Hashing

Hashing Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don't care about sorted order

Hash Tables – contd.

- More formally:
  - Given a table T and a record x, with key (= symbol) and satellite data, we need to support:
    - Insert (T, x)
    - Delete (T, x)
    - Search(T, x)
  - We want these to be fast, but don't care about sorting the records
- The structure we will use is a *hash table*
  - Supports all the above in O(1) expected time!

Hashing: Keys

- In the following discussions we will consider all keys to be (possibly large) natural numbers
- *How can we convert floats to natural numbers for hashing purposes?*
- *How can we convert ASCII strings to natural numbers for hashing purposes?*

Direct Addressing

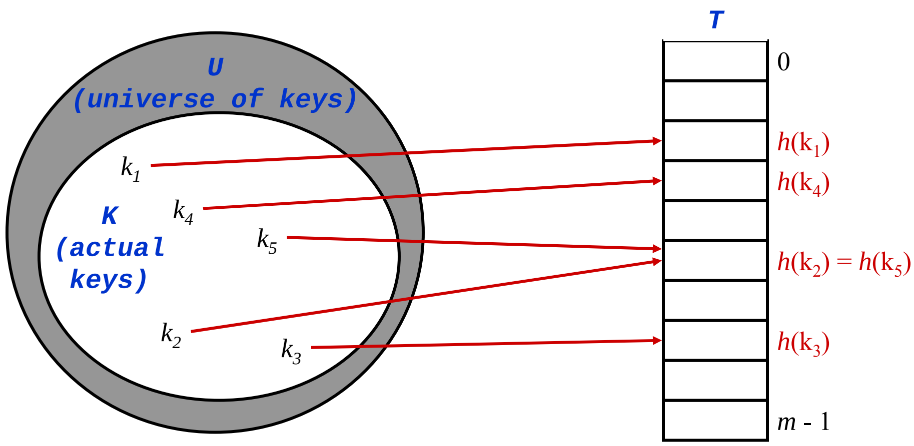
- Suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- The idea:
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

The Problem With Direct Addressing

- Direct addressing works well when the range m of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..m-1
- This mapping is called a *hash function*

Hash Functions

- Next problem: *collision*



Resolving Collisions

- *How can we solve the problem of collisions?*
- Solution 1: *chaining*
- Solution 2: *open addressing*

Resolving Collisions – contd.

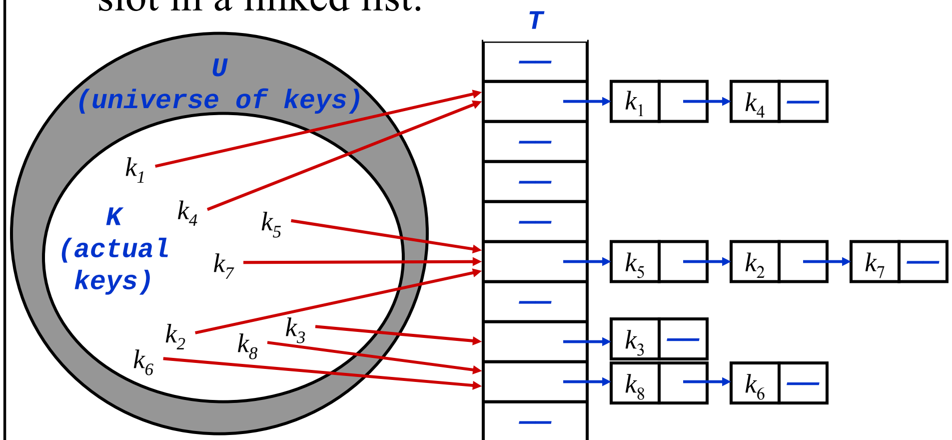
- *How can we solve the problem of collisions?*
- *Open addressing*
  - To insert: if slot is full, try another slot, and another, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
- *Chaining*
  - Keep linked list of elements in slots
  - Upon collision, just add new element to list

Open Addressing

- Basic idea (details in Section 11.4):
  - To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than n

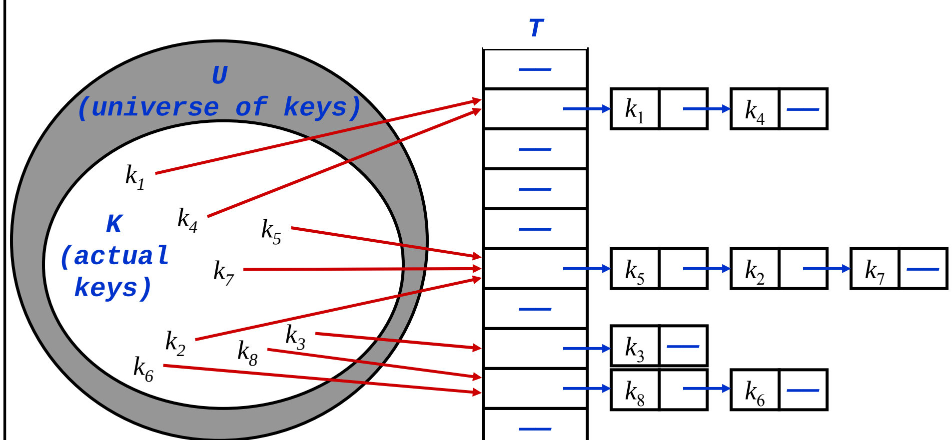
Chaining

- Chaining puts elements that hash to the same slot in a linked list:



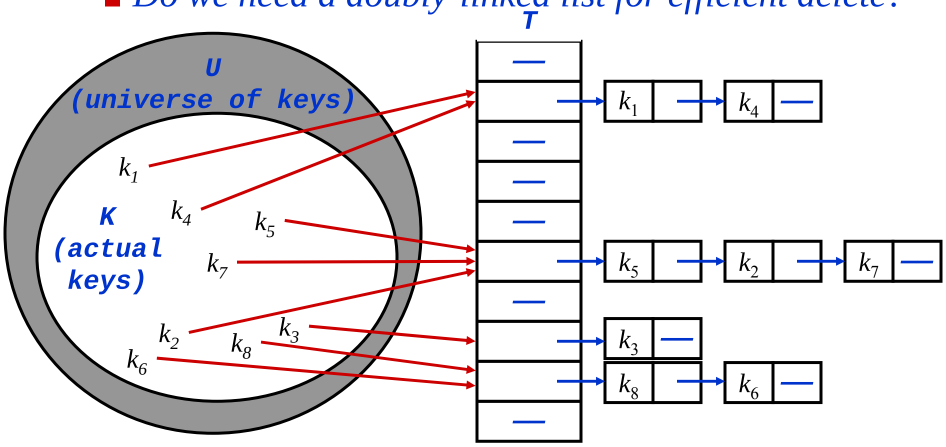
Chaining

- *How do we insert an element?*



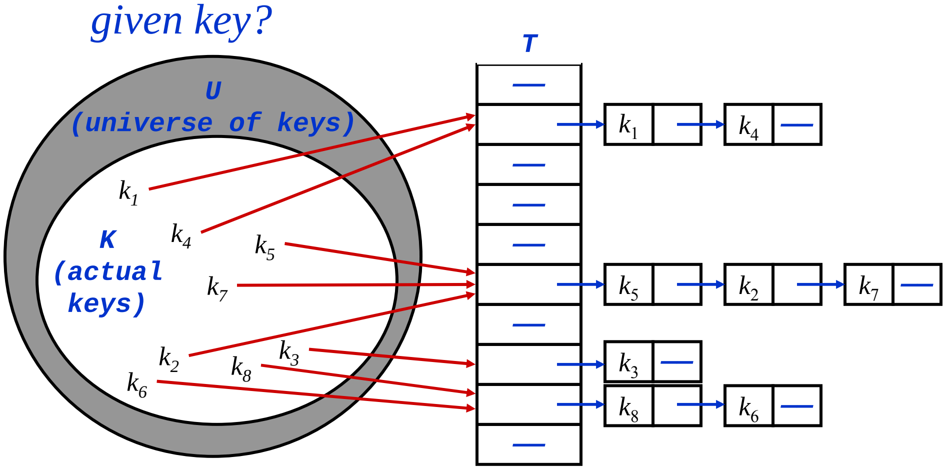
Chaining

- *How do we delete an element?*
  - *Do we need a doubly-linked list for efficient delete?*



Chaining

- *How do we search for a element with a given key?*



Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table: the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- *What will be the average cost of a successful search?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

- *What will be the average cost of a successful search?* A: $O(1 + \alpha/2) = O(1 + \alpha)$

Analysis of Chaining Continued

- So the cost of searching = $O(1 + \alpha)$
- *If the number of keys n is proportional to the number of slots in the table, what is α?*
- A: $\alpha = O(1)$
  - In other words, we can make the expected cost of searching constant if we make α constant

Choosing A Hash Function

- Clearly choosing the hash function well is crucial
  - *What will a worst-case hash function do?*
  - *What will be the time to search in this case?*
- *What are desirable features of the hash function?*
  - Should distribute keys uniformly into slots
  - Should not depend on patterns in the data

Hash Functions: The Division Method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- *What happens to elements with adjacent values of k?*
- *What happens if m is a power of 2 (say 2^P)?*
- *What if m is a power of 10?*
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Hash Functions: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\text{What does this term represent?}} \rfloor$

Hash Functions: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of kA}} \rfloor$
- Choose $m = 2^P$
- Choose A not too close to 0 or 1
- Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions: Worst Case Scenario

- Scenario:
  - You are given an assignment to implement hashing
  - You will self-grade in pairs, testing and grading your partner's implementation
  - In a blatant violation of the honor code, your partner:
    - Analyzes your hash function
    - Picks a sequence of "worst-case" keys, causing your implementation to take $O(n)$ time to search
- *What's an honest CS student to do?*

Hash Functions: Universal Hashing

- As before, when attempting to foil an malicious adversary: randomize the algorithm
- *Universal hashing*: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
  - Guarantees good performance on average, no matter what keys adversary chooses

Universal Hashing – contd.

- When attempting to foil an malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins (*not* upon every insert!)
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from

Universal Hashing

- Let ς be a (finite) collection of hash functions
  - …that map a given universe U of keys…
  - …into the range $\{0, 1, \ldots, m - 1\}$.
- ς is said to be *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions h $\in \varsigma$ for which $h(x) = h(y)$ is $|\varsigma|/m$
  - In other words:
    - With a random hash function from ς, the chance of a collision between x and y is exactly $1/m$ $(x \neq y)$

Universal Hashing

- Theorem 11.3:
  - Choose h from a universal family of hash functions
  - Hash n keys into a table of m slots, $n \leq m$
  - Then the expected number of collisions involving a particular key x is less than 1
  - Proof:
    - For each pair of keys y, z, let $c_{yx} = 1$ if y and z collide, 0 otherwise
    - $E[c_{yz}] = 1/m$ (by definition)
    - Let C_x be total number of collisions involving key x
    - $E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] = \frac{n-1}{m}$
    - Since $n \leq m$, we have $E[C_x] < 1$

A Universal Hash Function

- Choose table size m to be prime
- Decompose key x into r+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < m
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of r+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function $h_a \in \varsigma$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

  - With this definition, ς has m^{r+1} members

A Universal Hash Function

- ς is a universal collection of hash functions (Theorem 11.4)
- How to use:
  - Pick r based on m and the range of keys in U
  - Pick a hash function by (randomly) picking the a's
  - Use that hash function on all keys

Open Addressing

- Basic idea (details in Section 11.4):
  - To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than *n*

Open addressing

- Store all elements in the table
- Probe the hash table in event of a collision
- Key idea: probe sequence is NOT the same for each element, depends on initial key
- h: U x {0, 1, …, m-1} → {0, 1, …, m-1}
- Permutation requirement
  - h(k,0), h(k,1), …, h(k,m-1) is a permutation of (0, …, m-1)

Operations

- Insert, search straightforward
- Why can we not simply mark a slot as deleted?
  - If keys need to be deleted, open addressing may not be the right choice

Probing schemes

- uniform hashing: each of m! permutations equally likely
  - not typically achieved
- linear probing: $h(k,i) = (h'(k) + i) \bmod m$
  - Clustering effect
  - Only m possible probe sequences are considered
- quadratic probing: $h(k,i) = (h'(k)+ci+di^2) \bmod m$
  - constraints on c, d, m
  - better than linear probing as clustering effect is not as bad
  - Only m possible probe sequences are considered, and keys that map to same position do have identical probe sequences
- double hashing: $h(k,i) = (h(k) + iq(k)) \bmod m$
  - q(k) must be relatively prime wrt m
  - m^2 probe sequences considered
  - Much closer to uniform hashing

Search time

- Preliminaries
  - n elements, m slots, $\alpha = n/m$ with $n \le m$
  - Assumption of uniform hashing
- Expected search time on a miss
  - Given that h(k,i) is non-empty, what is the probability that h(k,i+1) is empty?
  - What is expected search time then?
- Expected insertion time is essentially the same. Why?
- Expected search time on a hit
  - Expected search time for ith element added is
  - If entry was i+1st element added, expected search time is $1/(1 - i/m) = m/(m-i)$
  - Sum this for all i, divide by n, and you get $1/\alpha\ (H_m - H_{m-n})$
  - This can be bounded by $1/\alpha \ln 1/(1-\alpha)$

The End