VE281

Data Structures and Algorithms

Hash Table Size, Rehashing, and Applications of Hashing

Learning Objectives:

- Know how to determine hash table size
- Know why rehashing is needed and how to rehash
- Know amortized analysis
- Know a few typical applications of hashing

Outline

- Hash Table Size and Rehashing
- Applications of Hashing

Determine Hash Table Size

- First, given **performance** requirements, determine the maximum permissible **load factor**.

- Example: we want to design a hash table based on **linear probing** so that on average
  - An **unsuccessful** search requires no more than 13 compares.
  - A **successful** search requires no more than 10 compares.

$$U(L) = \frac{1}{2}\left[1 + \left(\frac{1}{1-L}\right)^2\right] \le 13 \quad \Rightarrow L \le \frac{4}{5}$$

$$S(L) = \frac{1}{2}\left[1 + \frac{1}{1-L}\right] \le 10 \quad \Rightarrow L \le \frac{18}{19}$$

$$\boxed{L \le \frac{4}{5}}$$

Determine Hash Table Size

- For a fixed table size, estimate maximum number of items that will be inserted.

- Example: no more than 1000 items.
  - For load factor $L = \frac{|S|}{n} \leq \frac{4}{5}$, table size
$$n \geq \frac{5}{4} \cdot 1000 = 1250$$
  - Pick n as a **prime** number. For example, $n = 1259$.

However, sometimes there is no limit on the number of items to be inserted.

Rehashing

Motivation

- With more items inserted, the load factor increases. At some point, it will exceed the threshold (4/5 in the previous example) determined by the performance requirement.
- For the separate chaining scheme, the hash table becomes inefficient when load factor L is too high.
  - If the size of the hash table is fixed, search performance deteriorates with more items inserted.
- Even worse, for the open addressing scheme, when the hash table becomes full, we **cannot** insert a new item.

Rehashing

- To solve these problems, we need to **rehash**:
  - Create a **larger** table, scan the current table, and then insert items into new table using the new hash function.
  - **Note**: The order is from the beginning to the end of the current table. Not original insertion order.

- We can approximately double the size of the current table.

- Observation: The single operation of rehashing is time-consuming. However, it does not occur frequently.
  - How should we justify the time complexity of rehashing?

Amortized Analysis

- **Amortized analysis**: A method of analyzing algorithms that considers the entire sequence of operations of the program.
  - The idea is that while certain operations may be costly, they don't occur frequently; the less costly operations are much more than the costly ones in the long run.
  - Therefore, the cost of those expensive operations is **averaged** over a sequence of operations.
  - In contrast, our previous complexity analysis only considers a single operation, e.g., insert, find, etc.

Amortized Analysis of Rehashing

- Suppose the threshold of the load factor is 0.5. We will double the table size after reaching the threshold.
- Suppose we start from an empty hash table of size $2M$.

- Assume $O(1)$ operation to insert up to M items.
  - Total cost of inserting the first M items: $O(M)$
- For the $(M+1)$-th item, create a new hash table of size $4M$.
  - Cost: $O(1)$
- Rehash all M items into the new table. Cost: $O(M)$
- Insert new item. Cost: $O(1)$

Total cost for inserting $M+1$ items is $2O(M) + 2O(1) = O(M)$.

Amortized Analysis of Rehashing

Total cost for inserting $M + 1$ items is $O(M)$.

- The average cost to insert $M + 1$ items is O(1).
  - Rehashing cost is **amortized** over individual inserts.

Outline

- Hash Table Size and Rehashing
- Applications of Hashing

Application: De-Duplication

- Given: a stream of objects
  - Linear scan through a huge file
  - Or, objects arriving in real time
- Goal: remove duplicates (i.e., keep track of unique objects)
  - E.g., report unique visitors to website
  - Or, avoid duplicates in search result
- Solution: when new object x arrives,
  - Look x in hash table H
  - If not found, insert x into H

Application: 2-SUM Problem

- Given: an unsorted array A of n integers. Target sum t.
- Goal: determine whether or not there are two numbers x and y in A with

$$x + y = t$$

1. Naïve solution: exhaustive search of pairs of number
   - Time: $\Theta(n^2)$
2. Better solution: 1) Sort A; 2) For each x in A, look for $t - x$ in A via binary search.
   - Time: $\Theta(n \log n)$
3. Best: 1) Insert elements of A into hash table H; 2) For each x in A, search for $t - x$.
   - Time: $\Theta(n)$

Further Immediate Application

- Spellchecker

- Database

Hash Table

Summary

- Choice of the hash function.
- Collision resolution scheme.
- Hash table size and rehashing.

- Time complexity of hash table versus sorted array
  - insert(): $O(1)$ versus $O(n)$
  - find(): $O(1)$ versus $O(\log n)$

- When **NOT** to use hash?
  - **Rank search**: return the k-th largest item.
  - **Sort**: return the values in order.