CS 7280: Data Str & Alg Scalable Comp Spring 2025

Lecture 8 - February 5, 2025

Prof. Prashant Pandey *Scribe: Sina Ahmadi*

1 Previous Lecture

Look at the questions on balls and bins.

2 Hashing

Assuming a universe of size U, where $U = \{0, 1, 2, \ldots, U-1\}$, we have a set called S, where $S \subseteq U$. We are looking to do three operations:

- **insert(x)**: add element $x \in U$ to set S, i.e., $S \cup \{x\}$
- **find(x)**: determine whether element $x \in S$
- **delete(x)**: remove element x from set S, i.e., $S \leftarrow S - \{x\}$

Note: The difference between hashing and the other data structures we have seen so far (e.g., van Emde Boas trees, X-fast and Y-fast trees, and so on) is that we don't do successors. Instead, what we want to achieve is point queries.

Question: What data structure can we use for hashing?

Answer: Hash Tables

> "The three most important data structures are hashing, hashing and hashing."
> — Udi Manber, former CTO of Yahoo!

3 Hash Collisions

Definition: When a record x maps to an already occupied slot in T, a **collision** has occured.

3.1 Avoiding Collisions

Collisions cannot be avoided altogether. However, we should try to keep it as low as possible, while dealing with collisions. There are certain ways to resolve this issue.

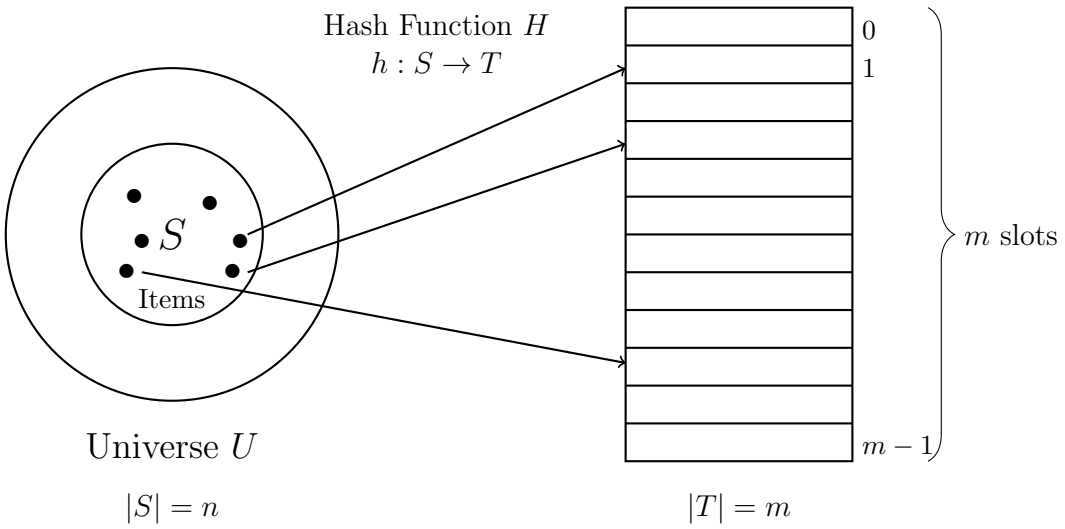

Figure 1: Hashing

3.1.1 Chaining

In the event of a collision, x will be written to the tail of a linked list in that location. This way, we will have a chain of elements starting from that specific slot. Assuming we have n keys and m slots, $n \leq m$ for most cases. Figure 2 shows a simple example of how chaining works.

Question: Are operations constant time when chaining?

Answer: Not necessarily, not with high probability. We need to bound the size of linked lists.

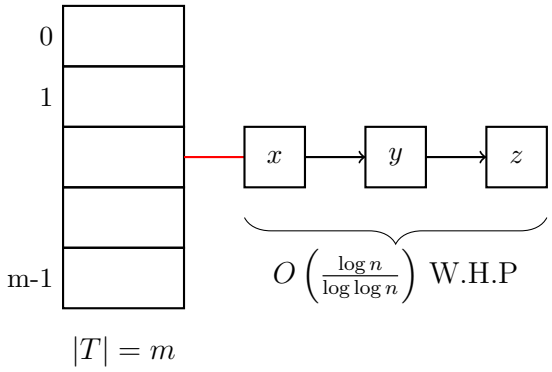

Figure 2: Chaining

3.1.2 Open Addressing

No storage is used outside of hashtable itself. Insertion systematically probes the table until an empty slot is found.

Open addressing can lead to what's called *clustering effect*. Clustering refers to the problem where groups of consecutive occupied slots build up in the hash table. This can make finding an empty

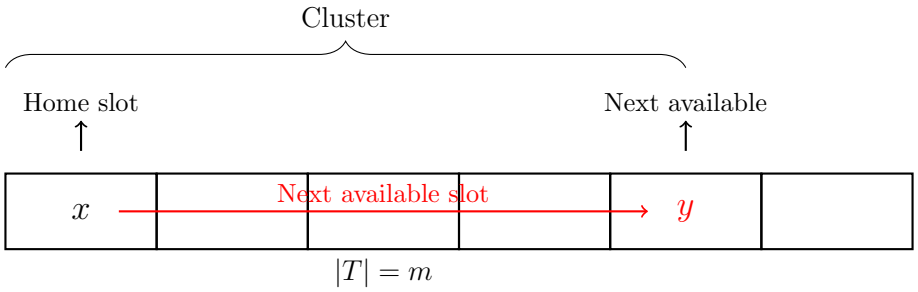

Figure 3: Open Addressing

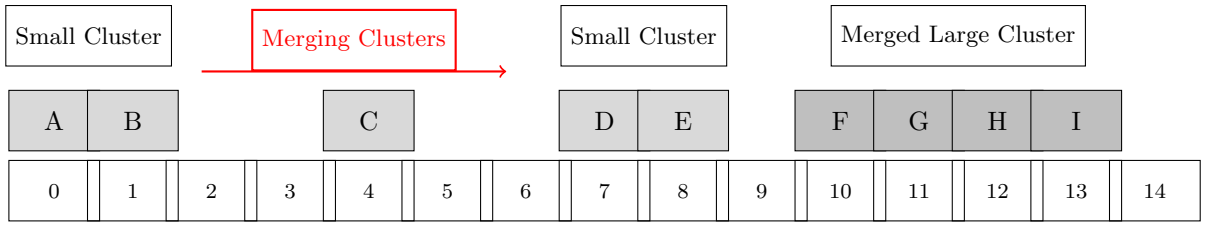

Figure 4: Heavy-tailed

slot slower and worsen the performance of both insertions and lookups.

We can use different probing strategies to find the next available slot for a key that faces collision.

Linear Probing
It is a widely employed strategy by big companies. Upon a collision (when two keys hash to the same index), the algorithm checks the next available slot sequentially (index + 1, index + 2, and so on) until an empty slot is found. The search wraps around to the beginning of the table if necessary. While simple and easy to implement, it is susceptible to primary clustering, where long groups of consecutive occupied slots can form, leading to slower insertions and lookups as the table fills.

We can have the probing length bounded by O(log n) under ideal conditions due to probabilistic properties. When a good hash function distributes keys uniformly and the hash table is not too full (load factor $\alpha < 1$), the expected length of probing sequences grows logarithmically with the number of keys. This happens because the probability of encountering long chains of filled slots decreases exponentially as the chains grow, making most probes short. As a result, with proper resizing and uniform key distribution, long probing sequences are rare, and the performance remains efficient.

Note (Heavy-tailed phenomenon): Linear probing in hash tables causes collisions to form small clusters of consecutive occupied slots. As keys are inserted and collisions push them forward, small clusters merge into larger ones, leading to primary clustering. Over time, a few large clusters dominate, creating heavy-tailed clustering, where most clusters remain small, but some grow disproportionately large. These large clusters degrade performance, as future keys take longer to find empty slots. To mitigate this, good hash functions, low load factors, and alternative probing methods like quadratic probing or double hashing are recommended.

Figure 4 demonstrates how small clusters form and gradually merge into larger clusters during linear probing. Initially, small clusters (e.g., containing keys `A`, `B` or `D`, `E`) form due to collisions when keys are inserted into adjacent slots. As more keys are inserted and collisions are resolved by probing the next available slots, these small clusters merge into a large cluster (e.g., containing keys `F`, `G`, `H`, `I`). Over time, this merging effect creates large, heavy-tailed clusters, where most clusters remain small, but a few large clusters dominate the table. This leads to performance degradation, as future insertions and searches take longer when encountering large clusters. The arrow labeled "Merging Clusters" shows this gradual merging of small clusters into larger, inefficient clusters.

Question: Given what was said about open addressing using linear probing, is there any advantage in using that versus chaining?

Answer: There is, namely:

- **Memory Efficiency:** Open addressing stores all keys directly in the hash table, avoiding the need for additional memory allocation for linked lists, which is required in chaining. This can be beneficial when memory is limited.
- **Cache Friendliness:** Linear probing accesses consecutive memory locations during probing, taking advantage of *spatial locality* and improving cache performance. Chaining, on the other hand, involves pointer dereferencing and accessing scattered memory locations, which can lead to cache misses.
- **Simpler Data Structure:** Open addressing does not require external linked lists or dynamic memory management, making it simpler to implement compared to chaining.
- **No Overhead for Linked Lists:** In chaining, each slot points to a linked list, adding memory overhead and extra time to traverse the list. Linear probing avoids this overhead entirely.

4 A Note on Memory Hierarchy

The memory hierarchy in Figure 5 highlights the significant difference in access times across various levels of memory, from registers and caches to main memory (RAM) and disk storage. In the context of open addressing versus chaining, this difference is crucial. With open addressing, keys are stored directly in the hash table, often within contiguous memory locations, allowing faster access due to cache locality. This benefits from the fast access times of the CPU caches (e.g., L1 cache access is 0.3 ns, compared to 62.9 ns for main memory). In contrast, chaining requires linked lists that involve pointer dereferencing, which scatters memory access patterns and increases the likelihood of fetching data from RAM, resulting in much slower access. Since fetching from main memory is significantly slower than accessing cache (by nearly two orders of magnitude), open addressing typically outperforms chaining in scenarios where cache locality is critical, especially when the load factor is low and hash collisions are minimal.

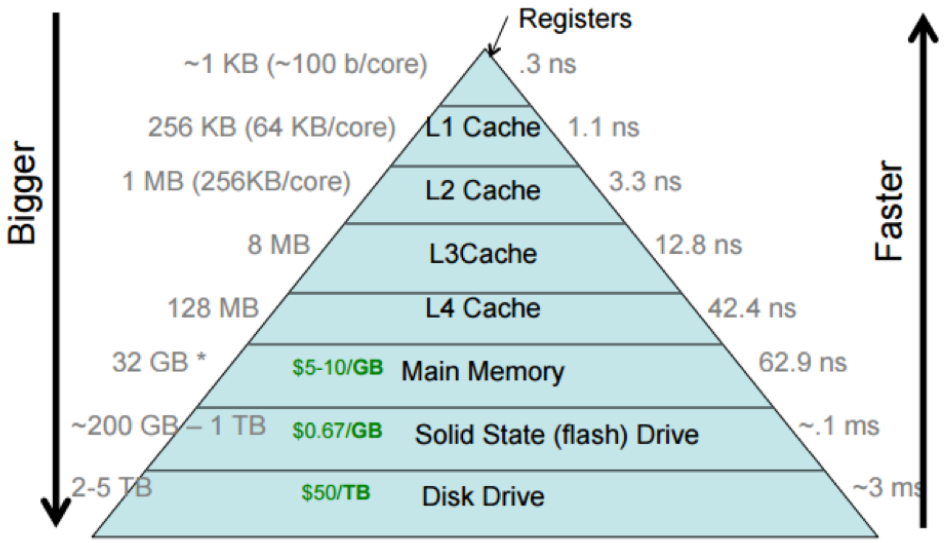

Figure 5: Memory Hierarchy

5 Hash Function with Strong Guarantees

Total Random: Total random (or truly random hashing) refers to an idealized and theoretical concept where the hash function behaves like a completely random function. That is, for any input x , the hash value h(x) is chosen uniformly at random from the entire possible range of hash values. Importantly, the outputs are independent of each other, meaning that knowing the hash of one input provides no information about the hash of another input.

$$\mathbb{P}[h(x) = t] = \frac{1}{m}$$

Independent of $h(y)$ for all $x \neq y \in U$

Information: $\Theta(U \log m)$ bits of space are required to make this work — **too big!**

This can be implemented using simple uniform hashing.

6 Universal Hashing

In contract to total random, universal hashing [1] is a more practical approach. Choose h from family H of hash functions with probability that

$$\mathbb{P}[h(x) = h(y)] = o(\frac{1}{m})$$

Let's build a simple univesal hash function:

Hash function 1

$$h(x) = [a \cdot x \mod p] \mod m, \quad \text{where } |T| = m$$

$$0 < a < p \quad \text{and} \quad p < |U|$$

Hash function 2

$$h(x) = (a \cdot x \gg (\log U - \log m)) \text{ bits}$$

Where m and U are both powers of two.

7 k-Wise Independence:

k-Wise Independent: A family $\mathcal{H}$ of hash functions which satisfy:

$$\mathop{\mathbb{P}}_{h \in \mathcal{H}}[h(x_1) = t_1 \wedge h(x_2) = t_2 \wedge \cdots \wedge h(x_k) = t_k] = O\left(\frac{1}{m^k}\right)$$

for all distinct inputs $x_1, x_2, \ldots, x_k \in U$.

Note: Pairwise (2-wise) independence is **stronger** than universal hashing.

Note 2: Pairwise independence and even higher levels of k-wise independence can be computationally expensive to implement in practice. One can use tools like *Objdump* to see the number of actual CPU instructions.

$$h(x) = [(a \cdot x + b) \mod p] \mod m$$
$$0 < a < p, \quad 0 \leq b < p$$

where a and b are randomly chosen integers and

Note: For stronger pairwise independence (e.g., 3-wise, ...):

$$h(x) = [\left(\sum_{i=0}^{k} a_i x^i\right) \mod p] \mod m$$
$$0 < a_{k-1} < p, \quad 0 \leq a_i < p$$

This requires $O(k)$ time to compute.

—

Chaining Bounds:

$$\mathbb{E}\left[c_t = \text{length of chain t}\right] = \sum_i \mathbb{P}[h(x_i) = t]$$

—

Universal Bounds:

$$O\left(\frac{n}{m}\right) = O(1) \quad \text{for } m = \Omega(n)$$

Variance $[c_t]$

$$E[C_t^2] - E[C_t]^2$$

Assuming h is symmetric:

$$E[c_t^2] = \frac{1}{m}\sum E[c_s^2]$$

$$= \frac{1}{m}\sum_i \mathbb{P}[h(x_i) = h(x_j)]$$

$$= \frac{1}{m} n^2 O(\frac{1}{m})$$

$$= O(\frac{n^2}{m^2})$$

$$= O(1) \quad \text{for} \quad m = \Omega(n)$$

—

References

[1] J. L. Carter, M. N. Wegman. *Universal Classes of Hash Functions*. Proceedings of the 9th Annual ACM Symposium on Theory of Computing, 1977.