Searching

Apurba Sarkar

IIEST Shibpur

November 13, 2019

Searching

INPUT

• sequence of numbers (database)

• a single number (query)

OUTPUT

• index of the found number or *NIL*

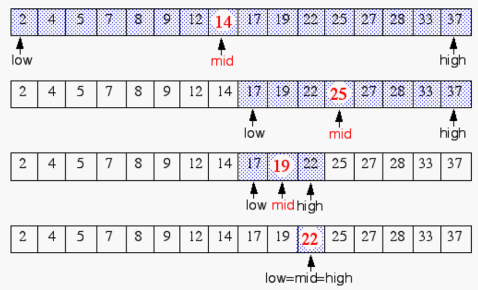
$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow j$

2 5 4 10 7; 5 $\longrightarrow$ 2

2 5 4 10 7; 9 $\longrightarrow$ *NIL*

Binary search

- Uses Divide and conquer technique.
- narrow down the search range in every steps



```
Algorithm BinarySearch(A, k, low, high)
    if low > high then return Nil
    else mid ← (low+high) / 2
         if k = A[mid] then return mid
         else if k < A[mid] then
               return BinarySearch(A, k, low, mid-1)
         else return BinarySearch(A, k, mid+1, high)
```

Running Time of Binary Search

- The range of candidate items to be searched is halved after each comparison.
- In the array-based implementation, access by rank takes $O(1)$ time, thus binary search runs in $O(\log n)$ time. Proof?

Searching in an unsorted array

- INPUT: $A[1..n]$ an array of integers, q an integer.
- OUTPUT: an index j such that $A[j] = q$. NIL, if $\forall j (1 \leq j \leq n) : A[j] \neq q$

```
j ← 1
while j ≤ n and A[j] ≠ q
  do j++
 if j ≤ n then return j
 else return NIL
```

- Worst-case running time: $O(n)$, average-case: $O(n)$
- We can't do better. This is a lower bound for the problem of searching in an arbitrary sequence.

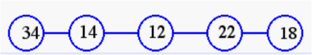
The problem

There is a large phone company, and they want to provide caller ID capability:

- given a phone number, return the caller's name
- phone numbers range from 0 to $r = 10^8 - 1$
- There are n phone numbers, $n << r$.
- want to do this as efficiently as possible

Using an unordered sequence

Unsorted sequence



- searching and removing takes $O(n)$ time
- inserting takes $O(1)$ time
- applications to log files (frequent insertions, rare searches and removals)

Using an ordered sequence

Array based ordered sequence

- searching takes $O(logn)$ time (binary search)
- inserting and removing takes $O(n)$ time
- application to look-up tables (frequent searches, rare insertions and removals)

Other Suboptimal ways

Direct addressing: an array indexed by key:

- takes $O(1)$ time,
- $O(r)$ space where r is the range of numbers (10^8)
- huge amount of wasted space

(null)	(null)	Ankur	(null)	(null)
0000-0000	0000-0000	9635-8904	0000-0000	0000-0000

Another Solution

- Can do better, with a Hash table: $O(1)$ expected time, $O(n+m)$ space, where m is table size
- Like an array, but come up with a function to map the large range into one which we can manage
  - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index
  - Insert ($9635-8904$, Ankur) into a hashed array with, say, five slots. $96358904 \; mod \; 5 \;=\; 4$

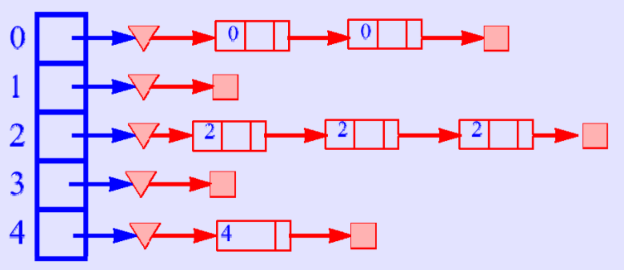
(null)	(null)	(null)	(null)	Ankur
0	1	2	3	4

An Example

- Let keys be entry nos of students in $CS302$. eg. $2018CS10110$.
- There are 100 students in the class. We create a hash table of size, say 100.
- Hash function is, say, last two digits.
- Then $2018CS10110$ goes to location 10.
- Where does $2018CS50310$ go? Also to location 10 and we have a collision!!

Collision Resolution

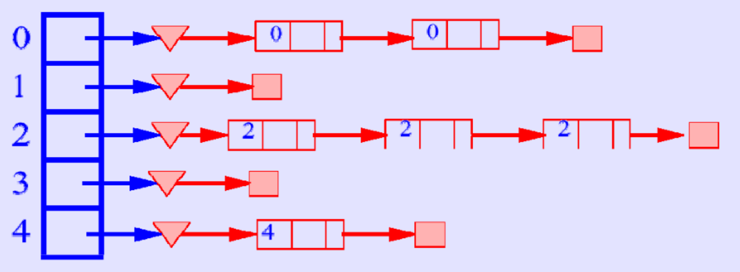
- How to deal with two keys which hash to the same spot in the array?
- Use chaining
  - Set up an array of links (a table), indexed by the keys, to lists of items with the same key



  - Most efficient (time-wise) collision resolution scheme

Collision Resolution contd.

- To find/insert/delete an element
  - using h, look up its position in table T
  - Search/insert/delete the element in the linked list of the hashed slot



Analysis of Hashing

- An element with key k is stored in slot $h(k)$ (instead of slot k without hashing)
- The hash function h maps the universe U of keys into the slots of hash table $T[0 \ldots m-1]$
  $h : U \rightarrow \{0, 1, ..., m-1\}$
- Assume time to compute $h(k)$ is $\Theta(1)$

Analysis of Hashing contd.

- A good hash function is one which distributes keys evenly amongst the slots.
- An ideal hash function would pick a slot, uniformly at random and hash the key to it.
- However, this is not a hash function since we would not know which slot to look into when searching for a key.
- For our analysis we will use this simple uniform hash function
- Given hash table T with m slots holding n elements, the load factor is defined as $\alpha = n/m$

Analysis of Hashing contd.

Unsuccessful search

- element is not in the linked list
- Simple uniform hashing yields an average list length $\alpha = n/m$
- expected number of elements to be examined α
- search time $O(1+\alpha)$ (includes computing the hash value)

Analysis of Hashing contd.

Successful search

- assume that a new element is inserted at the end of the linked list
- upon insertion of the i^{th} element, the expected length of the list is $(i-1)/m$
- in case of a successful search, the expected number of elements examined is 1 more that the number of elements examined when the sought-for element was inserted!

Analysis of Hashing contd.

- The expected number of elements examined is thus

$$
\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) \\
&= 1+\frac{1}{nm}\cdot\frac{(n-1)n}{2} \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{n}{2m}-\frac{1}{2m} \\
& 1+\frac{\alpha}{2}-\frac{1}{2m}
\end{aligned}
$$

- Considering the time for computing the hash function, we obtain

$$\Theta(2+\alpha/2-1/2m)=\Theta(1+\alpha)$$

Analysis of Hashing contd.

- Assuming the number of hash table slots is proportional to the number of elements in the table
- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$
- searching takes constant time on average
- insertion takes $O(1)$ worst-case time
- deletion takes $O(1)$ worst-case time when the lists are doubly-linked

Good Hash funtion

- the function which can be computed quickly
- it should distribute the keys uniformly over the hash table.
- good hash functions are very rare.
- birth day paradox which says
  Even in a group 35 or more students sitting in the class there is a very high probability that 2 of the students would have the same birthday.

How to deal with non-integer keys

- find some way of turning keys into integers
  - eg.remove hyphen in $9635 - 8904$ to get 96358904
  - for a string, add up ASCII values of the characters of your string
- then use standard hash function on the integers

Hash Functions

- The mapping of keys to indices of a hash table is called a hash function
- A hash function is usually the composition of two maps, a hash code map and a compression map.
  - An essential requirement of the hash function is to map equal keys to equal indices
  - A good hash function minimizes the probability of collisions
- Hash code map $keys \rightarrow integer$
- Compression map $integer \rightarrow [0 \ldots m-1]$

Popular Hash-Code Maps

- Integer cast: for numeric types with 32 bits or less, we can reinterpret the bits of the number as an int
- Component sum: for numeric types with more than 32 bits (e.g., long and double), we can add the 32-bit components.
- Why is the component-sum hash code bad for strings?

Popular Hash-Code Maps contd.

- Polynomial accumulation: for strings of a natural language, combine the character values (ASCII or Unicode) $a_0 a_1 \ldots a_{n-1}$ by viewing them as the coefficients of a polynomial:
  $a_0 + a_1 x + \ldots + x^{n-1} a_{n-1}$
- The polynomial is computed with Horner's rule, ignoring overflows, at a fixed value x:
  $a_0 + x(a_1 + x(a_2 + \ldots x(a_{n-2} + x a_{n-1})\ldots))$
- The choice $x = 33, 37, 39,$ or 41 gives at most 6 collisions on a vocabulary of $50,000$ English words.

Compression Maps

- Use the remainder: $h(k) = k\ mod\ m$, k is the key, m the size of the table.
- Need to choose m
- $m = b^e$ (bad)
  - if m is a power of $2, h(k)$ gives the e least significant bits of k.
  - all keys with the same ending go to the same place
- m is a prime (good)
  - helps ensure uniform distribution
  - primes not too close to exact powers of 2

Compression Maps contd.

Example

- hash table for $n = 2000$ character strings
- we dont mind examining 3 elements
- $m = 701$
  - a prime near $2000/3$
  - but not near any power of 2

Compression Maps contd.

- Use
  - Use $h(k) = \lfloor m(kAmod1) \rfloor$
  - k is the key, m the size of the table, and A is a constant $0 < A < 1$
- The steps involved
  - map $0 \ldots k_{max}\ into\ 0 \ldots k_{max}A$
  - take the fractional part $(mod\ 1)$
  - map it into $0 \ldots m - 1$

Choice of m and A

- value of m is not critical, typically use $m = 2^p$
- optimal choice of A depends on the characteristics of the data
- Knuth says use $A = \frac{\sqrt{5}-1}{2}$ (conjugate of the golden ratio) Fibonacci hashing

Compression Maps contd.

Multiply, Add, and Divide (MAD): $h(k) = |ak + b|\ mod\ N$

- eliminates patterns provided a is not a multiple of N
- same formula used in linear congruential (pseudo) random number generators

Universal Hashing

- For any choice of hash function, there exists a bad set of identifiers
- A malicious adversary could choose keys to be hashed such that all go into the same slot (bucket)
- Average retrieval time is $\Theta(n)$
- Solution:
  - a random hash function
  - choose hash function independently of keys!
  - create a set of hash functions H, from which h can be randomly selected

Universal Hashing

- A collection H of hash functions is universal if for any randomly chosen f from H (and two keys k and l), $Pr\{f(k) = f(l)\} \leq 1/m$

More on Collisions

- A key is mapped to an already occupied table location. What to do?
- Use a collision handling technique
  - Chaining
  - Can also use Open Addressing
    - Linear Probing
    - Double Hashing

Open Addressing

- All elements are stored in the hash table (can fill up!), i.e., $n \leq m$
- Each table entry contains either an element or null
- When searching for an element, systematically probe table slots

Open Addressing contd.

- Modify hash function to take the probe number i as the second parameter
  $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- Hash function, h, determines the sequence of slots examined for a given key
- Probe sequence for a given key k given by
  $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ - a permutation of $0, 1, \ldots, m-1$

Linear Probing

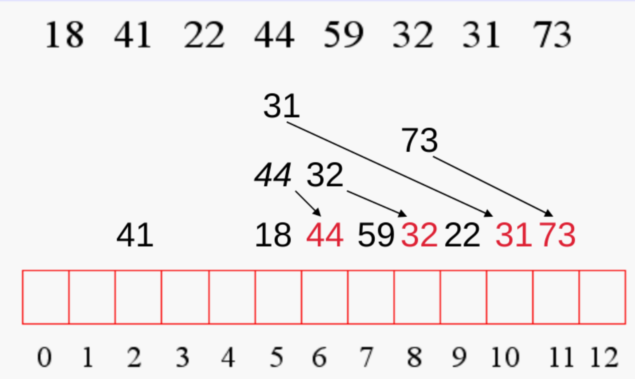
- If the current location is used, try the next table location

```
LinearProbingInsert(k)
if (table is full) error
probe = h(k)
while (table[probe] occupied)
        probe = (probe+1) mod m
table[probe] = k
```

- Uses less memory than chaining as one does not have to store all those links
- Slower than chaining since one might have to walk along the table for a long time

Linear Probing Example

- $h(k) = k \ mod \ 13$
- insert keys: $18, 41, 22, 44, 59, 32, 31, 73$



Look up in Linear Probing

- To search for a key k we go to ($k \ mod \ 13$) and continue looking at successive locations till we find k or encounter an empty location.
- Successful search: To search for 31 we go to $(31 \ mod \ 13) = 5$ and continue onto $6, 7, 8...$ till we find 31 at location 10
- Unsuccessful search: To search for 33 we go to $(33 \ mod \ 5 = 7)$ and continue till we encounter an empty location (12)

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Deletion in Linear Probing

- To delete key 32 we first search for 32.
- 32 is found in location 8. Suppose we set this location to null.
- Now if we search for 31 we will encounter a null location before seeing 31.
- Lookup procedure would declare that 31 is not present.

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Deletion in Linear Probing

- Instead of setting location 8 to null place a tombstone (a marker) there.
- When lookup encounters a tombstone it ignores it and continues with next location.
- If Insert comes across a tombstone it puts the element at that location and removes the tombstone.
- Too many tombstones degrades lookup performance.
- Rehash if there are too many tombstones.

		41			18	44	59	X	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Double Hashing

- Uses two hash functions, $h1, h2$
- $h1(k)$ is the position in the table where we first check for key k
- $h2(k)$ determines the offset we use when searching for k
- In linear probing $h2(k)$ is always 1.

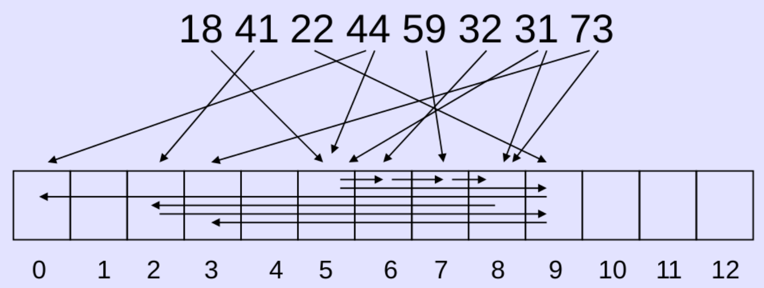
```
DoubleHashingInsert(k)
if (table is full) error
probe = h1(k); offset = h2(k)
while (table[probe] occupied)
    probe = (probe+offset) mod m
table[probe] = k
```

Double Hashing contd

- If m is prime, we will eventually examine every position in the table
- Many of the same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing

Double Hashing Example

- $h1(k) = k \ mod \ 13$
- $h2(k) = 8 - (k \ mod \ 8)$
- insert keys: $18, 41, 22, 44, 59, 32, 31, 73$



Analysis of Double Hashing

- We assume that every probe looks at a random location in the table.
- $1 - \alpha$ fraction of the table is empty.
- Expected number of probes required to find an empty location (unsuccessful search) is $1/(1 - \alpha)$

Analysis of Double Hashing

- Average no of probes for a successful search = average no of probes required to insert all the elements.
- To insert an element we need to find an empty location.

inserting	Avg no of probes	Total no of probes
First m/2	<= 2	m
Next m/4	<= 4	m
Next m/8	<= 8	m

Analysis of Double Hashing

- No of probes required to insert $m/2 + m/4 + m/8 + \ldots + m/2^i$ elements = number of probes required to leave 2^{-i} fraction of the table empty $= m \times i$.
- No of probes required to leave $1 - \alpha$ fraction of the table empty $= -mlog(1 - \alpha)$
- Average no. of probes required to insert n elements is $-(m/n)log(1 - \alpha) = -(1/\alpha)log(1 - \alpha)$

Expected Number of Probes

- Load factor $\alpha < 1$ for probing
- Analysis of probing uses uniform hashing assumption - any permutation is equally likely
  - What about linear probing and double hashing?

	unsuccessful	successful
chaining	$O(1+\alpha)$	$O(1+\alpha)$
probing	$O\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$

Expected Number of Probes