Motivation

To search for an entry in a table:

	Time complexity
Linear search (if the data is not sorted)	$O(N)$

	name	score
0	Parker	323
1	Davis	434
2	Harris	314
3	Corea	323
4	Hancock	416
5	Brecker	378
6	`empty`	
$\vdots$	$\vdots$	
$N-1$	Mark	541

Motivation

To search for an entry in a table:

	Time complexity
Linear search (if the data is not sorted)	$O(N)$
Binary search (if the data is sorted)	$O(\log N)$
Sorting	$O(N \log N)$

	name	score
0	Brecker	378
1	Corea	323
2	Davis	434
3	Hancock	416
4	Harris	314
5	Mark	541
6	Parker	323
⋮	⋮	
$N-1$	`empty`	

Can we improve the situation to $O(1)$?

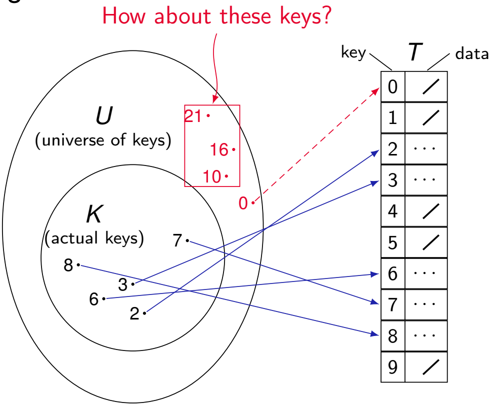
CPT108 Data Structures and Algorithms

Lecture 18

Hashtables

Direct Addressing

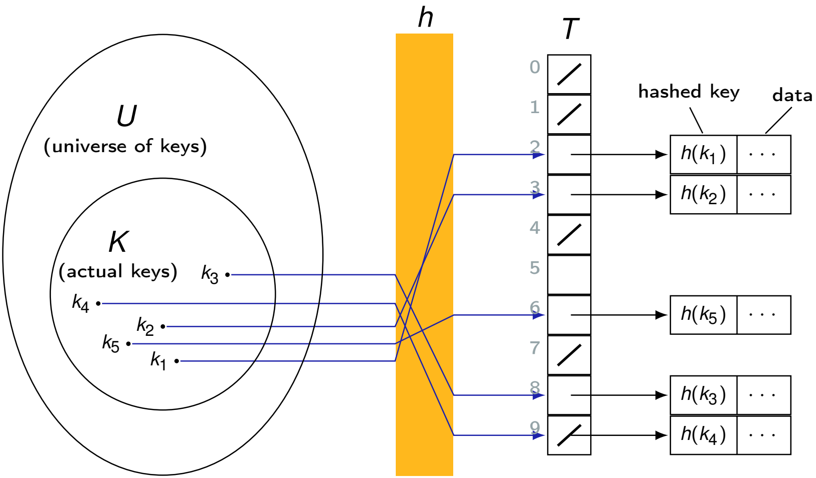
- A map is a data structure that supports the use of a key (as "*address*") that help locate an entry in operations such as searchings.



Direct Addressing

Problems (cont.)

- If the universe U is large or infinite, storing T of size $|U|$ may be impractical
- Waste too much space if the universe is too large compared with the actual number of elements to be stored.
  - E.g., suppose the student IDs are 8-digit integers
    - So the universe is $\underline{10^8}$,
    - But we only have about <u>8000 students</u>

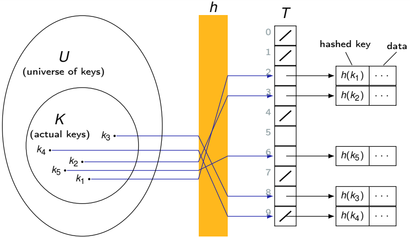

h
T
U
(universe of keys)
K
(actual keys)
k_3
k_4
k_2
k_5
k_1
0
1
2
3
4
5
6
7
8
9
hashed key
data
$h(k_1)$
$h(k_2)$
$h(k_5)$
$h(k_3)$
$h(k_4)$
· · ·

Hashing:

$\{k_1, \ldots, k_n\}$ possible keys

$$\big\downarrow \textit{hash}$$

$T[0, \ldots, m-1]$ hash table



Hash table:

- Also known as *hash map*, an abstract data type (ADT) that implements associative array (of some fixed size).
- Supports finds, insertions, and deletions of any *named item*.
- Allows operations to be executed in constant average time ($O(1)$)
- A *slot* (or *bucket*) number is calculated by a *hash function*, h, that
  - takes a variable-size input k, and
  - return a fixed-size `string` (or `int` in $[0, \ldots, m-1]$), $h(k)$, which is called the hash value of k, and m is the capacity of the table.
- Usually, $m << N$

Hash table

Hash table:

- It provides a fast way to maintain a set of keys or map keys to values, even when the keys are *objects*, like strings, while “relationship” between elements are of less, or not, important.
- *However*...
  - Operations that requires any ordering information among elements are *not* supported
    - `findMin` and `findMax`
    - Successor and predecessor
    - Report data within a given range
    - List out data in order
    - ...

Hash table

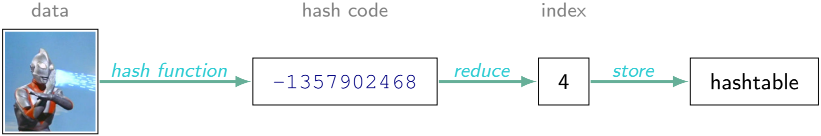
Applications

- Compilers use hash table (symbol table) to keep track of declared variables
- On-line spell checkers. After pre-hashing the entire dictionary, one can check each word in constant time and print out the misspelled word in order of their appearance in the document.
- ...

Hash function

Hashtables in practice:

1. Data is *converted* into a hash code through the use of a *hash function*
2. The hash code is then *reduced* to a valid *index*
3. Data is then *stored* in a slot (or bucket) corresponding to that index



Hash function (cont.)

- To recap, with hashing, we store an element with a key k into $T[h(k)]$, where $h : U \mapsto T$ is a hash function.
- Collisions appears when two keys are hashed to the same slot
- Can we ensure that any two distinct keys get different cells?
  - No! This is unavoidable as we assume $m << N$, where m is the size of the hash table and N is the number of keys

Hash function (cont.)

- What makes a good hash function?
  - A good hash function should satisfies the assumption of *independent uniform hashing*
    - Each key is equally likely to hash to any of the m slots, independently of where any other keys have hashed to
    - Unfortunately, there is, in general, no way to check this condition since we rarely know the probability distribution from which the keys are drawn, or whether they are drawn independently.
    - Qualitative information about the distribution of keys may be useful in the design process.
    - I.e., when two closely related symbols, such as `pt` and `pts`, often appear in the same dataset, a good hash function would minimize the chance that such variants hash to the same slot.

Hash function (cont.)

Task 1: How to design a good hash function that

- is *fast* to compute,
- *spread* the keys *evenly* in the table, and
- can *minimize* the number of collisions?

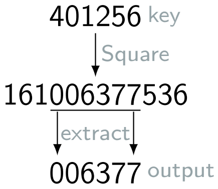
Task 2: How to *resolve* the collisions when they occur?

Design hash function: Integer keys

- The Division method: $h(k) = k \mod m$
  - Simple and reasonable strategy
  - Requires only a single division operation, which is quite fast
  - E.g., when $k = 100$ and $m = 12$, $h(k) = 4$
  - However, certain values of m should be avoided
    - E.g., if $m = 2^p$, then $h(k)$ is just the p lowest bits of k; i.e., the hash function does not depend on all the bits
    - Similarly problem will appear if the keys are a decimal numbers and setting m to be a power of 10.
  - In general, it is good to set m to be a *prime number* that is not too close to exact powers of 2.

Design hash function: Integer keys (cont.)

- The Mid-square method
  - Squaring the key value first,
  - takes out the middle r bits of the results, giving a value in the range 0 to $2^r - 1$.
- This works well because most or all bits of the key contribute to the result



Example of mid-square method using a 6-digit key and with the middle 6 digits as output.

Design hash function: String-type keys

- Most hash function assume that the keys are *natural numbers*
- If keys are not natural numbers, a way must be found to interpret them as natural numbers
- Note:
  - Letters and digits fall in range `0101` and `0172` octal
  - i.e., all useful information is in lowest 6 bits

** Must be careful to cover range from 0 through the capacity of the hash table, *m*

Design hash function: String-type keys (cont.)

Method 1 Adding up the ASCII (or unicode) values of the characters in the string

- E.g., if $m =$ 'test' (ASCII={$116, 101, 115, 116$}) and hash function $=$ *modulo* 11, then

$$\begin{aligned} \text{hash value} &= (116 + 101 + 115 + 116) \mod 11 \\ &= 448 \mod 11 \\ &= 8 \end{aligned}$$

- However, different permutations of the same set of characters, e.g., "eat", "ate", and "tea", would have the same hash value.

Design hash function: String-type keys (cont.)

Method 2 Adding up the ASCII (or unicode) values of the characters in the string

- Computes:

$$\Sigma_{i=0}^{L-1} k[L-i-1] \cdot 37^i$$

where L is the size of k, $k[i]$ is the i^{th} character of the key k.

- Or another way to look at the formula is:

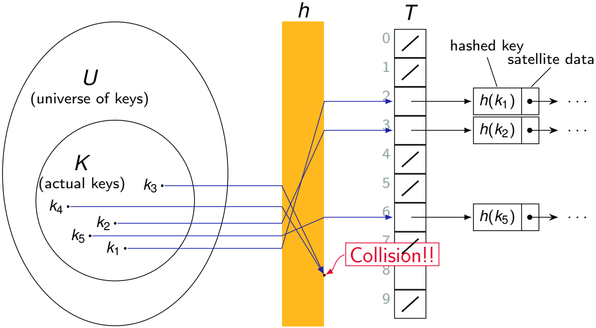
$$h = \Pi_{i=0}^{L-1} h_i \mod m$$

$$\text{where } \; h_i = \begin{cases} k[L-1] & \text{if } i = 0 \\ 37 \cdot h_{i-1} + k[L-i-1] & \text{otherwise} \end{cases}$$

- This method is better in a sense that it involves all characters in the key and be expected to distribute well
- However, it will take a bit longer to compute if the keys are very long

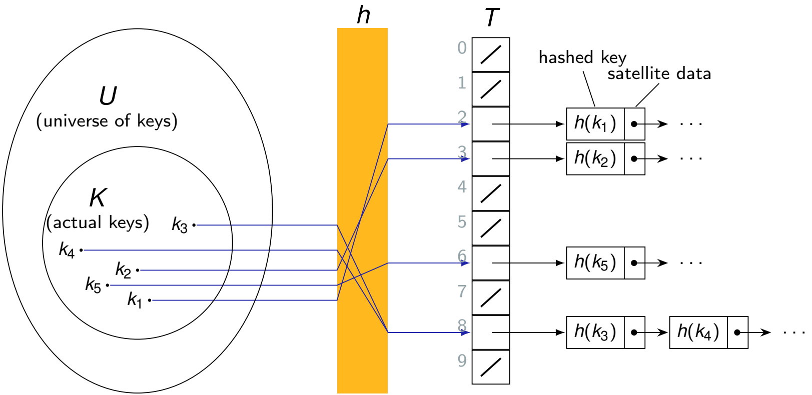
How to handle collision in hash table?

- Collision appears when two (or more) keys are hashed into the same slot
- A suitable hash function can be choose to avoid or minimize the number of collision



How to handle collision in hash table? (cont.)

- Separate chaining (or simply, chaining)
  - Each array slot is a linked list (or search list)

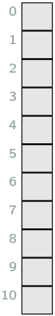


Collision Handling: Separate chaining

Separate chaining: Exercise

Consider a hashtable that uses the *separate chaining* technique with the following hash function $f(x) = (5x + 4)\%11$. (The hashtable is of size 11.)

If we insert the value 3, 9, 2, 1, 14, 6 and 25 into the table, in that order, show where these values would end up in the table?



How to handle collision in hash table? (cont.)

- Separate chaining (cont.)
  - If the hash function works well, the number of keys in each linked list will be a small constant.
  - Each operation (search, insertion, and deletion) can be done in constant time
  - Advantages
    - Never get “full”
    - Deletion is easy
  - Disadvantages
    - Elements are stored *outside* of the hash table itself
    - Memory allocation in linked list manipulation may slow down the program.

How to handle collision in hash table? (cont.)

- Open addressing
  - Instead of following pointers, compute the sequence of slots to be examined
  - *Relocate* the key k to be inserted if it collides with an existing key
  - That is:
    - When a new element is to be inserted into the table, it is placed in its "first-choice" location if possible
    - If that location is already occupied, the new element is placed in its "second-choice" location
    - This process continues until an empty slot is found in which to place the new element

How to handle collision in hash table? (cont.)

- The hash function is in a form:

$$h_i(k) = (\underbrace{h(k)}_{probe} + \underbrace{f(i)}_{offset}) \mod m,$$

with $f(0) = 0$

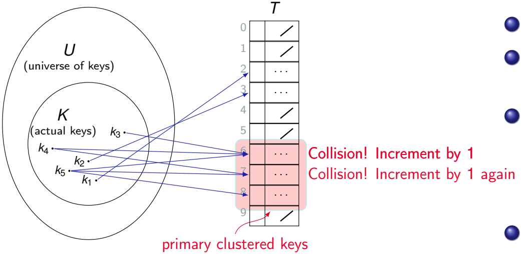
  - f : collision resolution strategy or relocation scheme

How to handle collision in hash table? (cont.)

- Open addressing
  - Two issues arise
    - What is the relocation scheme?
    - How to search for k later?
  - Commonly used approaches include: *linear probing*, *quadratic probing*, and *double hashing*

Collision Handling: Open addressing (cont.)

- Linear probing
  - Put $f(i) = i$
  - I.e., $h_i(k) = (h(k) + i) \mod N$, for $i = 1, 2, 3, \ldots$
  - Slot are probed sequentially (with wrap-around), i.e., increment the hash value by a constant 1 until a free slot is found



If $h(k_3) = h(k_4) = h(k_5) = 6$

- Simple to implement
- Cell are probed sequentially (with wrap-around)
- If we cannot find an empty entry to put k, it means the table is full and we should report an error.
- But may leads to *primary clustering*

Collision Handling: Open addressing (cont.)

Linear probing: Exercise

Consider a hashtable that uses the *linear probing* technique with the following hash function $f(x) = (5x + 4)\%11$. (The hashtable is of size 11.)

If we insert the value 3, 9, 2, 1, 14, 6 and 25 into the table, in that order, show where these values would end up in the table?

index	value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Collision Handling: Open addressing (cont.)

- Primary clustering
  - A cluster is a block of contiguously occupied table entries.
  - On the average, when we insert a new key k, we may hit the middle of a cluster.
  - Therefore, the time to insert k would be proportional to half the size of a cluster.
  - I.e., the larger the cluster, the longer time it needs to find an empty slot, and the slower the performance

Collision Handling: Open addressing (cont.)

- In Linear probing (cont.)
  - Once $h(k)$ falls into a cluster, this cluster will definitely grow in size by one, which may worsen the performance of insertion in the future.
  - If two clusters are only separated by one slot, then inserting one key into a cluster can merge the two clusters together. Thus, the cluster size can increase drastically by a single insertion, meaning that the performance of insertion can deteriorate drastically after a single insertion.
  - Large cluster are easy targets for collision.

Collision Handling: Open addressing (cont.)

- Quadratic probing
  - Put $f(i) = c_1 * i + c_2 * i^2$, where $c_2 \neq 0$
  - I.e., $h_i(k) = (h(k) + c_1 * i + c_2 * i^2) \mod N$, for $i = 1, 2, 3, \ldots$
  - E.g., if $h_i(k) = (h(k) + 2i + i^2) \mod m$, then the probe sequence will be $h(k)$, $h(k) + 3$, $h(k) + 8$, $h(k) + 15$, etc.
  - If the table size is prime, then a new key can always be inserted if the table is at least half empty (Weiss, 2013, Thm 5.1)
  - Incurs less clustering than linear probing. However, it may leads to secondary clustering
    - keys that hash to the same home position will probe the same alternative slots
    - Simulation results suggest that it generally causes less than an extra half probe per search
    - To avoid secondary clustering, the probe sequence need to be a function of the original key value, not the home position, which leads us to the next approach

Collision Handling: Open addressing (cont.)

Quadratic probing: Exercise

Consider a hashtable that uses the *quadratic probing* technique with the following hash function $f(x) = (5x + 4)\%11$. (The hashtable is of size 11.)

If we insert the value 3, 9, 2, 1, 14, 6 and 25 into the table, in that order, show where these values would end up in the table?

index	value
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Collision Handling: Open addressing (cont.)

- Double hashing
  - To alleviate the problem of clustering, the sequence of probes for a key should be independent of its primary position
  - Put $f(i) = i * h2(k)$, where $h2(k)$ is a hash function different from the original one and $h2(k) \neq 0$
  - I.e., $h_i(k) = (h(k) + i * h2(k)) \mod N$, for $i = 1, 2, 3, \ldots$
  - Interval depends on k, which minimize repeated collision and the effects of clustering
  - For any key k, $h2(k)$ must be *relatively prime* to the table size m; otherwise, we will only be able to examine a fraction of the table entries.
    - E.g., if $h(k) = 0$ and $h2(k) = m/2$, then we can only examine the entries $T[0]$ and $T[m/2]$, and nothing else!
  - One solution is to make m prime, and choose R to be a prime smaller than m, and set:

Collision Handling: Open addressing (cont.)

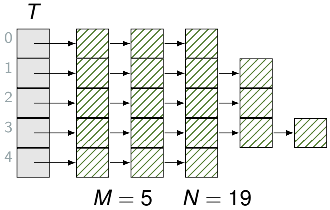
Some notes

- Elements are stored in the hash table itself
- Each table entry contains either an element of the dynamic set or NIL.
- However, *unlike* Separate Chaining
    - the hash table can "fill up"
    - Actual deletion *cannot* be performed in open addressing hash tables
        - Otherwise this will isolate entries further down the probe sequence
        - Solution: Add an extra bit to each table entry, and mark a deleted slot by storing a special value "`DELETED`" (tombstone)

Hashtable Performance

Suppose we have:

- A *fixed* number of slots, M
- An *increasing* number of elements, N
- Average list is around N/M elements



If the elements are spread out evenly,

lists are of length $Q = \dfrac{N}{M}$

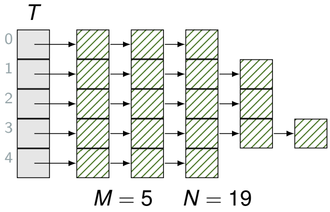
- E.g., for $M = 5$, $N = 19$, $Q = \dfrac{19}{5} = 3.8 \approx \Theta(N)$

$\Rightarrow$ results in *linear time* operations when N is large !

Hashtable Performance (cont.)

Suppose we have:

- A *increasing* number of slots, M
- An *increasing* number of elements, N
- As long as $M = \Theta(N)$, then $O(N/M) = O(1)$



Assuming elements are evenly distributed (as above), lists will be approximately N/M elements long, resulting in $\Theta(N/M)$ runtimes

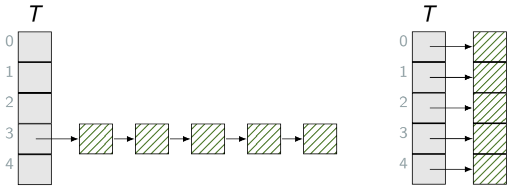
We want to ensure that $N/M = O(1)$

- However, by doubling M every time, the size of the hashtable may gets too big

Hashtable Performance (cont.)

To recap, even distribution of item is critical for good hashtable performance

The hashtables below have *load factor* of $N/M = 1$



- Which hashtable performs better?
- Why?

Hashtable: Resize

- *Resize* when load factor *exceeds* some constant
  - Allocate a larger hash table
  - *Rehash* the table
  - Delete the smaller table

Hashtable Performance (cont.)

- In elements are spread out nicely, you get $\Theta(1)$ average runtime

	`contains(x)`	`add(x)`
Binary search trees (BSTs)	$\Theta(N)$	$\Theta(\lg N)$
Hashtable (with *no* resizing)	$\Theta(N)$	$\Theta(N)$
Hashtable (with resizing[1])	$\Theta(1)$	$\Theta(1)$

[1] Assuming elements are evenly spread

Time Complexity

- $O(1)$ in most operations
- Why it appears most search mechanisms have performance at $O(N)$ or $O(\log N)$?
- Why hash table can give us best performance?
- Where does the magic come from?

Reading

- Chapter 11, Cormen (2022)
- Ch. 5, Weiss (2013)