COMP 480/580 — Probabilistic Algorithms and Data Structures Sep 10, 2024

Lecture 4: Hashing, Chaining, and Probing Analysis

Lecturer: Anshumali Shrivastava

Scribe By:
Abdullah Jahangir mj48
Ian Rundle irr2
Jonathan Mak jem12
Byron Hart blh7

1 Chaining

1.1 Definition

Chaining is a technique used to handle collisions in hashmaps. Because there is the potential that two different keys are hashed to the same index, we can use chaining to resolve this dispute by organizing the collided keys into a linked list. In a well designed hashmap that implements hashmaps, we can make insertion and deletion O(1) time complexity.

1.2 Collision probability

Given that our hashmaps are inherently limited by size and space concerns, we have to calculate the collision probability to evaluate our hash function's performance and efficiency. We can calculate the collision property of our hash functions through the following formula: $P(h(x) = h(y)) \leq \frac{1}{N}$ where h is our hash function and n is our number of possible outputs from our hash function.

Example: Given a hash function $h(x) = x \mod 6$, we know that our hash function has a total of 6 outputs (buckets). By using the expression $P(h(x) = h(y)) \leq \frac{1}{N}$, we know that the probability of two random inputs x and y colliding should be no more than 1/6. While we could pre select x and y to guarantee that they don't collide, given two random, x and y, we know that at most, they will collide 1/6 of the time, with an even distribution in probability of colliding.

1.3 Comparing Hash functions

While in theory, we would like to use perfect hash functions every time, in the real world, we need to consider other limiting factors like memory usage. With real world hash functions, there is a trade off between closeness to perfection in building the hash table and amount resources used to generate said hashtable. In general, cheap hash functions will have better memory usage efficiency compared to perfect hash functions, with a wide spectrum of functions in between.

1.4 Chain Size

Since chaining is designed to improve the speed of searches, we naturally want individual chains to be short. Chain length starts to become problematic when the length is of O(log n). A chain becoming this long tends to be rare. Pr(length of any chain $>$ O(log n)) $<$ 0.0001 serves as a good metric for a hash table being robust enough in preventing over-long chains. We can generalize a function for the probability that a bucket gets a certain amount of items, like so: Pr(Bucket “i” gets $>$ K elements) $= \binom{n}{k} \times \frac{1}{nk}$

2 Universal Hash Functions

2.1 Definition

A universal hash function is a type of hash function that seeks to minimize the number of collisions between k inputs. More formally, we call a hash function k universal if for any number of k distinct inputs, the probability that these inputs all map to the same hash value is bounded by a small fraction of the hash space.

2.2 K Universal Hash Functions

We define a K universal hash function if it has the following probability of k distinct inputs hashing to the same value: $P(h(x_1) = h(x_2) = \ldots = h(x_k)) \leq \frac{1}{N^{k-1}}$. This formula can be generalized for any number of k distinct inputs. For example, a 2-universal hash function would have the following characteristic $P(h(x) = h(y)) \leq \frac{1}{N}$ and the probability that 3 inputs in a 3-universal hash function would hash to the same value would be $P(h(x) = h(y) = h(z)) \leq \frac{1}{N^2}$

Example: $h(x) = ax^2 + bx + c \mod N$ is an example of 3-universal Hash Function. This function has 3 degrees of freedom due to its 3 terms. This means that the values for a,b,c can all be chosen at random and the odds of that 3 distinct values all hash to the same value is $leq\frac{1}{N^2}$.

3 Linear Probing

3.1 Definition

Linear probing is a collision resolution technique in hash tables where, instead of forming a chain when a collision occurs, the object is placed in the next available slot. This approach utilizes contiguous memory to store elements, offering cache locality and faster retrieval compared to chaining, which involves jumping between pointers.

3.2 How Probing works:

Given a hash function h(x), start inserting at the index determined by h(x). If a collision occurs, check the next slot in the array (index + 1), and continue linearly until an empty slot is found.

Example: If the hash table size is 10 and the hash function is h(x) = x mod 10, inserting elements like 7, 8, and 10 will occupy slots 7, 8, and 0 respectively. Inserting 17 will first attempt slot 7 (collision), then slot 8 (collision), then slot 9 (available), so it gets inserted there.

3.2.1 Benefits:

Cache Efficiency: Linear probing works with contiguous memory, making it cache-friendly. Once part of the table is loaded into the cache, probing usually involves examining memory already in the cache, resulting in faster searches.

Avoids Pointer Overhead: Unlike chaining, which uses pointers and involves dynamic memory access, linear probing avoids the overhead of pointer dereferencing.

3.2.2 Pitfalls:

Primary Clustering: Long probing sequences can connect, leading to "clustering." This can degrade performance to linear time (O(n)), especially if the load factor (α) is high.

High Load Factor Issues: As the number of elements (m) approaches the number of slots (N), searching time can degrade significantly. Ideally, m should be about one-third of N (i.e., $\alpha = \text{m/N} \leq 1/3$).

3.3 Analysis of Linear Probing

3.3.1 Load Factor and Performance:

Load Factor (α): Defined as m/N. Keeping α around 1/3 ensures that each object has, on average, 3 slots available, reducing the likelihood of long probing sequences.

Optimal Condition: If m is significantly less than N (e.g., m = N/3), the hash table behaves closer to a perfect hash function, minimizing collisions.

3.3.2 Probability and Chebyshev's Inequality:

To analyze linear probing, instead of focusing on individual slots, we consider regions within the hash table. The expected number of elements in a region S is approximately S/3. Problems arise when the number of elements approaches 2S/3. Chebyshev's inequality can be used to bound the probability that a region becomes overfilled, causing performance degradation.

3.3.3 Alternative Probing Strategies:

Other Probing Methods: Alternatives like quadratic probing (i^2, i^3) or double hashing attempt to spread out the collisions differently. However, they can lose the cache locality benefits provided by linear probing.

No Guarantees: Despite different probing strategies, linear probing with a well-chosen load factor often remains the most efficient in practice due to its balance of simplicity and performance.

3.4 Visualization Concept

Hash Table as a Circle: Visualizing the hash table as a circle where objects are hashed and fill in along arcs can help understand the distribution and clustering of elements. Regions with fewer insertions generally result in faster searches.

4 Cuckoo Hashing

4.1 Definition

Cuckoo hashing is a hashing method that results in a guaranteed O(1) lookup time rather than the O(n) worst case lookup time for chaining or linear probing. It uses two hash tables, T_1 and T_2 each of size n with different hash functions h_1 and h_2.

4.2 Insertion

To insert an element k, the algorithm hashes it with the first table's hash function, placing it in the hash table's index. However, if there was something in that slot before, that value is stored, hashed with the second table's hash function, and stored in that hash table's index instead. And again, if there was something in that index already, it will be stored, hashed with the first table's hash function, and stored. This recursive cycle will keep going until an empty spot is found in the hash table.

Algorithm 1 Cuckoo Hashing Insertion

```
Hash tables $T_1, T_2$ of size $n$ and hash functions $h_1$ and $h_2$
function INSERT(k)
    INSERT TABLE($k, T_1$)
end function

function INSERT TABLE(k, $T_i$)
    if $T_i[h_i(k)]$ is empty then $T_i[h_i(k)] = k$
    else
        $x \leftarrow T_i[h_i(k)]$
        $T_i[h_i(k)] = k$
        INSERT TABLE($x, T_{3-i}$)                    ▷ Insert x into the next table
    end if
end function
```

However, it is possible that a cycle is found instead of an empty space in the hash table. If a value that was just hashed and stored in a table in the same insert operation is removed from that hash table, we found a cycle. This means that given the hash functions and elements in the hash table, it is impossible to store all of the values in the two tables. In this case, new hash functions must be selected and each element in the hash tables must be re-inserted. This is the sacrifice that must be made for worst-case O(1) lookup.

Practically, there is also a maximum loop parameter set in most applications instead of cycle tracking. If the number of calls to INSERT TABLE is more than this parameter, all items will be rehashed with new hash functions.

Additionally, the probability of a cycle decreases with the size of the hash table. Insertion is expected to be O(1) amortized including the possibility of rehashing the whole table as long as the load factor (ratio of elements to the size of the hash table) is below 50 percent.

Using more hash tables is also an effective solution, but this uses more memory.

4.3 Lookup

Lookup is much more simple for insertion. We must just make 1-2 checks to see if the element we are looking for are in the hash tables.

Algorithm 2 Cuckoo Hashing Lookup

```
Hash tables $T_1, T_2$ of size $n$ and hash functions $h_1$ and $h_2$
function LOOKUP(k)
    if $T_1[h_1(k)] = k$ then
        return true
    end if
    if $T_2[h_2(k)] = k$ then
        return true
    end if
    return false
end function
```

4.4 Deletion

Similar to lookup, deletion just requires that checks are made in one or both hash tables. If the element we want to delete is present in the hash table, we delete it from that hash table.

Algorithm 3 Cuckoo Hashing Deletion

```
Hash tables $T_1, T_2$ of size $n$ and hash functions $h_1$ and $h_2$
function DELETE(k)
    if $T_1[h_1(k)] = k$ then
        delete $T_1[h_1(k)]$
        return
    end if
    if $T_2[h_2(k)] = k$ then
        delete $T_2[h_2(k)]$
    end if
end function
```