



Algorithms and Data Structures

(Overflow) Hashing

Ulf Leser

How Fast can we Search an Element in a List?

	Searching by Key	Inserting	Pre-processing
Unsorted array	$O(n)$	$O(1)$	0
Sorted array	$O(\log(n))$	$O(n)$	$O(n \cdot \log(n))$
Sorted linked list	$O(n)$	$O(n)$	$O(n \cdot \log(n))$
Priority Queue	$O(1)$ for min	$O(\log(n))$	$O(n)$
Our dream	$O(1)$	$O(1)$	0

Beyond $\log(n)$ in Searching

- Assume you have a company and ~ 2000 employees
- You often search **employees by name** to get their ID
- No employee is more important than any other
 - No differences in access frequencies, SOL or PQ don't help
- Best we can do until now
 - Sort list in array
 - Binsearch will require $\log(n) \sim 11$ comparisons per search
 - Interpolation search might be faster, but WC is the same
- Can we do better?

Recall Bucket Sort

- Bucket Sort
 - Assume $|S|=n$, the length of the longest value in S is m , alphabet Σ with $|\Sigma|=k$
 - We first sort S on first position into k buckets
 - Then sort every bucket again for second position
 - Etc.
 - After at most m iterations, we are done
 - Time complexity: $O(m*(|S|+k))$
- Fundamental idea: For finite alphabets, the characters give us a sorted partitioning of all possible values

Bucket Sort Idea for Searching

- Fix an m (e.g. $m=3$)
- There are “only” $26^3 \sim 18.000$ different prefixes of length 3 that a (German) name can start with
- Thus, we can sort any name s with prefix $s[1..m]$ in constant time into an array A with $|A|=k^m$
 - Index in A : $A[(s[0]-1)*k^0 + (s[1]-1)*k^1 + \dots + (s[m-1]-1)*k^{m-1}]$
- We can use the same formula to look-up names
- Cool: Search and insert complexity is $O(1)$ for a fixed m
 - Actually rather in $O(m)$ – we need to compute the index
 - Pre-processing is $O(m*|S|)$, inserting is $O(m)$
- But ... what if two names start with the same m -prefix?

Collisions

- Assume we use the first m characters
- $\langle \text{Müller, Peter} \rangle$, $\langle \text{Müller, Hans} \rangle$, $\langle \text{Müllheim, Ursula} \rangle$, ...
 - All start with the same 4-prefix
 - All are mapped to the **same position of A** if $m < 5$
 - Such cases are called **collisions**
- To reduce collisions, we can **increase m**
 - Requires **exponentially more space** ($a = |\Sigma|^m$)
 - But we have only 2000 employees – what a waste
 - Can't we find better ways to **map a name into an array**?

Abstraction: Dictionary Problem

- **Dictionary problem:** Manage a list S of $|S|$ keys
 - We use an **array A with $|A|=a$** (important: $a \sim n$? $a > n$? $a \gg n$?)
 - We want to support three operations
 - Store a key k in A
 - Look-up a key in A
 - Delete a key from A
- **Applications**
 - Compilers: **Symbol tables** over variables, function names, ...
 - Databases: Lists of attribute values, e.g. names, ages, incomes, ...
 - Search engines: Lists of words appearing in documents
 - ...

Content of this Lecture

- Hashing
- Collisions
- External Collision Handling
- Hash Functions
- Application: Bloom Filter

Hash Function

- Definition

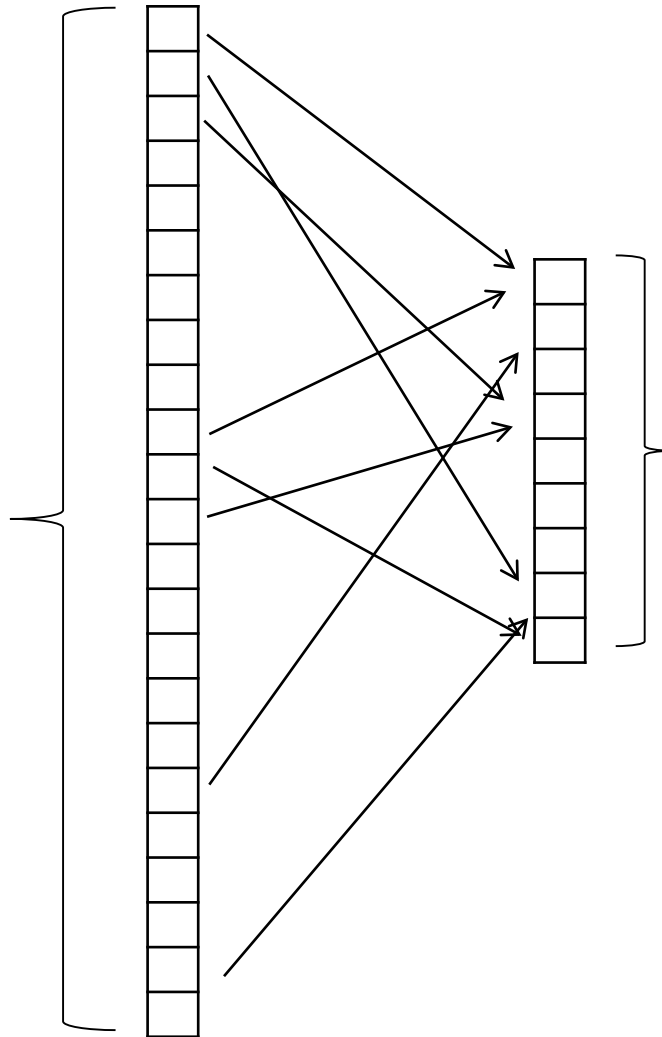
Let S with $|S|=n$ be a set of keys from a universe U and let A be an array with $a=|A|$

- A *hash function* h is a total function $h: U \rightarrow [0 \dots a-1]$
- Every pair $k_1, k_2 \in S$ with $k_1 \neq k_2$ and $h(k_1) = h(k_2)$ is called a *collision*
- h is *perfect* iff it never produces collisions
- h is *uniform*, iff $\forall i \in A: p(h(k)=i) = 1/a$
- h is *order-preserving*, iff: $k_1 < k_2 \Rightarrow h(k_1) < h(k_2)$

- Inserting: $s \in S$ is hashed into A by setting $A[h(s)] = s$
- Searching q : If $A[h(q)] = q$ then $q \in A$; otherwise not
- If we use an array A in this way, we call A a *hash table*

Illustration

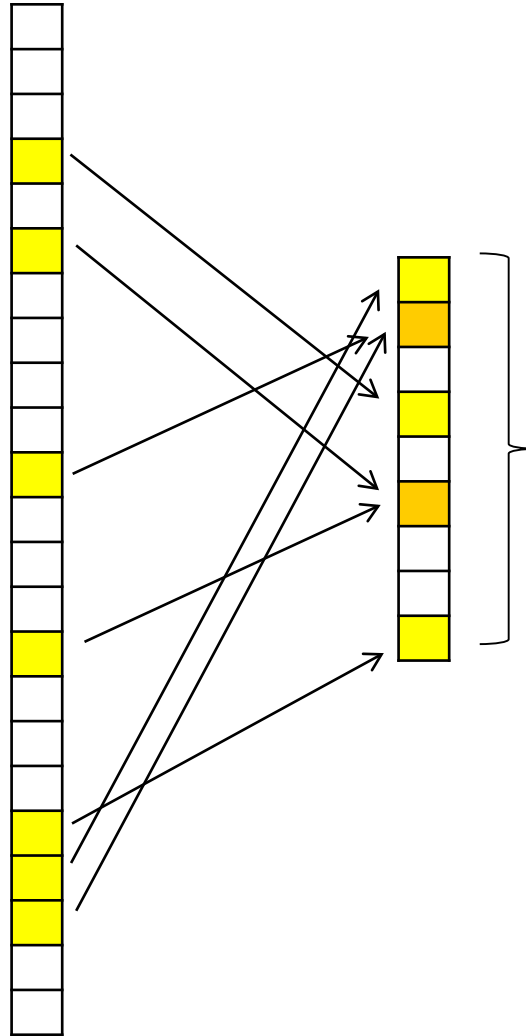
U: All possible values of a key k from S



A: All possible indexes of a k in hash table

Illustration

Actual values
of k in S



Hash table A
with collisions

Topics

- We want hash functions with as **few collisions** as possible
 - Knowing U and making **assumptions about S**
 - Example: We build a hash table for person names (U), we don't know which ones (S), but have an idea of how many ($|S|$)
- Hash functions should be **computed quickly**
 - Bad idea: Sort S and then use rank as address
- **Collisions** must be handled
 - Even if a collision occurs, we still need to give correct answers
- Don't waste space: **$|A|$ should be as small** as possible
 - Clearly, it must hold that $a \geq n$ if collisions should be avoided
- Note: Order-preserving hash functions are rare
 - Hashing is bad for **range queries**

Example

- We usually have $a \gg |S|$ yet $a \ll |U|$
 - But many different scenarios!
 - Sometimes $a < |S|$ makes perfectly sense, especially when data sets get very large (see bloom filter)
 - If S may grow and shrink a lot: [Dynamic hashing](#)
- If k is an integer (or can be turned into an integer): A simple and surprisingly good hash function:
 $h(k) := k \bmod a$ with $a = |A|$ being a prime number

Content of this Lecture

- Hashing
- Collisions
- External Collision Handling
- Hash Functions
- Application: Bloom Filter

Are Collisions a Problem?

- Assume we have a uniform hash function that maps an arbitrarily chosen key k to all positions in A with **equal probability**
- Given $|S|=n$ and $|A|=a$ – how big are the **chances to produce collisions?**

Two Cakes a Day?

- Assume an Übungsgruppe has 32 students
- Every time one has birthday, he/she brings a cake
- The Übungsgruppe meets every day over an entire year – even weekends!
- What is the chance of **having to eat two pieces of cake on at least one day in the year?**
- **Birthday paradox**
 - Each day has the same chance to be a birthday for every person
 - We ignore seasonal bias, twins, etc.
 - Guess – 5% 20% 30% 50% ?

Analysis

- Abstract formulation: **Urn with 365 balls**
 - We draw 32 times and place the ball back after every drawing
 - What is the probability $p(32, 365)$ to **draw any ball at least twice**?
- Complement of the chance to draw **no ball more than once**
 - $p(32, 365) = 1 - q(32, 365)$
 - $q(n, a)$: We draw n times one of the a balls and they are **all different**
- We draw a first ball. Then
 - Chances that the second is different from all previous balls: $364/365$
 - Chances that the 3rd ball is different from 1st and 2nd (which must be different from the 1st) is $363/365$

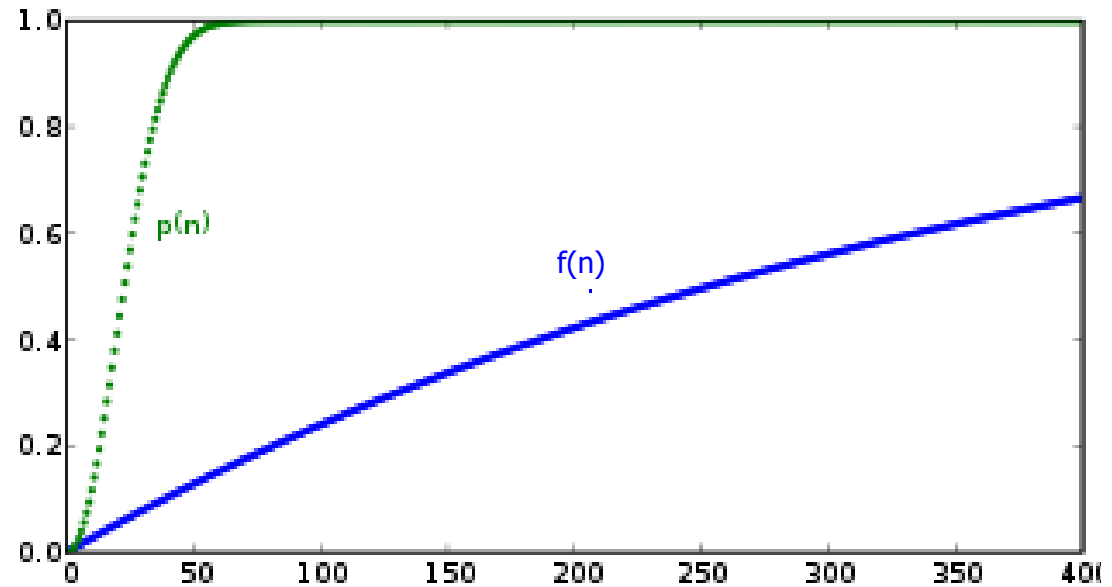
– ...

$$p(n, a) = 1 - q(n, a) = 1 - \left(\prod_{i=1}^n \frac{a - i + 1}{a} \right) = 1 - \frac{a!}{(a - n)! * a^n}$$

Results

Source: Wikipedia

5	2,71
10	11,69
15	25,29
20	41,14
25	56,87
30	70,63
32	75,33
40	89,12
50	97,04



- $p(n)$ here means $p(n,365)$
- $f(n)$: Chance that someone has birthday on the **same day as you**

Take-home Messages

- Just by chance, there are **many more collisions** than one intuitively expects
- Collision handling is a **real issue**
- **Additional time/space** it takes to manages collisions must be taken into account

Content of this Lecture

- Hashing
- Collisions
- External Collision Handling
- Hash Functions
- Application: Bloom Filter

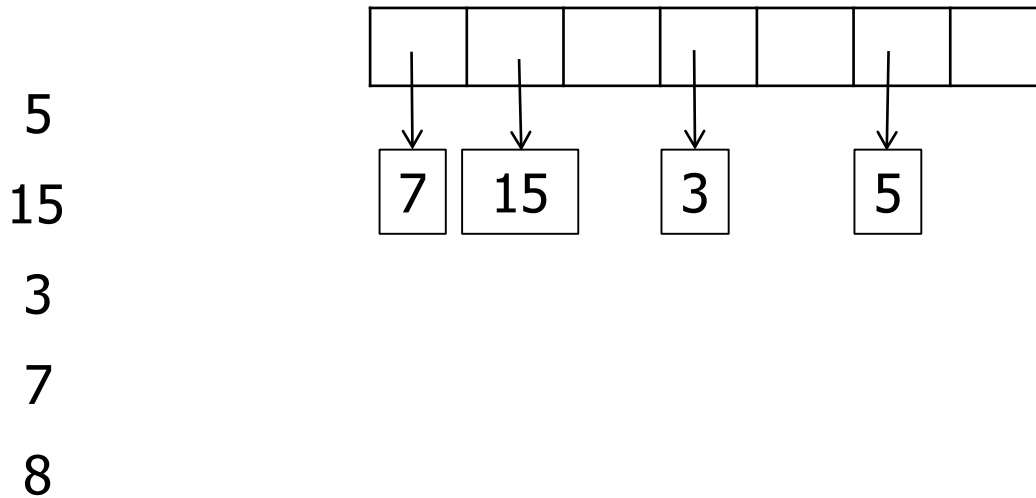
Hashing: Three Fundamental Methods

- **Overflow hashing:** Collisions are stored outside A
 - We need additional storage
 - Solves the problem of A having a fixed size despite that S might be growing (without changing A)
- **Open hashing:** Collisions are managed inside A
 - No additional storage
 - $|A|$ is upper bound to the amount of data that can be stored
 - Next lecture
- **Dynamic hashing:** A may grow/shrink
 - Not covered here – see Databases II

Overflow Hashing

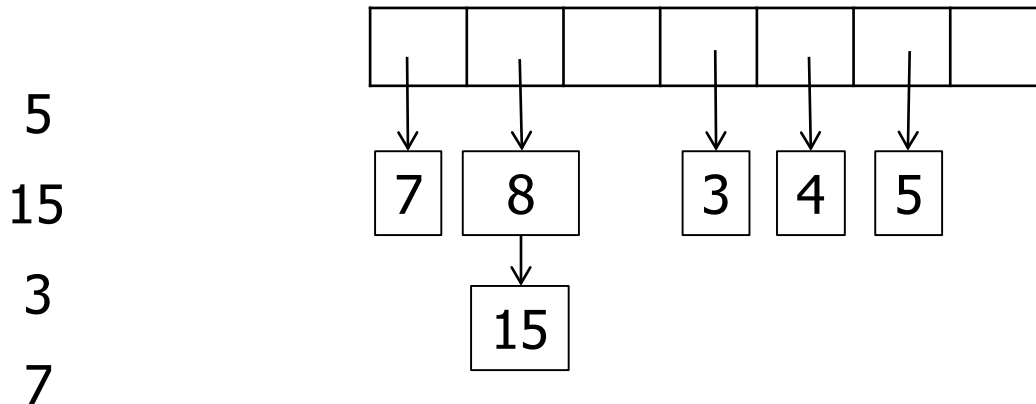
- Two possibilities (assuming a linked list)
- **Separate chaining**: $A[i]$ stores **tuple** (k_0, p) , where p is a pointer to a list storing all keys k with $h(k)=A[i]$ except the first one k_0
 - For 1 key we need space $|k|+|prt|$; for 2: $2*(|k|+|prt|)$; for 3 ...
 - Separate treatment of 1st key in all operations
 - Good if collisions are rare (zero pointer chasing)
- **Direct chaining**: $A[i]$ is a **pointer** to list storing all keys mapped to i
 - For 1 key we need $|prt|+|k|+|prt|$; for 2: $|prt|+2*(|k|+|prt|)$; ...
 - Uniform treatment
 - More efficient if collisions are frequent (less “if ... then ... else”)

Example, Direct Chaining ($h(k) = k \bmod 7$)



- Assume a **linked list**, insertions at list head

Example ($h(k) = k \bmod 7$)



5

15

3

7

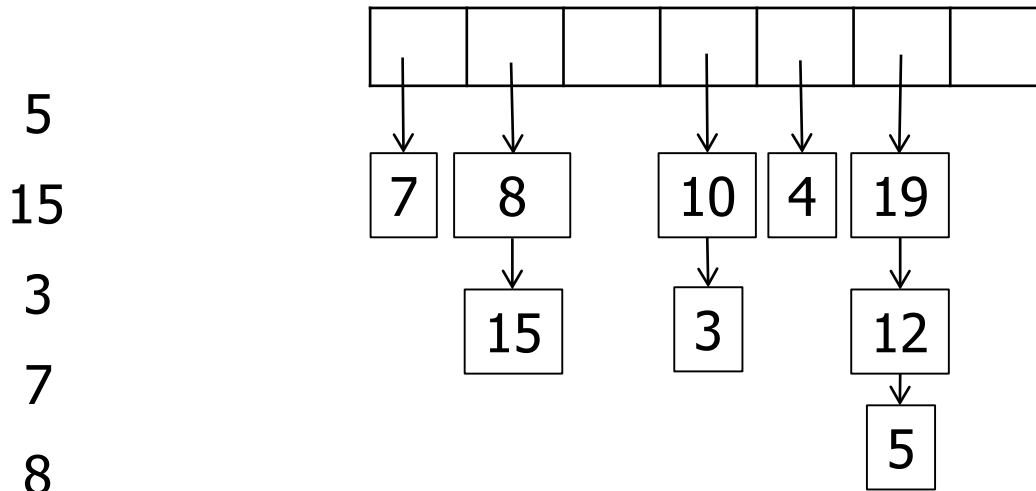
8

4

12

- Assume a **linked list**, insertions at list head

Example $(h(k) = k \bmod 7)$



5

15

3

7

8

4

12

19

10

- Assume a **linked list**, insertions at list head
- Space complexity: $O(a+n)$
- Time complexity (worst-case)
 - Insert: $O(1)$
 - Search: $O(n)$ – if all keys map to the same bucket
 - Delete: $O(n)$ – we first need to search

Average Case Complexities

- Assume h uniform and elements are inserted in randomized order
- After having inserted n values, every overflow list has $\alpha \sim n/a$ elements
 - α is called the **fill degree** of the hash table
- How long does the $n+1^{\text{st}}$ operation **take on average?**
 - Insert: $O(1)$
 - Search: **If $k \in L$: $\alpha/2$ comparisons**; else α comparisons
 - This is in $O(n/a)$
 - Delete: Same as search
 - OK'ish, if α is small and hashing is uniform, i.e., if $|A| \sim O(|S|)$

Improvement

- We may keep every **overflow list sorted**
 - If stored in a (dynamic) array, binsearch requires $\log(\alpha)$
 - Disadvantage: **Insert requires $\alpha/2$** to keep list sorted (AC)
 - If stored in a linked list, searching k ($k \in L$ or $k \notin L$) requires $\alpha/2$
 - Disadvantage: **Insert requires $\alpha/2$** to keep list sorted (AC)
 - If we first have many inserts (build-phase of a dictionary), then mostly searches, it is better to first build unsorted overflow lists and sort only once the phase **changes**
- We may also use a **second (smaller) hash table** with a different hash function
 - Especially if some overflow lists grow very large (skew)
 - See Double Hashing (next lecture)

But ...

- Searching with $\sim \alpha/2$ comparisons on average doesn't seem too attractive
- But: One typically uses hashing in cases where α is small
 - Often, $\alpha < 1$ – search on average takes only constant time
 - $1 \leq \alpha \leq 10$ – search takes only ~ 5 comparisons
- For instance, let $|S|=n=10.000.000$ and $a=1.000.000$
 - Hash table (uniform, average): ~ 5 comparisons
 - Binsearch: $(\log(1E7), \text{average}) \sim 23$ comparisons
- But: In many situations values in S are skewed
 - Uniformity assumption wrong, if hash function cannot handle skew
 - Average case estimation may go grossly wrong
 - Experiments help

Content of this Lecture

- Hashing
- Collisions
- External Collision Handling
- Hash Functions
- Application: Bloom Filter

Hash Functions

- Requirements
 - Should be **computed quickly**
 - Should **spread keys equally** over A – for any S
 - Should use all positions in A with equal probability
- Simple and often good: $h(k) := k \bmod a$
 - “**Division-rest method**”
 - If a is prime: Few collisions for many real world data (empirical observation)

Hash-Algorithmen [\[Bearbeiten\]](#)

Bekannte [\[Bearbeiten\]](#)

- Divisions-Rest-Methode
- Doppel-Hashing
- Brent-Hashing
- Kuckucks-Hashing
- Multiplikative Methode
- Mittquadratmethode
- **Zerlegungsmethode**
- **Ziffernanalyse**
- Quersumme

Allgemeine [\[Bearbeiten\]](#)

- Adler-32
- FNV
- Hashtabelle
- Merkes Meta-Verfahren
- Modulo-Funktion
- Parität
- Prüfsumme
- Prüfziffer
- Quersumme
- Salted Hash
- Zyklische Redundanzprüfung

Gitterbasierte [\[Bearbeiten\]](#)

- **Ajtai**
- **Micciancio**
- **Peikert-Rosen**
- Schnelle Fourier-Transformation (FFT Hashfur
- **LASH**^[3]

Algorithmen in der Kryptographie [\[B](#)

- MD2, MD4, MD5
- SHA

Other Hash Functions

- “Multiplikative Methode”: $h(k) = \text{floor}(a * (k * x - \text{floor}(k * x)))$
 - Multiply k with some x , remove the integer part, multiply with a and cut to the next smaller integer value
 - x : any real number; best distribution on average for $x = (1 + \sqrt{5})/2$ - [Goldener Schnitt](#)
- “Quersumme”: $h(k) = (k \bmod 10) + \dots$
- For strings: $h(k) = (f(k) \bmod a)$ with $f(k) =$ “add byte values of all characters in k ”
- No limits to fantasy
 - Look at your data and its [distribution of values](#)
 - Make sure local clusters are resolved

$$\frac{a + b}{a} = \frac{a}{b}$$

Java hashCode()

```
1. /** * Returns a hash code for this string. The hash code for a
2. * <code>String</code> object is computed as
3. * <blockquote><pre>
4. *  $s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$ 
5. * </pre></blockquote>
6. * using <code>int</code> arithmetic, where <code>s[i]</code> is the
7. * <i>i</i>th character of the string, <code>n</code> is the length of
8. * the string, and <code>^</code> indicates exponentiation.
9. * (The hash value of the empty string is zero.) *
```

- **Object.hashCode()**

The [default hashCode\(\)](#) method uses the 32-bit internal JVM address of the Object as its hashCode. However, if the Object is moved in memory during garbage collection, the hashCode stays constant. This default hashCode is not very useful, since to look up an Object in a HashMap, you need the [exact same key Object](#) by which the key/value pair was originally filed. Normally, when you go to look up, you don't have the original key Object itself, just some data for a key. So, unless your key is a String, [nearly always you will need to implement a hashCode and equals\(\)](#) method on your key class.

Hashing

- **Two key ideas** to achieve scalability for relatively simple problems on very large datasets: **Sorting / Hashing**



Foodnetwork.com

Pros / Cons

Sorting

- Search: $O(\log(n))$ in WC/AC
- Preprocessing: $O(n \cdot \log(n))$
- Insert: $O(n)$ (wait for [AVL](#))
- [Robust against skew](#)
- App/domain independent method
- No additional space
- Sometimes preferable

Hashing

- Search: AC $O(1)$, WC $O(n)$
- Preprocessing: Linear
- Insert: AC $O(1)$, WC $O(n)$
- Sensible to skew
- [App/domain specific hash functions](#) and strategies
- Usually add. space required
- Sometimes preferable

Content of this Lecture

- Hashing
- Collisions
- External Collision Handling
- Hash Functions
- **Application: Bloom Filter**

Searching an Element

- Assume we want to know if k is an element of a list S of 32bit integers – and S is very large
- S must be stored on disk
 - Assume testing k in memory costs very little, but loading a block (size $b=1000$ keys) from disk costs enormously more
 - Thus, we only count IO – how many blocks do we need to load?
 - Everything in main memory is assumed free – negligible cost
- Assume $|S|=1E7$ ($1E4$ blocks), but we have enough memory for only 1000 blocks ($=1E6$ keys)
 - Thus, enough for only 10% of the data
- How can we test efficiently if a given query k is in S ?

Options

- If S is not sorted
 - If $k \in S$, we need to load 50% of S on average: $\sim 0.5E4$ IO
 - If $k \notin S$, we need to load S entirely: $\sim 1E4$ IO
- If S is sorted
 - It doesn't matter whether $k \in S$ or not
 - We need to load $\log(|S|/b) = \log(1E4) \sim 14$ blocks
 - If we can address blocks by their position within the list in $O(1)$
- Notice that we are not using our memory ...

Idea of a Bloom Filter

- Build a **hash map A** as **big as the memory**
- Use A to **indicate** whether a key is in S or not
- The test may go wrong, but only in one direction
 - If $k \in A$, we don't know if $k \in S$
 - If $k \notin A$, we **know for sure that $k \notin S$**
- A acts as a filter: **A Bloom filter**
 - Bloom, B. H. (1970). "Space/Time Trade-offs in Hash Coding with Allowable Errors." Communications of the ACM 13(7): 422-426.

Bloom Filter: Simple

- Create a bitarray A with $|A|=a=1E6*32$ bits
 - We fully exploit our memory
 - A is always kept in memory
- Choose a (uniform) hash function h into A
- Initialize A (offline) and keep in memory: $\forall k \in S: A[h(k)]=1$
 - Preprocessing
- Searching k given A (in memory)
 - If $A[h(k)]=0$, we know that $k \notin S$ (with 0 IO)
 - If $A[h(k)]=1$, we need to search k in S
 - Because we didn't handle collisions

Bloom Filter: Advanced

- Choose j independent (uniform) hash functions h_j
 - Independent: The values of one hash function are statistically independent of the values of all other hash functions
- Initialize A (offline): $\forall k \in S, \forall j: A[h_j(k)] = 1$
- Searching k given A (in memory)
 - If any of the $A[h_j(k)] = 0$, we know that $k \notin S$
 - If all $A[h_j(k)] = 1$, we need to search k in S

Analysis

- Assume $k \notin S$
 - Let C_n be the cost of such a (negative) search
 - We only access disk if all $A[h_j(k)] = 1$ – how often?
 - In all other cases, we perform no IO and have 0 cost
- Assume $k \in S$
 - We will certainly access disk, as all $A[h_j(k)] = 1$ but we don't know if this is by chance or not (collisions)
 - Thus, $C_p = 14$
 - Using binsearch, assuming S is kept sorted on disk
- Average cost of u searches then is:
$$C_{\text{avg}} := (w_1 * C_n + w_2 * C_p) / u$$

Chances for a False Positive

- For one $k \in S$ and one (uniform) hash function, the chance for a given position in A to be 0 is $1-1/a$
- For j hash functions, chances that all remain 0 is $(1-1/a)^j$
 - Assuming all hash functions are statistically independent
- For j hash functions and n values, chances to remain 0 is $q=(1-1/a)^{j*n}$
- Prob. of a given bit being 1 after inserting n values is $1-q$
- Now let's look at a search for key k , which tests j bits
- Chances that all of these are 1 by chance is $(1-q)^j$
- Thus, $C_n=(1-q)^j*C_p + (1-(1-q)^j)*0$
 - We have $n=|S|=1E7$, $a=|A|=32E6$
 - This gives: $j=2$: 13,94; $j=5$: 4,31; $j=10$: 8,93
 - Trade-off: Small j -> little filtering; large j -> cluttered hash table

Average Case

- Assume we look for **all possible values** ($|U|=u=2^{32}$) with the same probability
- $(u-|S|)/u$ of the searches are negative, $|S|/u$ are positive
- **Average cost per search** is

$$C_{\text{avg}} := ((u-|S|)*C_n + |S|*C_p) / u$$

- For $j=5$: 5,49
- For $j=10$: 0,64
 - Larger j decreases average cost, but increase effort for each single test, which is not part of our cost model
 - What is the optimal value for j ?
- **Much better than sorted lists**

Exemplary questions

- Assume $|A|=a$ and $|S|=n$ and a uniform hash function. What is the fill degree of A ? What is the AC search complexity if collisions are handled by direct chaining? What if collisions are handled by separate chaining?
- Assume the following hash functions $h=\dots$ and S being integers. Show A after inserting each element from $S=\{17,256,13,44,1,2,55,\dots\}$
- Describe the standard JAVA hash function. When is it useful to provide your own hash functions for your own classes?