Hashing

Prepared by:

Dr Annushree Bablani

The Dictionary Problem

- Given a series of operations: OP_1, OP_2, ... These operations are to be performed on an initially empty set S. Each operation OP_i can be one of the following:
  - Insert(x) An item with key value x is inserted into the set S.
  - Lookup(x) Check if an item with key value x is present in the set S.
  - Delete(x) The item with key value x, if present, is deleted from the set S.

The Dictionary Problem

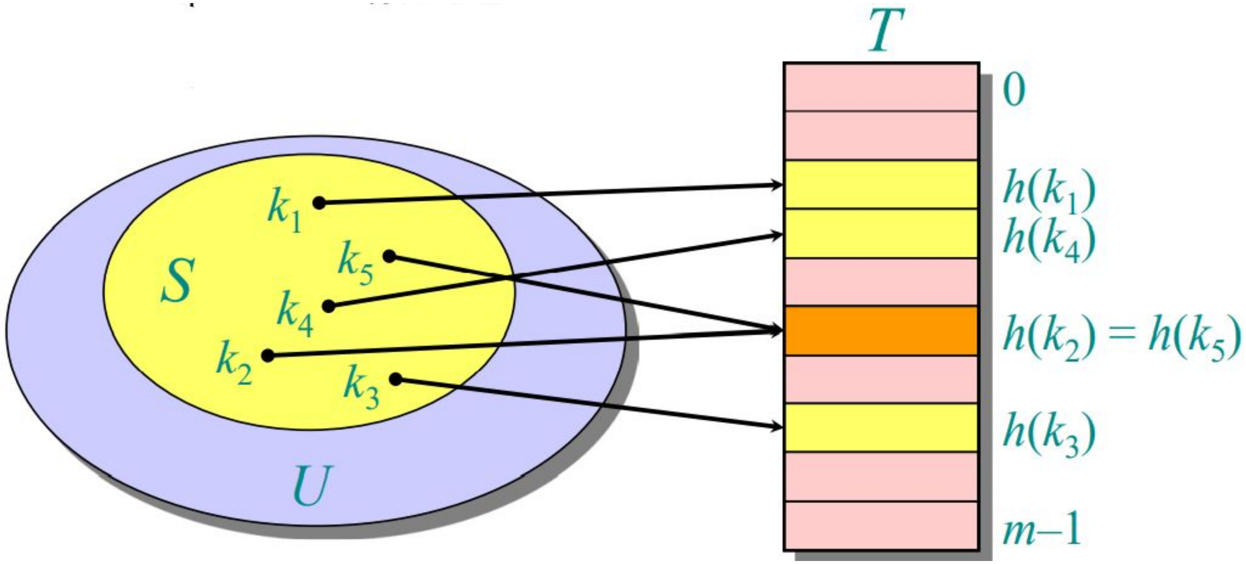
- Each of the key value *x* comes from a universe *U*, i.e. $x \in U$. In this document, we assume U = {1, 2, . . . N}.
- Observe that the set *S* is a dynamic set. Each of the Insert and Delete operations may modify the set.
- Hence the size of the set S changes with each operation.
- We bound the maximum size of the set to n (n << N).

Hash tables and hash functions

- What are the data structures that can be used to store the set S?
  - One option is to use a balanced binary search tree. But each of the operations would take O(log n) time. Moreover, a balanced binary search tree is more difficult to implement than, say, an array, or a singly linked list.
  - Could we store the set S in an array? Is there a data structure that would perform the above operations in constant time?
    - Yes, there is such a data structure, hash table, which provides an easy way of storing such information.

Hash tables and hash functions

- Let us denote the hash table by T . A hash function maps the elements of the universe to the hash table, $h : U \longrightarrow T$.

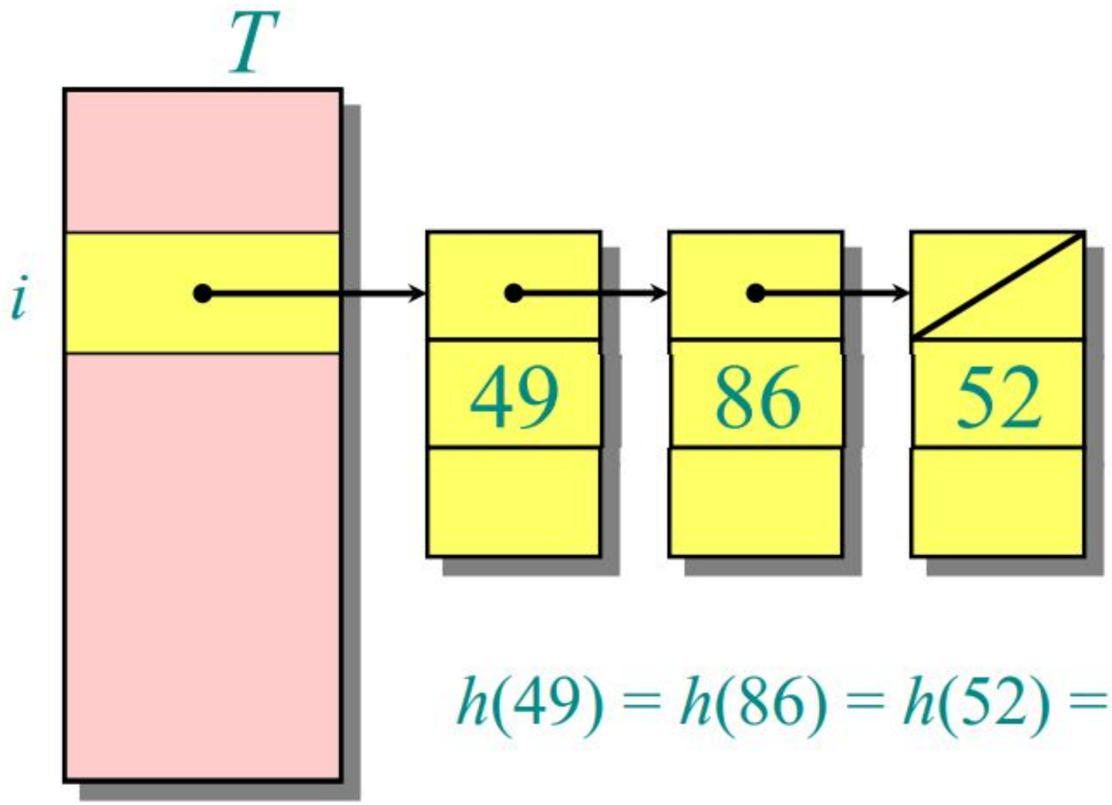


Hash tables and hash functions

- Let the size of the hash table be m. We would like the size of the hash table, m to be linear in the size of the set S.
- The hash table cannot be as large as the universe of keys, N, in which case, there would be no collisions.
- When the hash table is smaller than the universe, ***collisions will occur***.

Resolving Collision

- Link records in the same slot into a list.
- Worst case:
  - Every key hashes to the same slot.
  - Access time = Θ(n) if |S| = n



$h(49) = h(86) = h(52) = i$

Average-case analysis of chaining

- We assume of simple uniform hashing:
  - Each key $k \in S$ is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.
- Let n be the number of keys in the table, and let m be the number of slots.
- Define the load factor of T to be $\alpha = n/m$ = average number of keys per slot.

Search Cost

- The expected time for an unsuccessful search for a record with a given key is

$$= \Theta(\underbrace{1}_{\text{apply hash function and access slot}} + \underbrace{\alpha}_{\text{search the list}}).$$

- Expected search time = $\Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.
- A successful search has same asymptotic bound, but a rigorous argument is a little more complicated.

Choosing Hash function

- The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.
- Desired:
  - A good hash function should distribute the keys uniformly into the slots of the table.
  - Regularity in the key distribution should not affect this uniformity

Division Method

- Assume all keys are integers, and define *h(k) = k mod m*.
- Deficiency: Don't pick an ***m*** that has a small divisor ***d***. A preponderance of keys that are congruent *modulo d* can adversely affect uniformity.
- Extreme deficiency: If $m = 2^r$ then the hash doesn't even depend on all the bits of k:

$$h(k) = k \bmod m.$$

If $k = 1011000111\underbrace{011010}_{h(k)}{}_2$ and $r = 6$, then $h(k) = 011010_2$.

Division Method

- Pick ***m*** to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.
- Annoyance:
  - Sometimes, making the table size a prime is inconvenient.
  - But this method is popular, although the next method we'll see is usually superior.

Multiplication method

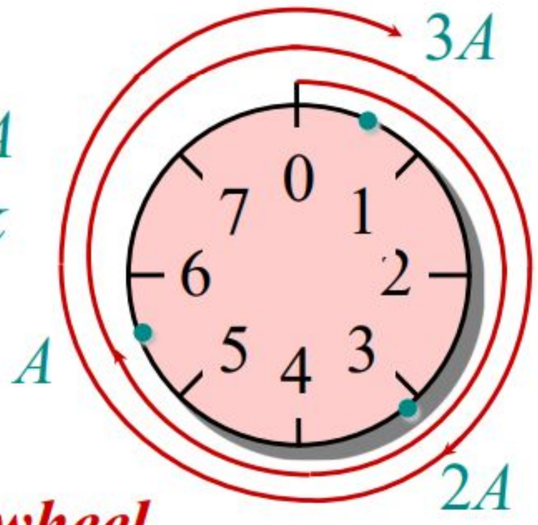
- Assume that all keys are integers, $m = 2^r$, and our computer has w-bit words.
- Define $\boldsymbol{h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)}$,
- where ***rsh*** is the "bitwise right-shift" operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.
- Don't pick A too close to 2^{w-1} or 2^w.
- Multiplication *modulo* 2^w is fast compared to division.
- The rsh operator is fast.

Multiplication method

$$h(k) = (A \cdot k \bmod 2^w) \operatorname{rsh} (w - r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words:

$$\begin{array}{lr}
 & 1\,0\,1\,1\,0\,0\,1 = A \\
\times & 1\,1\,0\,1\,0\,1\,1 = k \\
\hline
 & 1\,0\,0\,1\,0\,1\,0\,\underbrace{0\,1\,1}_{h(k)}\,0\,0\,1\,1
\end{array}$$



Modular wheel

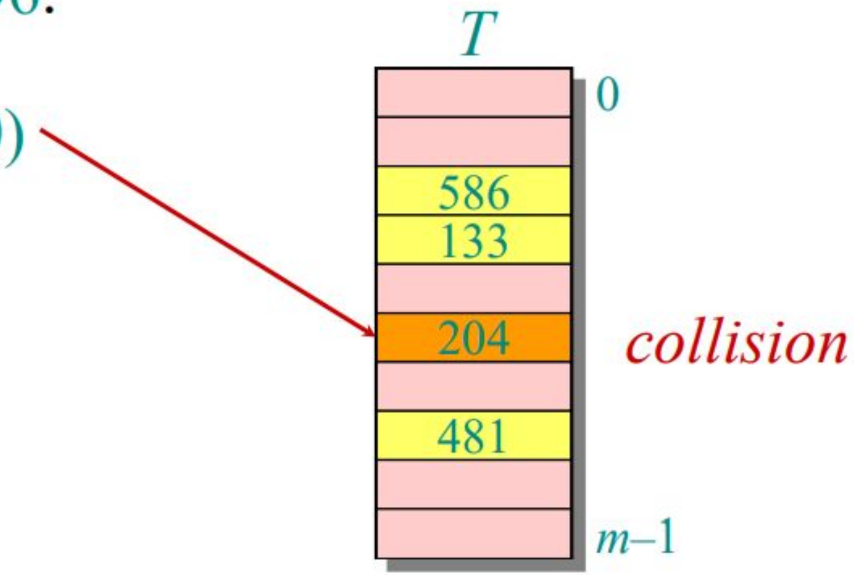
Resolving collisions by open addressing

- No storage is used outside of the hash table itself.
- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:
  - $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$.
- The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible).

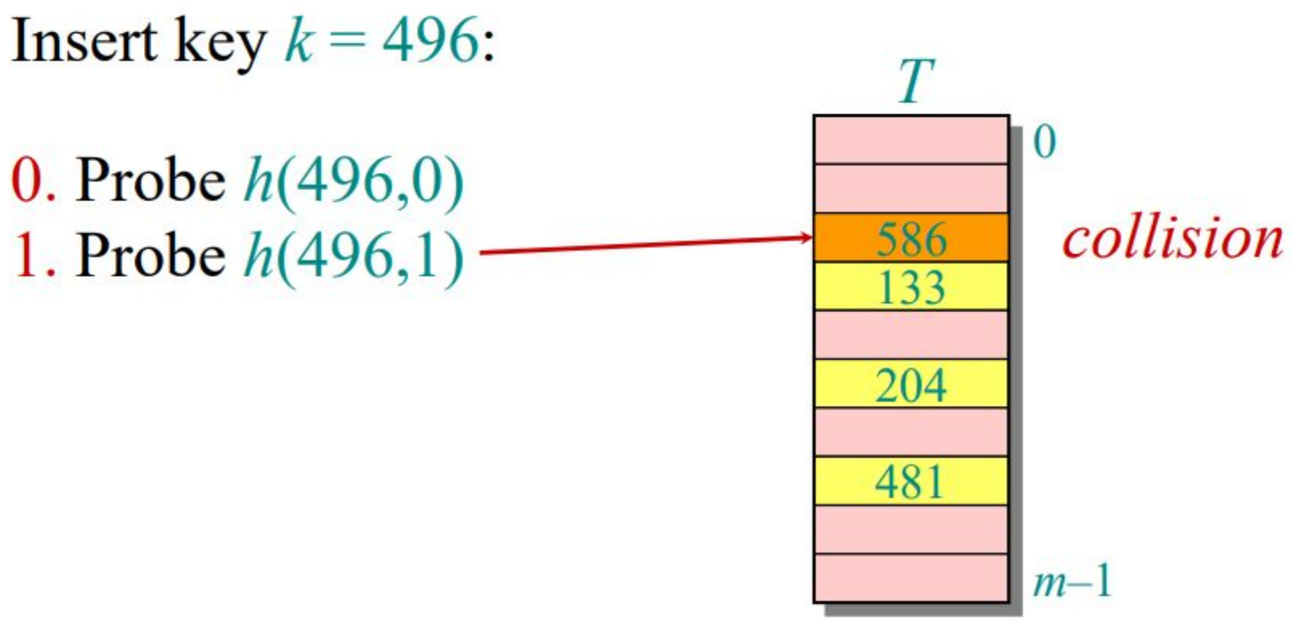
Example of open addressing

Insert key $k = 496$:

0. Probe $h(496,0)$



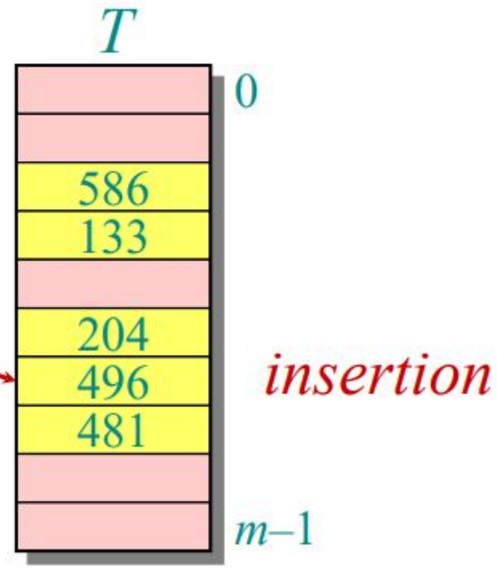
Example of open addressing



Example of open addressing

Insert key $k = 496$:

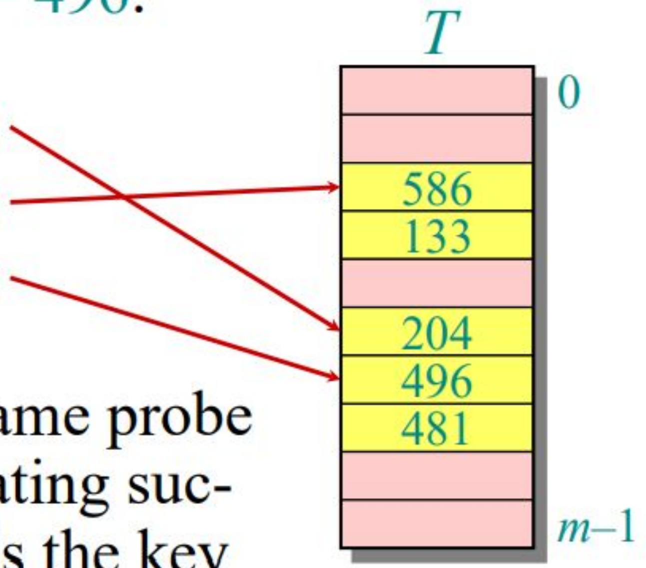
0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Example of open addressing

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

Probing strategies

- Linear probing:
  - Given an ordinary hash function *h'(k),* linear probing uses the hash function
    - ***h(k,i) = (h'(k) + i) mod m.***
  - This method, though simple, suffers from primary clustering, where long runs of occupied slots build up, increasing the average search time.
  - Moreover, the long runs of occupied slots tend to get longer.
- Double hashing:
  - Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function
    - $\boldsymbol{h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m}$.
  - This method generally produces excellent results, but $h_2(k)$ must be relatively prime to *m*.
  - One way is to make *m* a power of 2 and design $h_2(k)$ to produce only odd numbers

Analysis of open addressing

- We assume of uniform hashing:
    - Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.
    - Theorem:
        - Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Proof of the theorem

- Proof:
  - At least one probe is always necessary.
  - With probability n/m, the first probe hits an occupied slot, and a second probe is necessary.
  - With probability (n–1)/(m–1), the second probe hits an occupied slot, and a third probe is necessary.
  - With probability (n–2)/(m–2), the third probe hits an occupied slot, etc.
  - Observe that

$$\frac{n-i}{m-i} < \frac{n}{m} = \alpha \text{ for } i = 1, 2, \ldots, n.$$

Proof of the theorem

Therefore, the expected number of probes is

$$
\begin{aligned}
&1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right) \\
&\le 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots))) \\
&\le 1+\alpha+\alpha^2+\alpha^3+\cdots \\
&= \sum_{i=0}^{\infty}\alpha^i \\
&= \frac{1}{1-\alpha}\ . \quad \square
\end{aligned}
$$

The textbook has a more rigorous proof and an analysis of successful searches.

Implications of the theorem

- If α is constant, then accessing an open addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is
  $1/(1–0.5) = 2$.
- If the table is 90% full, then the expected number of probes is
  $1/(1–0.9) = 10$.

Universal Hashing

- How do we design a good hash function?
  - A set S of keys from a universe $U = \{0,1,...,m - 1\}$ is supposed to be stored in a table of size n with indices $T = \{0,1,...,n - 1\}$.
    - Assume collisions are resolved using auxiliary data structure.
  - What we need is a hash function $h : U \rightarrow T$ with the following main requirements:
    - The hash function should minimize the number of collisions.
    - The space used should be proportional to the number of keys stored. (i.e., $n \approx |S|$)

Universal Hashing

- Claim 1: If $m > n$, then for any h there exists a key set S such that h has collision w.r.t. S (i.e., $\exists x,y \in S, h(x) = h(y)$)
  - Claim 1.1: Any fixed hash function $h : U \rightarrow T$, must map at least elements of U to some index in the set T.
- Claim 2: For any fixed key set S such that $|S| \leq n$, there exists a hash function such that h has no collisions w.r.t. S.
- The issue is that the key set S is *not known* a-priori. That is, before using the data structure.
- Question: How do we solve this problem then?

Universal Hashing

- Question: How do we solve this problem then?
    - ***Randomly*** select a hash function from a ***family*** H of hash functions.

Definition (2-universality)

A hash function family H is said to be 2-universal iff:

$$\forall x, y \in U, x = y, \Pr_{h \leftarrow H}[h(x) = h(y)] \leq \frac{1}{n}.$$

Universal Hashing

- **Theorem**: Consider hashing using a 2-universal hash function family. Consider t insert operations, the expected cost of each operation is at most *(1 + t/n).*
  - **Proof sketch**: Consider any key *x*. The expected number of keys in location *h(x)* is at most *t/n*.

Definition (2-universality)

A hash function family H is said to be 2-universal iff:

$$\forall x, y \in U, x = y, \Pr_{h \leftarrow H}[h(x) = h(y)] \leq \frac{1}{n}.$$

Perfect hashing

- Given a set of n keys, construct a static hash table of size m = O(n) such that SEARCH takes Θ(1) time in the worst case.
- IDEA: Two level scheme with universal hashing at both levels.
    - *No collisions at level 2!*