains pointers to sorted data bit patterns stored in leaves
  ➤ Leaves contain ≤ M data bit patterns with dL identical leading bits
  ➤ Root is known as the directory; M is the size of a disk block
  ➤ Requires bits to be nearly random, so hash keys to long integers
✦ E.g.: Leaves store bit patterns with 2 identical leading bits
  ➤ See text (page 169)

Wednesday’s Class will be a Lab Session for help with the programming assignment (no lecture)
Where: Communications Bldg. B-027 and B-022
When: 11:30am-12:30pm
Charles and Jiwon will be in the lab to answer questions
To Do:
Finish reading Chapter 5
Programming Assignment #1 (due April 27)