Hash Tables and Hash Function

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 3.4), Prof. Janicki's course slides

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`

Q: Can we do better?

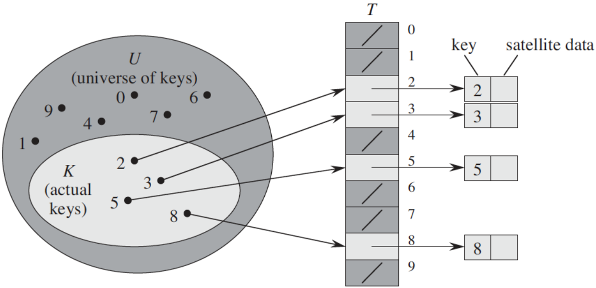
A: Yes, but with different access to the data.

Hash table: Introduction

- **Hash Table:** is a data structure used to implement Dynamic Sets/Symbol Table.
- Operations: FIND(x)/SEARCH(x), INSERT(x), DELETE(x)
- We want O(1), at least on average, for each operation!
- Typically used when the number of keys stored is far less than all the possible keys.
- A hash table generalizes the simpler notion of an ordinary array.

Array/Direct-address table (DAT)

- Universe $U = \{0, 1, \ldots, m-1\}$ of keys is small. Then simply use an array of direct address table $T[0..m-1]$.
- Every slot corresponds to a unique key in the universe U.
- If a key k absent in actual key set $K \Rightarrow T[k] = $ NIL (dark grey slots).



Array/DAT Vs. Hash table

- Directly addressing into an ordinary array enables us to search, insert and delete arbitrary elements in an array in O(1) time.
- Impractical/impossible if the universe $|U|$ is large.
- Wasted Memory: If the set of keys actually stored $|K| << |U|$.

- For above issues - use hash tables!
- Hash table typically uses an array of size proportional to the number of keys actually stored.
- Hash table – Instead of using the key as an array index directly, the array index is computed from the key using a hash function.

Hash table I

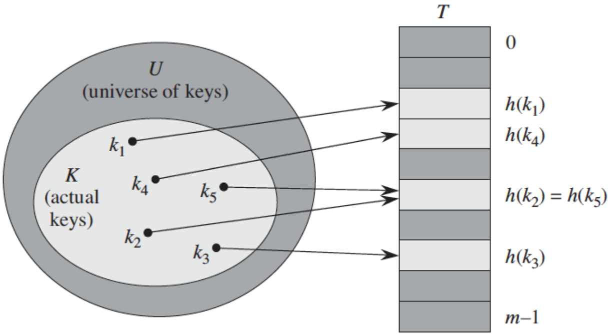
- With direct addressing, an element with key k is stored in slot k.
- With hashing, this element is stored in slot $h(k)$, where h is the **hash function**. h maps the universe U of keys into the slots of a **hash table** $T[0..m-1]$

$$h : U \rightarrow \{0, 1, \ldots, m-1\}, \ \texttt{where} \ m << |U|$$

- We say that an element with key k **hashes** to slot $h(k)$; we also say that $h(k)$ is the **hash value** of key k.
- The hash function reduces array size from $|U|$ to m = actually number of keys stored.

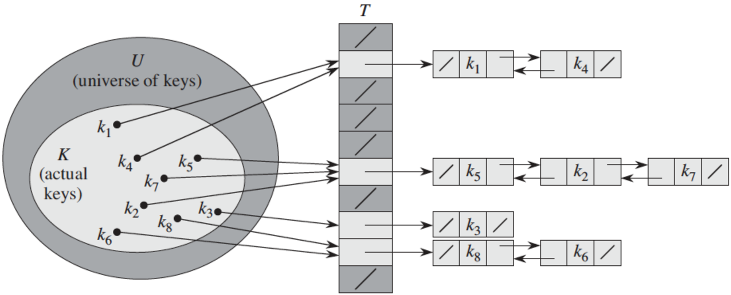
Hash table II

- Problem: Since $m << |U|$, two keys may hash to the same slot; that is, for $k_1, k_2 \neq k_1 \in U, h(k_1) = h(k_2)$. This is called **collision**.

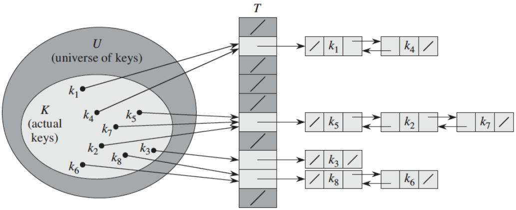


Hash table: Collision resolution by chaining I

- In **chaining**, we place all the elements that hash to the same slot into the same linked list.
- Slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains NIL.
- For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.



Hash table: Collision resolution by chaining II



- INSERT(k): insert k at the head of list $T(h(k))$ – time complexity O(1) (we assume that k is not in list).
- DELETE(k): delete k from the list $T(h(k))$ – time complexity, worst case O(m), where $m = |T|$
- FIND(k): search for an element k in the list $T(h(k))$ – time complexity, worst case O(m)

It is much better on average for 'good' hash functions h!

Hash Function - I

- Good Hash Function: The probability that $h(k_1) = h(k_2)$ for $k_1 \neq k_2$ is "small".
- Most hash functions assume that $U = \{0, 1, 2, \ldots\}$.
- Simple Uniform Hashing: Each element of U is equally likely to hash to any of the n slots, independently of where any other element of U has hashed to.
- Problem: One rarely knows the probability distribution according to which elements of U are drawn.
- In practice, we can often employ heuristic techniques to create a hash function that performs well.
- Qualitative information about the distribution of keys may be useful in this design process.
- A good hash function would minimize the chance that keys that are "close" (in some sense) hash to the same slot.

Hash Function - II

- Suppose $U = \{0, 1, \ldots, n-1\}$, $|T| = m$ and all elements of U are uniformly distributed. The function:

$$h(k) = \left\lfloor \frac{k}{n} m \right\rfloor$$

satisfies the condition of simple uniform hashing, but it is not considered a good hash function – as keys that are close hash to the same slot.

'Good' Hash Functions - Division method I

In the **Division method**, we map a key k into one of m slots by taking the remainder of k divided by m; that is, the hash function is

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

For example, if the hash table (T) has size $m = 12$, and the key is $k = 100$, then $h(k) = 4$.

- Hashing by division is quite fast – requires only a single division operation.
- Not all choices of m are good, m should satisfy $m \neq 2^p$, a prime number not to close to 2^p is a good choice.
- If $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k. Unless we know that all low-order p-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key.

'Good' Hash Functions - Division method II

In the **division method**, we map a key k into one of m slots by taking the remainder of k divided by m; that is, the hash function is

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

A prime number not to close too 2^p is a good choice.

- For example, suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2000$ character strings, where a character has 8 bits. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$. We could choose $m = 701$ because it is a prime near $2000/3$ but not near any power of 2.
- Treating each key k as an integer, our hash function would be

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

'Good' Hash Functions - Multiplication method II

The **multiplication method** for creating hash functions operates in two steps.

- We multiply the key k by a constant A in the range $0 < A < 1$, and extract the fraction part kA. which is equal to $kA \mod 1$.
- Then, we multiply $kA \mod 1$ by m and take the floor.
- Hence the hash function is:

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- An advantage of this method is that the value of m is not critical.
- We typically choose it to be a power of 2 ($m = 2^p$ for some integer p), so an implementation of $h(k)$ is easy.
- This method works with any A, it works better with some values than with others.
- The optimal choice depends on the characteristics of the data being hashed.
- Knuth suggests that $A \approx (\sqrt{5} - 1)/2 = 0.6180339887$ works well.

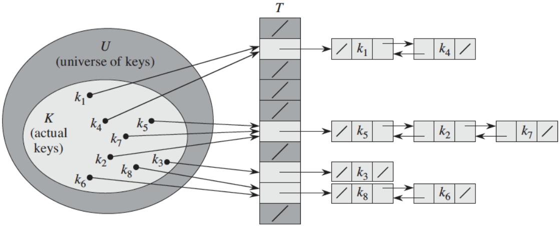
'Good' Hash Functions - Universal hashing

- For any fixed hash function and any n there is a universe that one can choose n keys that all hash to the same slot, yielding an average retrieval time of $\Theta(n)$.
- Any fixed hash function is vulnerable to such terrible worst-case behaviour; the only effective way to improve the situation is to choose hash function randomly in a way that is independent of the keys that are actually going to be stored.
- This approach, called **universal hashing**, can yield provably good performance on average, no matter what keys are chosen.
- This is a part of randomized algorithms and it will not be elaborated in this course.

Analysis of Hashing with Chaining

Given a hash table T with m slots that stores n elements, we define the **load factor** $\alpha = n/m$, that is the average number of elements stored in a chain.

Theorem: In a hash table in which collisions are resolved by chaining, both a successful and unsuccessful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.



Open Addressing (see demo for linear probing)

- In **open addressing**, all elements are stored in the hash table itself (no chaining!).
- **Linear Probing** Let $h' : U \mapsto \{0, 1, \ldots, m-1\}$ be some given 'good' hash function (called auxiliary hash function). The method of linear probing uses the hash function:

$$h(k, i) = (h'(k) + i) \mod m, \ \texttt{for} \ i = 0, 1, \ldots, m-1$$

- Given key k, the first slot probed is $T[h'(k)]$, if it is occupied we probe $T[h'(k) + 1]$, if occupied we probe $T[h'(k) + 2]$, and so on.
- Open addressing is easy to implement, but it suffers from primary clustering problem – Long runs of occupied slots build up, increasing the average search time.
- Long runs of occupied slots tend to get longer and avg. search time rises.

Quadratic Probing

Quadratic Probing uses a hash function of the form

$$h(k,i) = (h_0(k) + c_1 i + c_2 i^2) \mod m,$$

where h_0 is an auxiliary 'good' hash function, and $c_1 \neq 0$ and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$.

- The initial probe is $T[h_0(k)]$, later positions probed are at an offset by amounts that depend in a quadratic manner on the number i.
- This method works much better than linear probing, and clustering can be avoided.
- However, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.
- This property leads to a milder form of clustering, called secondary clustering.

Double Hashing - I

Double hashing is one of the best method available for open addressing.

Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m,$$

where h_1 and h_2 are auxiliary 'good' hash functions, and i is an iterative index.

- The initial probe is into the table T is at $T[h_1(k)]$ successive probe positions (for $i = 1, 2, \ldots$) are at an offset from previous positions by the amount $h_2(k)$ modulo m.
- Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.
- The performance of double hashing is very close to the performance of the "ideal" scheme of uniform hashing.

Double Hashing - II

- The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched.
- One way to ensure this condition is to let m be a power of 2, and to design h_2 so that it always produces an odd number.
- Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m.
- For example, we could choose m prime and let:
  $h_1(k) = k \mod m$
  $h_2(k) = 1 + (k \mod m')$
  where m' is chosen to be slightly less than m (say, $m - 1$).