

Hashing - Introduction

- **Dictionary** = a dynamic set that supports the operations INSERT, DELETE, SEARCH
- Examples :
 - ◆ a symbol table created by a compiler
 - ◆ a phone book
 - ◆ an actual dictionary
- **Hash table** = a data structure good at implementing dictionaries

Hashing - Introduction

- Why not just use an array with **direct addressing** (where each array cell corresponds to a key)?
 - ◆ Direct-addressing guarantees **$O(1)$ worst-case** time for Insert/Delete/Search.
 - ◆ BUT sometimes, the number K of keys actually stored is very small compared to the number N of possible keys. Using an array of size N would **waste space**.
 - ◆ We'd like to use a structure that takes up $\Theta(K)$ space and $O(1)$ average-case time for Insert/Delete/ Search

Hashing

- Hashing =
 - ◆ use a table (array/vector) of size m to store elements from a set of much larger size
 - ◆ given a key k , use a function h to compute the slot $h(k)$ for that key.
- Terminology:
 - ◆ h is a hash function
 - ◆ k hashes to slot $h(k)$
 - ◆ the hash value of k is $h(k)$
 - ◆ collision : when two keys have the same hash value

Hashing

- What makes a good hash function?
 - ◆ It is easy to compute
 - ◆ It satisfies uniform hashing
- hash = to chop into small pieces (Merriam-Webster)
 - = to chop any patterns in the keys so that the results are uniformly distributed (cs311)

Hashing

- What if the key is not a natural number?
- We must find a way to represent it as a natural number.
- Examples:
 - ◆ key *i* → Use its ascii decimal value, 105
 - ◆ key *inx* → Combine the individual ascii values in some way, for example,
$$105*128^2+110*128+120= 1734520$$

Hashing - hash functions

Truncation

- Ignore part of the key and use the remaining part directly as the index.
- *Example:* if the keys are 8-digit numbers and the hash table has 1000 entries, then the first, fourth and eighth digit could make the hash function.
- Not a very good method : does not distribute keys uniformly

Hashing

Folding

- Break up the key in parts and combine them in some way.
- *Example* : if the keys are 8 digit numbers and the hash table has 1000 entries, break up a key into three, three and two digits, add them up and, if necessary, truncate them.
- Better than truncation.

Hashing

Division

- If the hash table has m slots, define

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

- Fast
- Not all values of m are suitable for this. For example powers of 2 should be avoided.
- Good values for m are **prime numbers** that are not very close to powers of 2.

Hashing

Multiplication

- $h(k) = \lfloor m * (k * c - \lfloor k * c \rfloor) \rfloor, 0 < c < 1$
- In English :
 - ◆ Multiply the key k by a constant $c, 0 < c < 1$
 - ◆ Take the fractional part of $k * c$
 - ◆ Multiply that by m
 - ◆ Take the floor of the result
- The value of m does not make a difference
- Some values of c work better than others
- A good value is $(\sqrt{5} - 1) / 2$

Hashing

Multiplication

- *Example:*

Suppose the size of the table, m , is 1301.

For $k=1234$, $h(k)=850$

For $k=1235$, $h(k)=353$

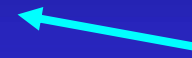
For $k=1236$, $h(k)=115$

For $k=1237$, $h(k)=660$

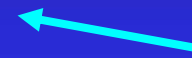
For $k=1238$, $h(k)=164$

For $k=1239$, $h(k)=968$

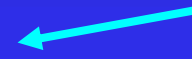
For $k=1240$, $h(k)=471$



pattern broken



distribution fairly



uniform



Hashing

Universal Hashing

- Worst-case scenario: The chosen keys all hash to the same slot. This can be avoided if the **hash function is not fixed**:
- Start with a collection of hash functions
- Select one in random and use that.
- **Good performance on average**: the probability that the randomly chosen hash function exhibits the worst-case behavior is very low.

Hashing

Universal Hashing

- Let H be a collection of hash functions that map a given universe U of keys into the range $\{0, 1, \dots, m-1\}$.
- If for each pair of distinct keys $k, l \in U$ the number of hash functions $h \in H$ for which $h(k) = h(l)$ is $|H| / m$, then H is called universal.

Hashing

- Given a hash table with m slots and n elements stored in it, we define the **load factor** of the table as $\lambda = n/m$
- The load factor gives us an **indication of how full** the table is.
- The possible values of the load factor depend on the method we use for resolving collisions.

Hashing - resolving collisions

Chaining a.k.a closed addressing

- Idea : put all elements that hash to the same slot in a **linked list** (chain). The slot contains a pointer to the head of the list.
- The load factor indicates the average number of elements stored in a chain. It could be less than, equal to, or larger than 1.

Hashing - resolving collisions

Chaining

- Insert : $O(1)$
 - ◆ worst case
- Delete : $O(1)$
 - ◆ worst case
 - ◆ assuming doubly-linked list
 - ◆ it's $O(1)$ after the element has been found
- Search : ?
 - ◆ depends on length of chain.

Hashing - resolving collisions

Chaining

- Assumption: simple uniform hashing
 - ◆ any given key is equally likely to hash into any of the m slots
- **Unsuccessful search:**
 - ◆ average time to search unsuccessfully for key $k =$ the average time to search to the end of a chain.
 - ◆ The average length of a chain is λ .
 - ◆ Total (average) time required : $\Theta(1 + \lambda)$

Hashing - resolving collisions

Chaining

■ Successful search:

- ◆ expected number e of elements examined during a successful search for key k
= 1 more than the expected number of elements examined when k was inserted.
 - ◆ it makes no difference whether we insert at the beginning or the end of the list.
- ◆ Take the average, over the n items in the table, of 1 plus the expected length of the chain to which the i th element was added:

Hashing - resolving collisions

Chaining

$$e = \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m} \right) = \dots = 1 + \frac{\lambda}{2} - \frac{1}{2m}$$

– Total time : $\Theta(1 + \lambda)$

Hashing - resolving collisions

Chaining

- Both types of search take $\Theta(1 + \lambda)$ time on average.
- If $n = O(m)$, then $\lambda = O(1)$ and the total time for Search is $O(1)$ on average
- Insert : $O(1)$ on the worst case
- Delete : $O(1)$ on the worst case

- Another idea: Link all unused slots into a free list

Hashing - resolving collisions

Open addressing

- Idea:
 - ◆ Store all elements in the hash table itself.
 - ◆ If a collision occurs, find another slot. (How?)
 - ◆ When searching for an element examine slots until the element is found or it is clear that it is not in the table.
 - ◆ The sequence of slots to be examined (probed) is computed in a systematic way.
- It is possible to fill up the table so that you can't insert any more elements.
 - ◆ idea: extendible hash tables?

Hashing - resolving collisions

Open addressing

- Probing must be done in a systematic way (why?)
- There are several ways to determine a probe sequence:
 - ◆ linear probing
 - ◆ quadratic probing
 - ◆ double hashing
 - ◆ random probing