Algorithms and Data Structures

(Overflow) Hashing

Ulf Leser

This Module

• Introduction	2
• Abstract Data Types	1
• Complexity analysis	1
• Styles of algorithms	1
• Lists, stacks, queues	2
• Sorting (lists)	3
• Searching (in lists, PQs, SOL)	5
• Hashing (to manage lists)	2
• Trees (to manage lists)	4
• Graphs (no lists!)	5
• Sum	**15/26**

How fast can we Search Elements?

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

Pros / Cons

Sorting

- Typically O(log(n)) for searching in WC and AC
- Requires sorting first (which can be reused)
- App/domain independent method
- No additional space
- Efficient for extensible DSs
- Sometimes preferable

Hashing

- Typically O(1) AC, but worst case O(n)
- Linear preprocessing (which can be reused)
- App/domain specific hash functions and strategies
- Usually add. space required
- Extensibility is difficult
- Sometimes preferable

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)~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, all characters of elements in S from 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 partitioning of all possible values

Bucket Sort Idea for Searching

- Fix an m (e.g. m=3)
- There are "only" 26^3~18.000 different prefixes of length 3 that a (German) name can start with (ignoring case)
- Thus, we can "sort" a name s with prefix s[1..m] in constant time into an array A with $|A|=k^m$
  - Index in A: $A[(s[1]-1)*k^0 + (s[2]-1)*k^1 + \dots +(s[m]-1)*k^{m-1}]$
- We can use the same formula to look-up names in O(1)
  - Actually it is O(m)
- Looks cool: Search complexity is O(1) for a fixed m
  - Pre-processing is O(|S|), inserting is O(1)
- But ….

Key Idea of Hashing

- Given a list S of $|S|=n$ values and an array A, $|A|=a$
  - S: Values we manage; A: Positions available for management
- Define a hash function $h: S \rightarrow [0,a-1]$
- Store each value $s \in S$ in $A[h(s)]$
- To test whether a value q is in S, check if $A[h(q)] \neq null$
- Inserting and lookup is O(1)
  - If computing the hash function is O(1)
- But wait ...

Collisions

- Assume h maps to the m first characters
- <Müller, Peter>, <Müller, Hans>, <Müllheim, Ursula>, ...
  - All start with the same 4-prefix
  - All are mapped to the same position of A if m<5
  - These cases are called collisions
- To minimize 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?
  - What are good hash functions?

Abstraction: Dictionary Problem

- Dictionary problem: Manage a list S of |S| keys
  - We use an array A with |A|=a (usually a>>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 objects such as names, ages, incomes, ...
  - Search engines: Lists of words appearing in documents
  - ...

Content of this Lecture

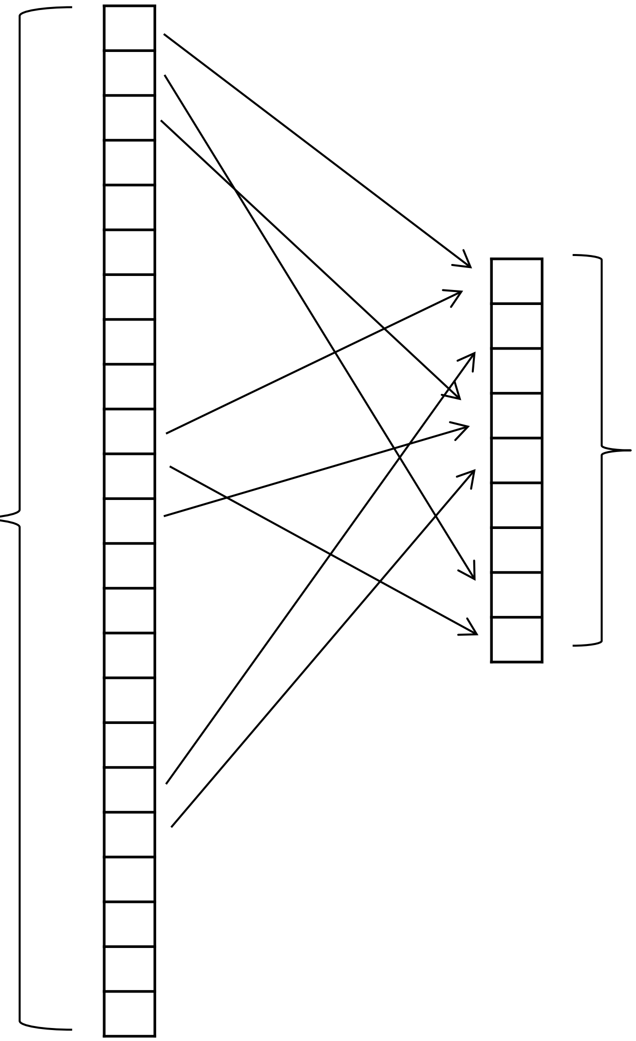
- Hashing
- Collisions
- External Collision Handling
- Hash Functions
- Application: Bloom Filter

Hash Function

- Definition
  Let S, $|S|=n$, be a set of keys from a universe U and A a (disjoint) set of values with $a=|A|$
  - *A hash function h is a total function $h: U \rightarrow A$*
  - *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: $k_1 < k_2 \Rightarrow h(k_1) < h(k_2)$*
- We always use $A=\{0,\dots,a-1\}$
  - Because we want to use $h(k)$ as address for storing k in an array

Illustration

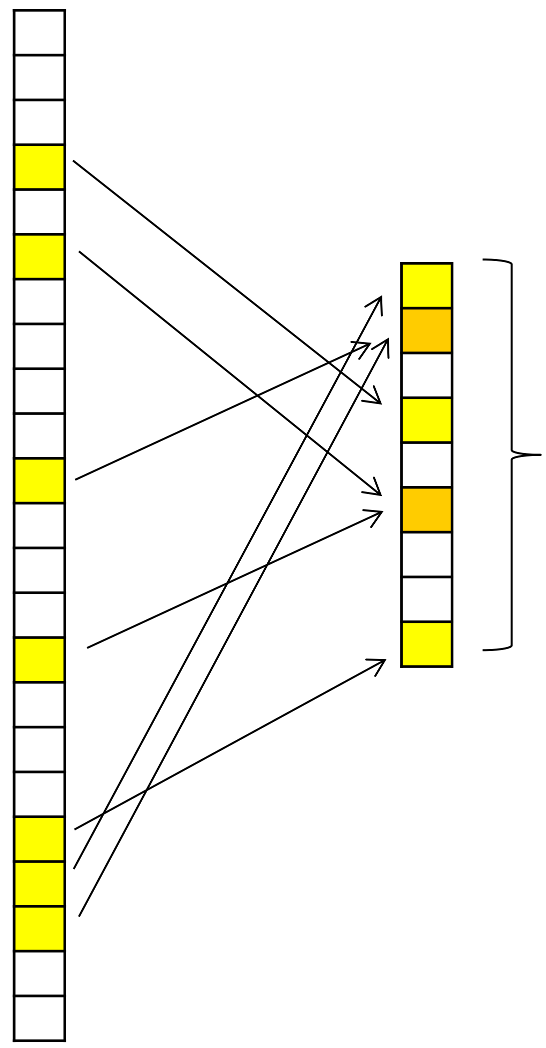
U: All possible values of k



All a addresses of hash table A

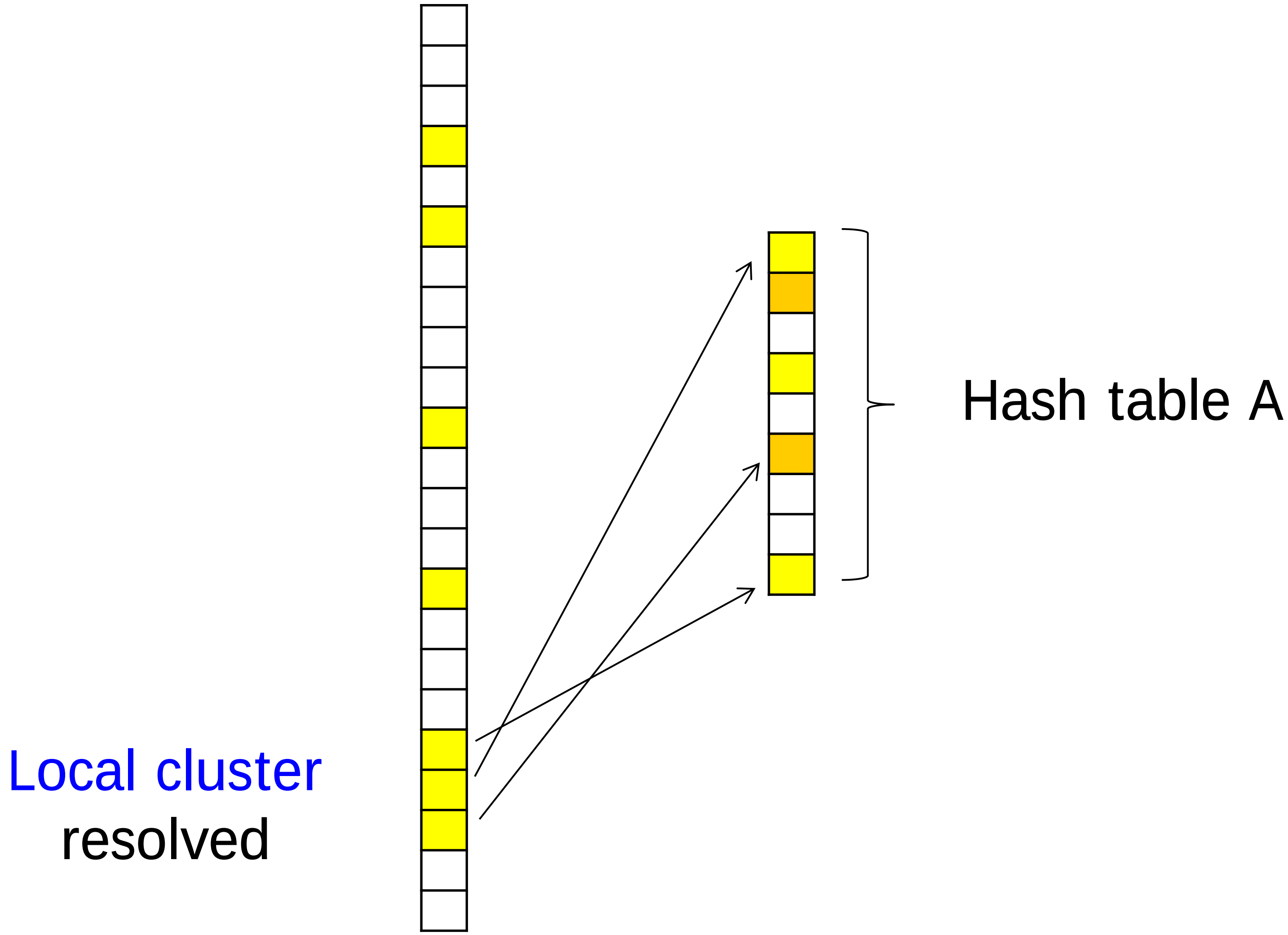
Illustration

Actual values
of k in S



Hash table A
with collisions

Illustration



Topics

- We want hash functions with as few collisions as possible
  - Knowing U and making assumptions about 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 must be avoided
- Note: Order-preserving hash functions are rare
  - Hashing is bad for range queries

Example

- We usually have $a >> |S|$ yet $a << |U|$
  - But many different scenarios! Sometimes $a < |S|$ makes perfectly sense, especially when data sets get very large
- 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?

- Each Übungsgruppe at the moment has ~32 persons
- Every time one has birthday, he/she brings a cake
- What is the chance of having to eat two pieces of cake on one day?
- 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

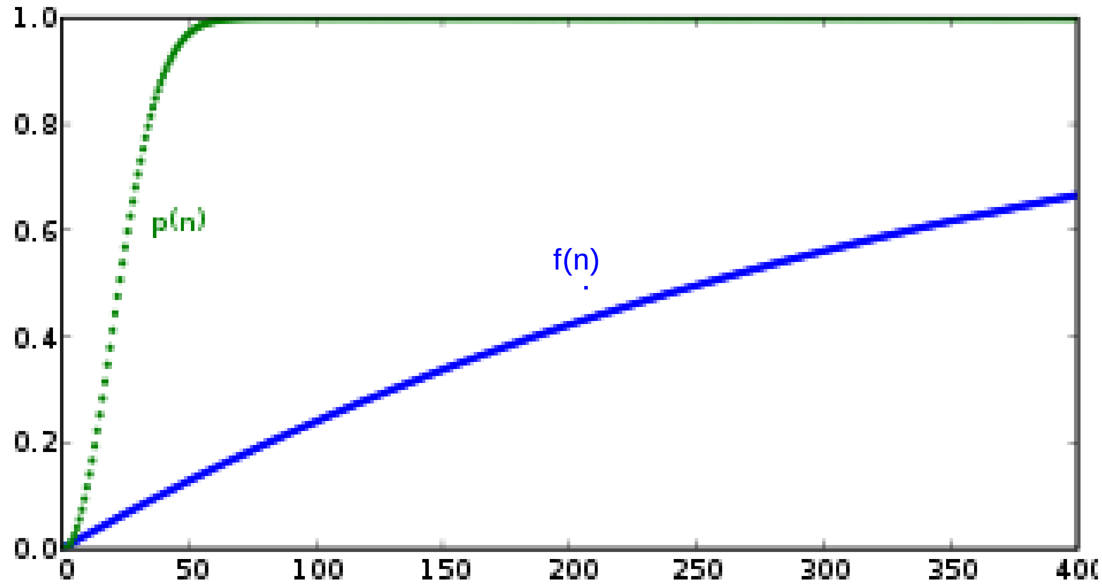
- Abstract formulation: Urn with 365 balls
  - We draw 32 times and place the ball back after every drawing
  - What is the probability p(32, 365) to draw any ball at least twice?
- Complement of the chance to draw no ball more than once
  - p(32, 365) = 1 – q(32,365)
- q(X,Y): We only draw different balls
- We draw a first ball. Then
  - Chances that the second is different from all previous balls: 364/365
  - Chances that the 3rd 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

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

Source: Wikipedia



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

Take-home Messages

- Collision handling is a real issue
- Just by chance, there are many more collisions than one intuitively expects
- 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 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

Collision Handling

- In Overflow Hashing, we store values not fitting into A in separate data structures (lists)
- Two possibilities
  - 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
    - Good if collisions are rare; if keys are small
  - Direct chaining: A[i] is a pointer to list storing all keys mapped to i
    - Less “if ... then ... else”; more efficient if collisions are frequent; if keys are large

Example, Direct Chaining (h(k)= k mod 7)

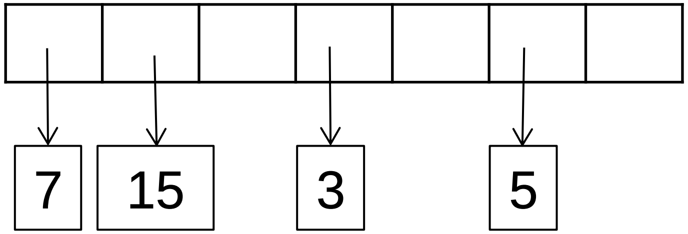
5

15

3

7

8



- Assume a linked list, insertions at list head

Example (h(k)= k mod 7)

5

15

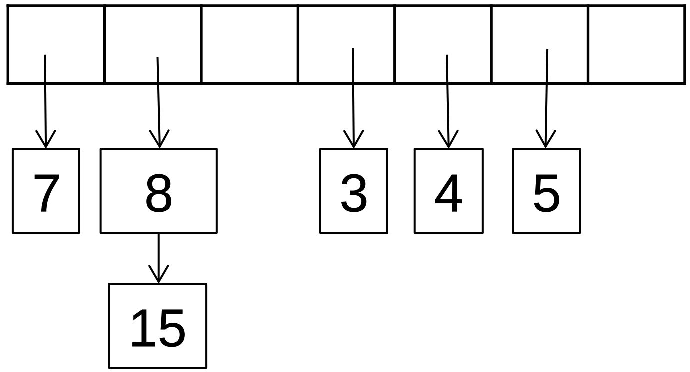
3

7

8

4

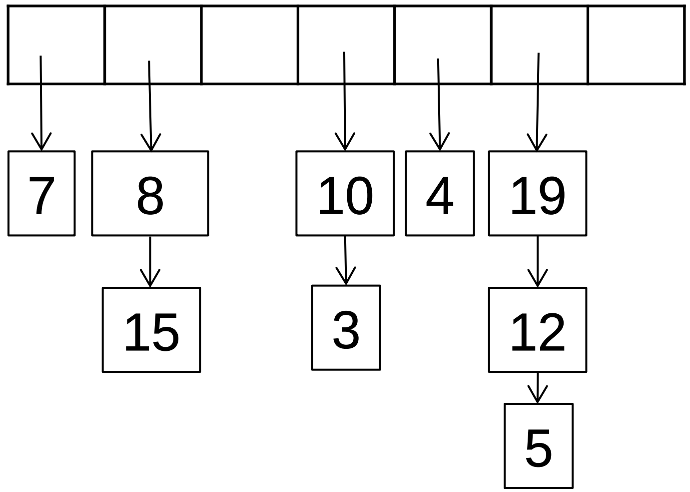
12



- Assume a linked list, insertions at list head

Example (h(k)= k mod 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 cell
  - 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+1st 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

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 overflows and only once sort overflow lists when changing phase
- We may also use a second (smaller) hash table with a different hash function
  - Especially if some overflow lists grow very large
  - See Double Hashing (next lecture)

But ...

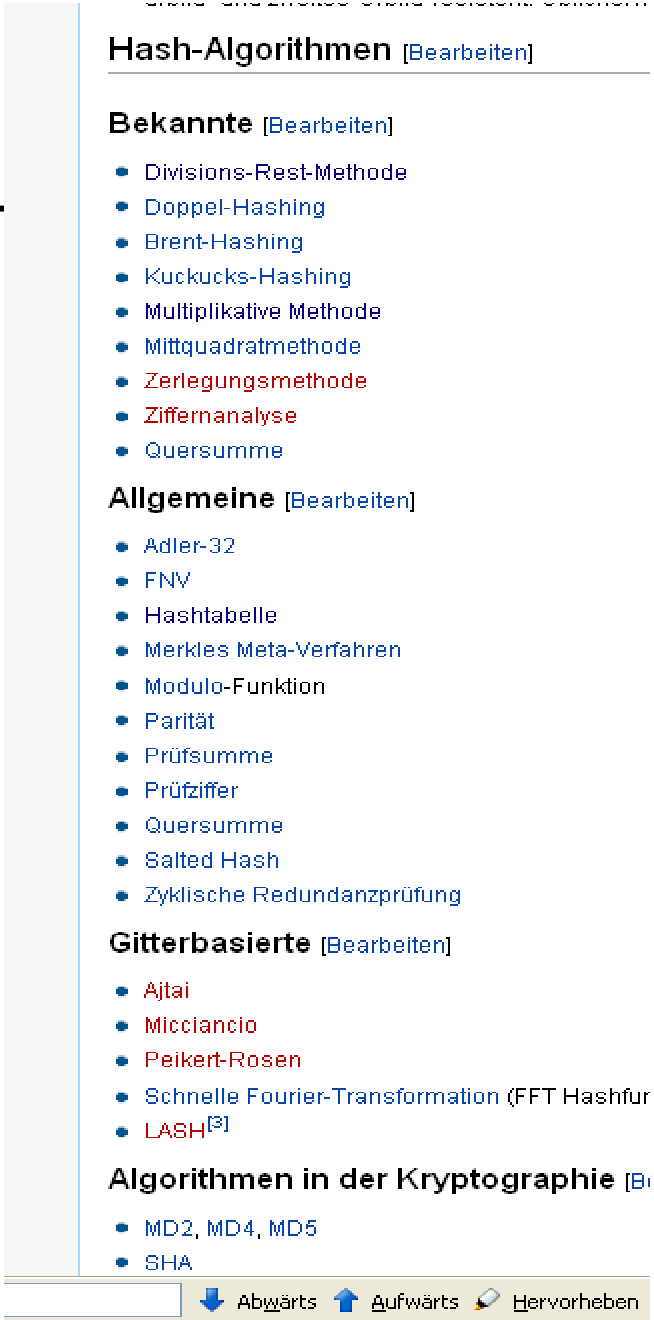
- Searching with ~α/2 comparisons on average doesn't seem too attractive
- But: One typically uses hashing in cases where α is small
  - Usually, α<1 – search on average takes only constant time
  - 1≤α≤10 – search takes only ~5 comparisons
- For instance, let |S|=n=10.000.000 and a=1.000.000
  - Hash table (uniform, average): ~5 comparisons
  - Binsearch: log(1E7, average)~23 comparisons
- But: In many situations values in S are highly 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 even if local clusters exist
  - Should use all positions in A with equal probability
- Simple and good: h(k) := k mod 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
- Merkles 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) = floor(a*(k*x–floor(k*x)))
  - Multiply k with x, remove the integer part (a bit like div), multiply with a and cut to the next smaller integer value
  - x: any real number; best distribution on average for x=(1+√5)/2 - Goldener Schnitt

- "Quersumme": h(k) = (k mod 10) + ...
- For strings: h(k) = (f(k) mod 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

Java hashCode()

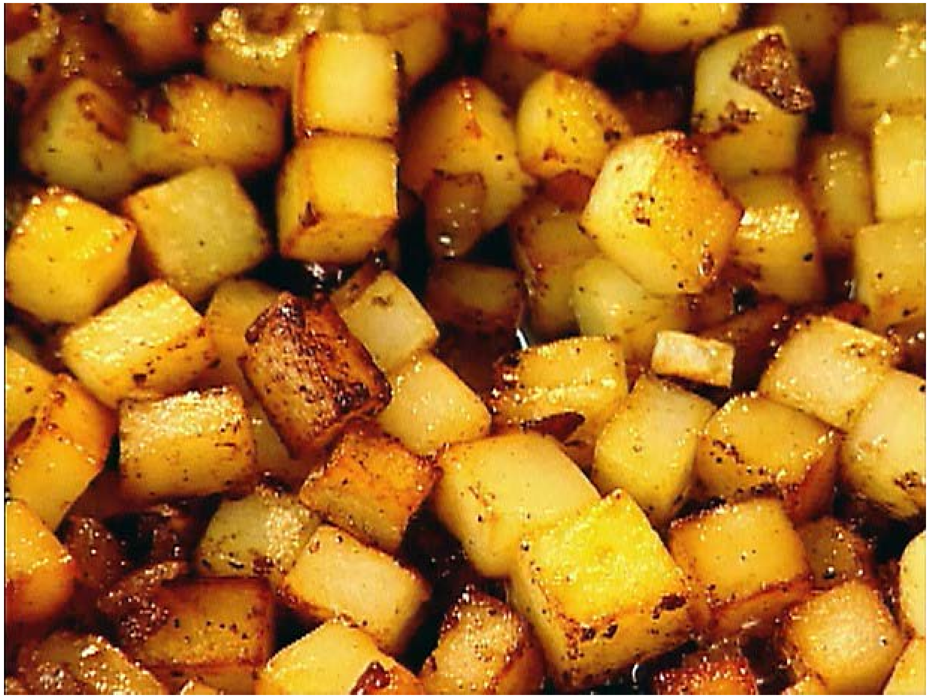
```
/** * Returns a hash code for this string. The hash code for a
* <code>String</code> object is computed as
* <blockquote><pre>
* s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
* </pre></blockquote>
* using <code>int</code> arithmetic, where <code>s[i]</code> is the
* <i>i</i>th character of the string, <code>n</code> is the length of
* the string, and <code>^</code> indicates exponentiation.
* (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

- Typically O(log(n)) for searching in WC and AC
- Requires sorting first (which can be reused)
- App/domain independent method
- No additional space
- Efficient for extensible DSs
- Sometimes preferable

Hashing

- Typically O(1) AC, but worst case O(n)
- Linear preprocessing (which can be reused)
- App/domain specific hash functions and strategies
- Usually add. space required
- Extensibility is difficult
- 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 – but S is very large
  - We shall from now on count in “keys” = 32bit
- 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|=1E9 (1E6 blocks) and we have enough memory for 1E6 keys
  - Thus, enough for 1000 of the 1 Million blocks

Options

- If S is not sorted
  - If $k \in S$, we need to load 50% of S on average: ~ 0.5E6 IO
  - If $k \notin S$, we need to load S entirely: ~ 1E6 IO
- If S is sorted
  - It doesn't matter whether $k \in S$ or not
  - We need to load log(|S|/b)=log(1E6)~20 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 fail, but only in one direction
  - If $k \in A$, we don't know for sure 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): $\forall k \in S$: A[h(k)]=1
- Searching k given A (online)
  - Test A[h(k)] 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

Bloom Filter: Advanced

- Create a bitarray A with |A|=a=1E6*32
  - We fully exploit our memory
  - A is always kept in memory
- 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 (online)
  - $\forall j$: Test $A[h_j(k)]$ 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 denote C_n 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 assume 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 of not (collisions)
  - Thus, $C_p = 20$
    - Using binsearch, assuming S is kept sorted on disk

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 $(1-q)^j$
  - By chance means: Case when k is not in S
- Thus, $C_n=(1-q)^j*C_p + (1-(1-q)^j)*0$
  - In our case, for j=5: 0.001; j=10: 0.000027

Average Case

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

$$c_{avg} := ((u-n)*C_n + n*C_p) / u$$

- For j=5: 0,14
- For j=10: 0,13
  - Larger j decreases average cost, but increase effort for each single test
  - 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=... and S being integers. Show A after inserting each element from S={17,256,13,44,1,2,55,...}
- Describe the standard JAVA hash function. When is it useful to provide your own hash functions for your own classes?