Data Structures – LECTURE 11

Hash tables

- Motivation
- Direct-address tables
- Hash tables
- Open addressing
- Chaining
- Hash functions
- Perfect hashing

Chapter 11 in the textbook (pp 221—252).

Motivation

- Many tasks require table operations: maintain a symbol table (dictionary) with access $O(1)$ on average to the entries. The keys need not have an order relation.
- Examples:
  - Programming language keywords for compiler (static)
  - By-laws of a contract (dynamic)
  - ID numbers, customer repair orders
- We want an ADT that supports Search, Insert, and Delete in $O(1)$ on average with no order relation between elements.
- Search trees require an order relation and take $O(\lg n)$.
- Hash tables allow general keys and take $O(1)$ on average.

Hash tables: overview

- Generalization of arrays $A[0..n–1]$.
- Instead of using the key k as an index to the array A, compute the array index with a hash function $h(k)$:
  $A[k] \rightarrow A[h(k)]$
- The size of the hash table is proportional to the number of elements, and not to their range.
- The function $h(k)$ need not be a one-to-one function.
- Need a mechanism to efficiently handle collisions. Two keys k_1 and k_2 collide when $h(k_1) = h(k_2)$.

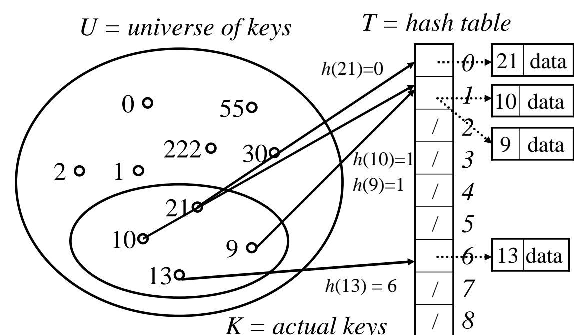
Examples

- Keywords of a programming language
  - [**for, if, then,** …] → h(**for**) = 0; h(**if**) = 1; …
  - Size of array is fixed; keyword set is static → array
- Car license plates
  - Order number is arbitrary, some numbers might not exist
  - Allocate a hash table of fixed size.
  - Hashing function: plate number modulo maximum size of hash table → h(55-080-32) = 5508032 mod 100,000
- User logins and passwords.

Formalization

- Let U be a universe of keys of size $|U|$, K an actual set of keys of size n, T a hash table of size $O(m)$, $m \leq |U|$.
- Let $h(k)$ be a hash function:
  $h(k): U \rightarrow [0..m–1]$
  that maps key values from U to indices in T.
  $h(k)$ is computed in $O(|k|) = O(1)$.
- Elements $T[i]$ in the hash table $T = [0..m–1]$ are accessed in $O(1)$ time. $T[i]$ = ***null*** is an empty entry.
- For simplicity, we assume that $U = \{0, …, N–1\}$.

Hashing: illustration



Hashing: key issues

- What are good hashing functions?
- How do we deal with collisions?
- What assumptions are necessary to guarantee $O(1)$ average time access?
  How about worst-case access time?

Main approaches:

1. Direct-address tables
2. Open-addressing
3. Chaining

Issues: good hashing function; perfect hashing.

1. Direct-address tables
טבלאות מיעון ישיר

- The hash table is an array of size m, $T[0..m–1]$.
- The number of actual keys is n is close to m, or m is reasonably small and there is sufficient storage.
- The key k is the index into T, i.e, the hash function is $h(k) = k$ (generalization: any one-to-one function).
- No collisions, access time is $O(1)$ in the worst case.
- No need to store the key itself, only the data.
- Problems:
  - Can be very wasteful in memory
  - Impractical when m is very large

Hash tables

- Use a many-to-one hash function $h(k)$ to map keys k to indices of T.
- The set of actual keys K can be much smaller than the universe of keys U, i.e., $m \ll |U|$.
- Resolve collisions, i.e., $h(k_1) = h(k_2)$ with either
  1. Open addressing
  2. Chaining

Simple uniform hashing and load factor

Simple uniform hashing assumption: each element is equally likely to hash into any of the m slots, independently of the other elements.

Definition: the **load factor** α of a hash table T with m slots is defined as $\alpha = n/m$, where n is the number of stored elements, and $0 \le \alpha \le 1$.

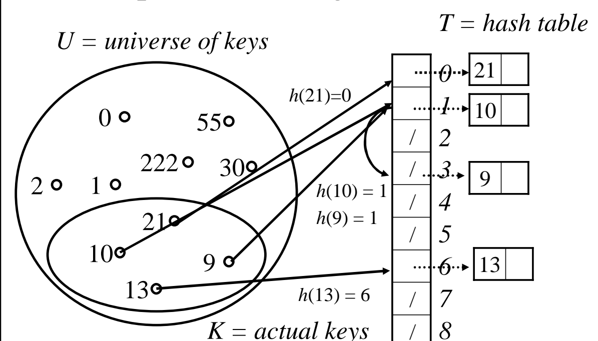
2. Collision resolution by open addressing

- All keys k_i that map into the same slot $T[h(k_i)]$ are mapped to the next available slot in the table. Looking for the next available spot is called probing.
- Slots contain the element themselves, or ***null***.
- The hash function is augmented with a probe number:

$$h(k,i): U \times [0..m–1] \rightarrow [0..m–1]$$

Probe sequence: $h(k,0)$, $h(k,1)$…

Open addressing: illustration



Open addressing operations

- Insert: probe the hash table until an empty slot is found. The sequence of probes depends on the key. If there is no empty slot after m probes, the table is full.
- Search: probe the same sequence of slots as insert and stops either when the key is found (success) or when an empty slot is reached (fail).
- Delete: cannot just delete the key! Instead, mark the slot as "deleted" so that probing can go over it.

Complexity: length of the probing sequence.

Probing strategies

Three probing techniques:

1. Linear probing
2. Quadratic probing
3. Double hashing

- None of them fulfils the *uniform hashing* assumption: each key is equally likely to have any of the m! permutations of $(0..m-1)$ as its probe sequence.
- However, they approximate it.

Linear probing

The hash function is:

$$h(k, i) = (h'(k) + i) \bmod m$$

where $h'(k)$ is an auxiliary probe-independent hash function.

Given a key k, the probing sequence is:

$T[h'(k)], T[h'(k)+1], \dots T[m-1], T[0], T[1] \dots T[h'(k)-1]$

Problem: *primary clustering*. Long runs of occupied slots build up, because an empty slot preceded by i full slots gets filled with probability $(i+1)/m$ → increases average search time. Generates m distinct probing sequences.

Quadratic probing

The hash function is:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where c_1 and c_2 are constants $\neq 0$ and $h'(k)$ is an auxiliary hash function.

In contrast with linear probing, the probed positions are offset by amounts that depend ina quadratic manner on the probe number i.

Generates m probing sequences. Suffers from *secondary clustering*: keys that hash to the same initial slot will probe the same alternative cells.

Double hashing

The hash function is:

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

where $h_1(k)$ and $h_2(k)$ are two hash functions.

The first probe is to $T[h_1(k)]$. Successive probes are offset from the previous position by $h_2(k) \bmod m$.

The value $h_2(k)$ must be prime to the hash table of size m so the entire table is searched.

Example of a choice of functions:

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$ with $m' < m$

Generates $\Theta(m^2)$ probe sequences.

Open addressing: analysis

Theorem: Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes, assuming uniform hashing is:

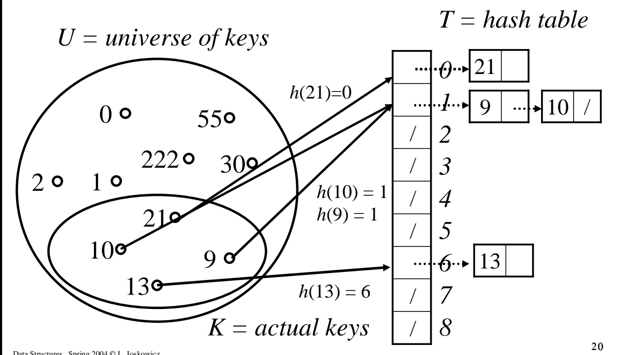
- at most $1/(1-\alpha)$ in an unsuccessful search
- at most $1/\alpha \ln 1/(1-\alpha)$ in a successful search

- When α is constant, the search time is $O(1)$ and we have a gives a bound on the number of probes that will happen.

3. Collision resolution by chaining
שרשור

- All keys k_i that map into the same slot $T[h(k_i)]$ are placed in a linked list L_j , $j = h(k_i)$.
- Slots contain pointers to the linked lists L_j
- Insert: new keys are inserted at the head of the list L_j → worst-case time $O(1)$.
- Search/Delete: find/delete the element with key k in linked list L_j → worst-case time proportional to length of longest list.

Chaining: illustration



Chaining: simple uniform hashing

Theorem: In a hash table with chaining, under the assumption of simple uniform hashing, both successful and unsuccessful searches take expected time $\Theta(1+\alpha)$ on the average, where α is the hash table load factor.

The simple uniform hashing assumption

- Is the simple uniform hashing assumption reasonable?
- Suppose we pick a function h. Then it cannot be that this h distributes the keys k approximately uniformly over the table for ALL possible sets of keys in the universe U., i.e, h cannot be a good hash function for all possible key sets K!
- The reason is as follows. We know that h maps the universe U, which is huge, into m possible indices. Consider the set S of elements k and the index i such that $h(k) = i$. There must exist one such index i for which more than $1/m$ of the possible keys go to. So there is one index to which we direct $|U|/m$ possible keys! Since $|U| >> m$, this is a larger number!
- Suppose we are now given n keys from S. They will all go to the same slot, yielding the worst case behavior!

Good hash functions

- The performance of hashing critically depends on the properties of the hash function and the actual key set patterns.
- A hash function that satisfies the simple uniform hashing assumption is a good one!
- However, it is typically not possible to check if the assumption holds, since we usually do not know the probability distribution according to which the keys are drawn, and keys may not be drawn independently.
- Two approaches:
  - Look for functions that do well "most of the time" → heuristics
  - Choose hash function randomly for provably good performance

Heuristic hashing functions

- The hash function should work well on most actual key sets, not on ones that were maliciously prepared to make the function perform badly.
- The actual key set K is usually not random, and has simple patterns, such as keys starting with the same few bits, or keys that are multiples of some integer.
- The goal is to find a hash function which divides the key universe U in a way which looks random to such average actual key sets K. The worst-case pattern for the selected hash function should be a rare one.
- Heuristic: a strategy or rule-of-thumb that works *most of the time.*

The division method

- Let $U = N = \{0,1,2, \ldots\}$, the set of natural numbers.
- Map a key k into one of m slots by taking the reminder of k divided by m:

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

- For this to work properly, avoid choosing m which is a power of 2 ($m = 2^p$) since this is like selecting the lowest p bits, which ignores useful discriminating information.
- Heuristic: pick m to be a prime number far from a power of two.

Example: the division method

- Suppose $|U| = n =$ 2000 and we can tolerate up to 3 collisions per key.
- What should be the size m of the hash table?
- We have that *floor*(2000/3) = 666; a prime number close to it and not a power of two is 701.
- The hash function is thus:

$$h(k) = k \bmod 701$$

- The keys 0, 701, and 1402 will all map to 0.

The multiplication method

- Map a key k into one of m slots by first multiplying it by a constant a in the range $0 < a < 1$, extracting the fractional part of ka, and then taking the integer part of the result multiplied by m:

$$h(k) = \lfloor m(ka - \lfloor ka \rfloor) \rfloor, \quad 0 < a < 1$$

- This method is less sensitive to the values of m because the "random" behavior comes from the fact that most actual key sets have no correlation with a.
- Heuristic: pick m to be a power of 2 and a to be close to the golden ratio: $a = (\sqrt{5} - 1)/2 = 0.6180...$

Universal hashing

- Idea: choose the hash function *randomly* in a way that is independent of the keys.
- Yields a *provably good performance on average.*
- It guarantees that no single input will always have the worst-case behavior (as for QuickSort).
- Issue: what should be the set of hash functions from which to choose? There are infinitely many functions!
- Choose from a *finite* collection of *universal hash functions*.

Universal hashing (1)

- Motivation: we want the simple uniform hashing assumption to hold, so that on average, the keys will be hashed uniformly.
- Properties of simple uniform hashing:
  - For any two keys k_1 and k_2, and any two slots y_1 and y_2, the chance that $h(k_1) = y_1$ and $h(k_2) = y_2$ is exactly $1/m^2$.
  - For two keys k_1 and k_2, the chance that they collide, that is $h(k_1) = h(k_2)$ is exactly $1/m$.
- We want a family of hash functions ***H*** that has the same chance of collision as simple uniform hashing.

Universal hashing (2)

Definition: Let ***H*** be a finite collection of hash functions that map a given universe U of keys into the range $\{0,1,\ldots,m-1\}$.

H is said to be *universal* if for every pair of distinct keys k_1 and k_2 in U, the number of hash functions h in ***H***, for which $h(k_1) = h(k_2)$ is at most $|\boldsymbol{H}|/m$.

- In other words, the chance of collision between distinct keys k_1 and k_2 is no more than the chance $1/m$ of a collision if $h(k_1)$ and $h(k_2)$ were randomly and independently chosen from the set $\{0,1,...,m-1\}$.

Expected list length in hash table

Theorem: Let h be a hash function chosen from a universal collection of hash functions and used to hash n keys into a table T of size m, using chaining resolution. Let $\alpha = n/m$ be the load factor.

- if k is not in the table, the expected length of the list that k is hashed to is at most α.
- if k is in the table, the expected length of the list that k is hashed to is at most $1+\alpha$.

Proof outline

- k is not in the table: $1/m$ of the keys will be hashed to that list, and so there will be on average α keys in the list.
- k is in the table: out of the $n-1$ remaining keys, $1/m$ on average will go to the same slot. So on average we have at most $(1 + (n-1)/m) < (1 + n/m) = 1+\alpha$.

Complexity of operation sequences

Corollary: using universal hashing and collision resolution by chaining in a table with m slots, it takes expected time $\Theta(n)$ to handle any sequence of n insert, search, and delete operations containing $O(m)$ insert operations.

Proof: because the number of insertions $n = O(m)$, the load factor $\alpha = O(1)$. So by the previous theorem, each operation takes $O(1)$ time on average and $\Theta(n)$ total.

Construction of universal classes (1)

- Choose a prime number $p > m$ and larger than the range of the actual keys K. Let $\boldsymbol{Z}_p$ denote the set $\{0,\ldots p-1\}$, and let a and b be two numbers from $\boldsymbol{Z}_p$.
- Consider the function:

$$h_{a,b}(k) = (ak + b) \bmod p$$

- The collection of all such hash functions is:

$$\boldsymbol{H}_{p,m} = \{h_{a,b} \mid a,b \in \boldsymbol{Z}_p \text{ and } a \neq 0\}$$

- To choose a random hash function, we pick a,b randomly from $\boldsymbol{Z}_p$.

Construction of universal classes (2)

- Claim: Pick any two different keys, k_1 and k_2. For any two elements x_1 and x_2 in $\boldsymbol{Z}_p$, the chance that k_1 will be hashed to x_1 and k_2 to x_2 is exactly $1/p^2$.
- Proof: we can write two equations with variables a and b:

$$ak_1 + b = x_1 \bmod p$$
$$ak_2 + b = x_2 \bmod p$$

These equations always have a unique solution when p is prime! For any two x_1 and x_2, there exists a hash function with parameters a and b which maps k_1 to x_1 and k_2 to x_2.

- Thus, the chance of picking that function is exactly the chance of picking the correct a and b, which is exactly $1/p^2$.

Construction of universal classes (3)

Since the range of keys can be very large, we correct the hash function to reduce it to m keys by taking an additional modulo m:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

The family

$$\boldsymbol{H}_{p,m} = \{h_{a,b}: = a, b \text{ in } \boldsymbol{Z}_p\}$$

is then a universal family of hash functions.

Universal hashing: summary

- Universal Hashing gives $O(1)$ performance on average for any set of actual keys → even if there are "crazy" patterns in the key set, we will manage to hash them nicely on average.
- The chance that the performance is really bad, (say a factor of 100 times the average) is really small (say, a chance of 1/100).
- However, if the set is dynamic, we do not know in advance whether the function will be good or not….

Perfect hashing (1)

- Universal hashing guarantees $O(1)$ average performance for any key set.
- Can we do better? Yes, in some cases!
- Perfect hashing guarantees $O(1)$ worst-case performance for a static key set, in which once the keys are stored, they never change.
- Examples of static key sets: reserved words in a programming language, file names on a CD-ROM.

Perfect hashing (2)

Idea: Use a two-level hashing scheme with universal hashing at each level.

Level 1: Hashing with chaining. The n keys of K are hashed to the m slots of T using hash function $h(k)$ chosen from a universal class.

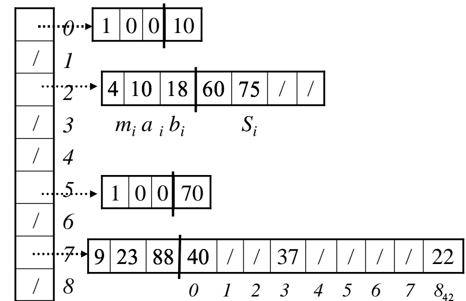
Level 2: Instead of making a list of keys hashing into slot j, use a secondary hash table S_j with associated hash function $h_j(k)$. Choose $h_j(k)$ to ensure that no collisions occur, and the size of S_j as the square of the number n_j of keys hashing to slot j: $|S_j| = n_j^2$.

Example: perfect hashing

Primary hash function: $h(k) = ((3k + 42) \bmod 101) \bmod 9$

Secondary has function: $h_i(k) = ((a_i k + b_i) \bmod p) \bmod m_i$

$K = \{10,22,37, 40,60,70, 75\}$



Perfect hashing: analysis

Theorem: If we store n keys in a (primary) hash table of size $m = n^2$ using a hash function $h(k)$ randomly chosen from a universal class of hash functions, then the probability of a collision is < ½.

Proof: There are $n(n-1)/2$ pairs of keys that may collide, and each pair collides with probability $1/m$ when h is chosen from a universal class of hash functions. When $m = n^2$ we have:

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

Therefore, it is more likely NOT to have a collision!

Perfect hashing: analysis

- When n is large, a hash table of size $m = n^2$ is excessive.
- To reduce the overall storage needs we adopt the following scheme:
  - Level 1: T is of size $m = n$
  - Level 2: S_j is of size $m_j = n_j^2$
- Since we ensure that there are no collisions in the secondary hash tables by picking the hash functions appropriately, the worst-case access time is constant.
- What is the expected combined size of all hash tables?

Perfect hashing: analysis space

- The size of the primary hash table is $O(n)$.
- The expected size of all the secondary hash tables is:

$$\sum_{j=1}^{m} n_j^2 = \sum_{j=1}^{m} \left(n_j + 2\frac{n_j(n_j-1)}{2}\right) =$$

$$= n + 2\sum_{j=1}^{m} \left(\frac{n_j(n_j-1)}{2}\right)$$

- The term in the sum is exactly the total number of collisions!
- On average it is $1/m$ times the number of pairs. Since $m = n$, it is at most $n/2$.
- Therefore, the expected total space for the secondary hash tables is less than $2n$.

Summary

- Hashing generalizes the array ADT.
- It achieves constant time access and linear storage for dynamic key sets.
- Hashing collisions are resolved by *open addressing* or *chaining*.
- Universal hashing functions guarantee expected average access time.
- Perfect hashing achieves worst case constant access time for static key sets.
- Hashing is NOT good for order queries (maximum, successor) since it has no key order relation.