Algorithms and Data Structures

Hashing

Ulf Leser

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, self-organizing lists don't help
- Best we can do until now
  - Sort list in array
  - Binsearch will require log(n)~13 comparisons per search
- Can't we do better?

Recall Bucket Sort

- Bucket Sort
  - Assume |A|=n, m being the length of the longest value in A, values from A over an alphabet Σ with |Σ|=k
  - We first sort A 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*|A|)
- Fundamental idea: For finite alphabets, the letters give us a partition of all possible values in linear time such that the partitions are sorted in the right order

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: $A[(s[1]-1)*k^0 + (s[2]-1)*k^1 + ... +(s[m]-1)*k^{m-1}]$
- We can use the same formula to look-up names in O(1)
- That's cool. Search complexity is constant; it does not depend on n

Key Idea of Hashing

- Given a list S of $|S|=n$ values and an array A of size a
- Define a hash function $h: S \rightarrow [0,a-1]$
- Store each value $s \in S$ in $A[h(s)]$
- To test whether a query q is in S, check if $A[h(q)] \neq null$
- Inserting and lookup is O(1)
- But wait …

Two Fundamental Problems in Hashing

- 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 (and hence a)
  - Increasing m requires exponentially more space ($n=|\Sigma|^m$)
  - But we have only 2000 employees – what a waste
  - Can't we find better ways to map a name into an array address?
  - What are good hash functions?

Example: 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
  - ...

Example: Hashing without Hash Table

- Unix applies a secret hash function to passwords
- The hashed codes are stored in clear text (user, h(pw))
- Authentication: For given p, test if h(p)=h(pw)
- Advantage: h is user-independent; no user-secret data is stored
- Here, we really want to avoid collision as much as possible

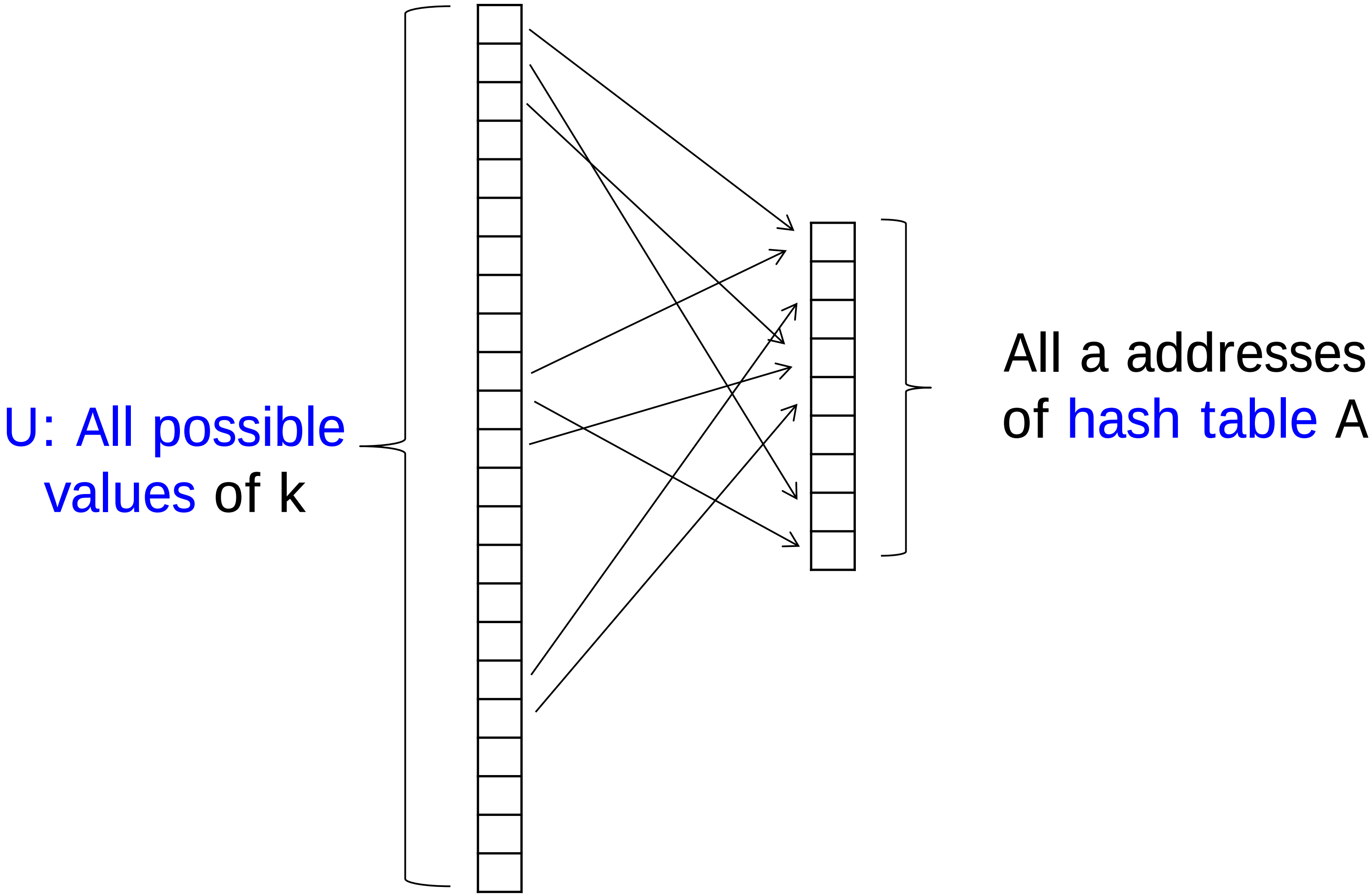
Content of this Lecture

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

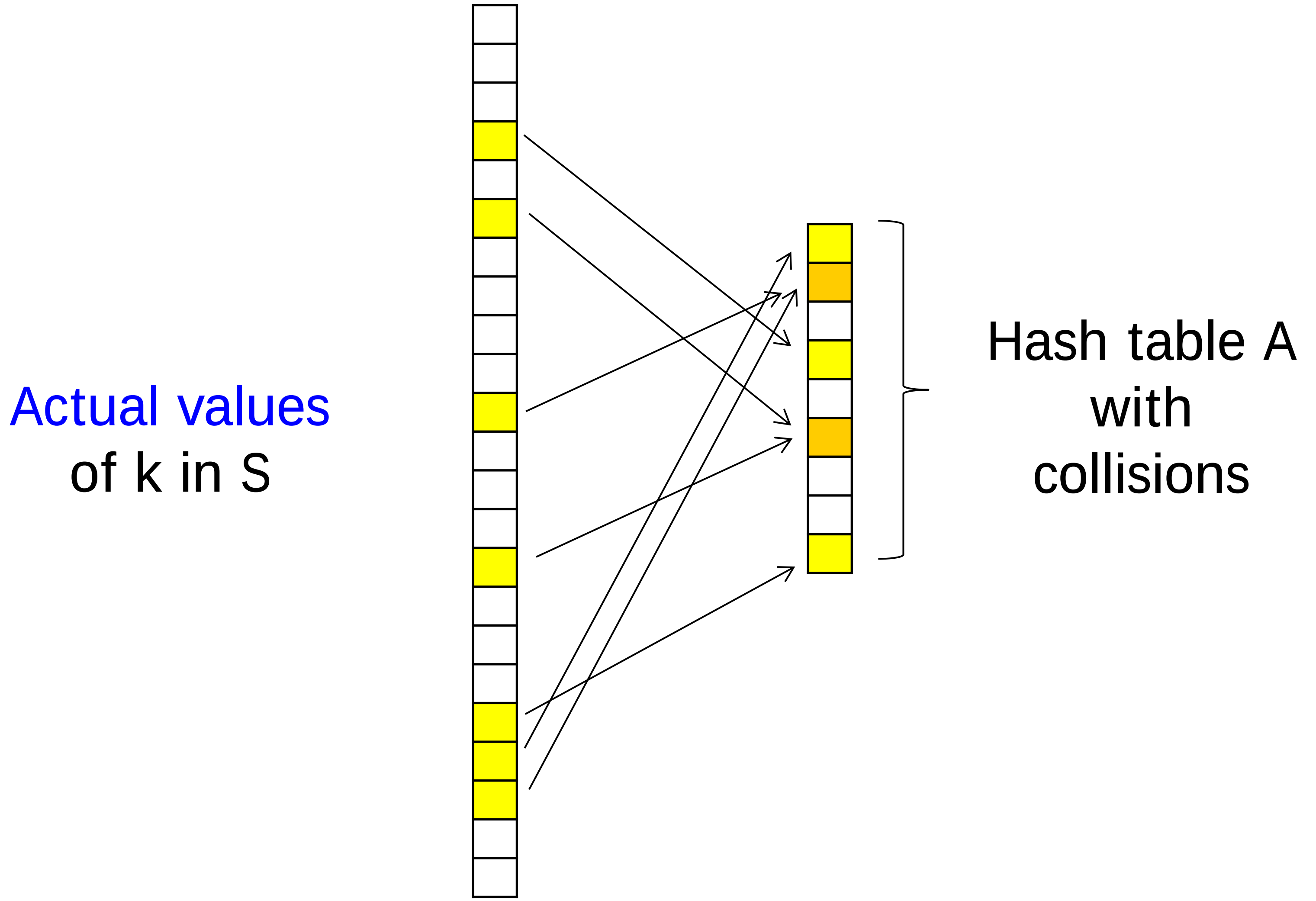
Hash Function

- Definition
  Let S, $|S|=n$, be a set of keys from a universe U and A a set of target 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* *if it never produces collisions*
  - *h* *is uniform*, *if $p(h(k)=i)=1/a$ for all positions $i \in A$*
  - *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

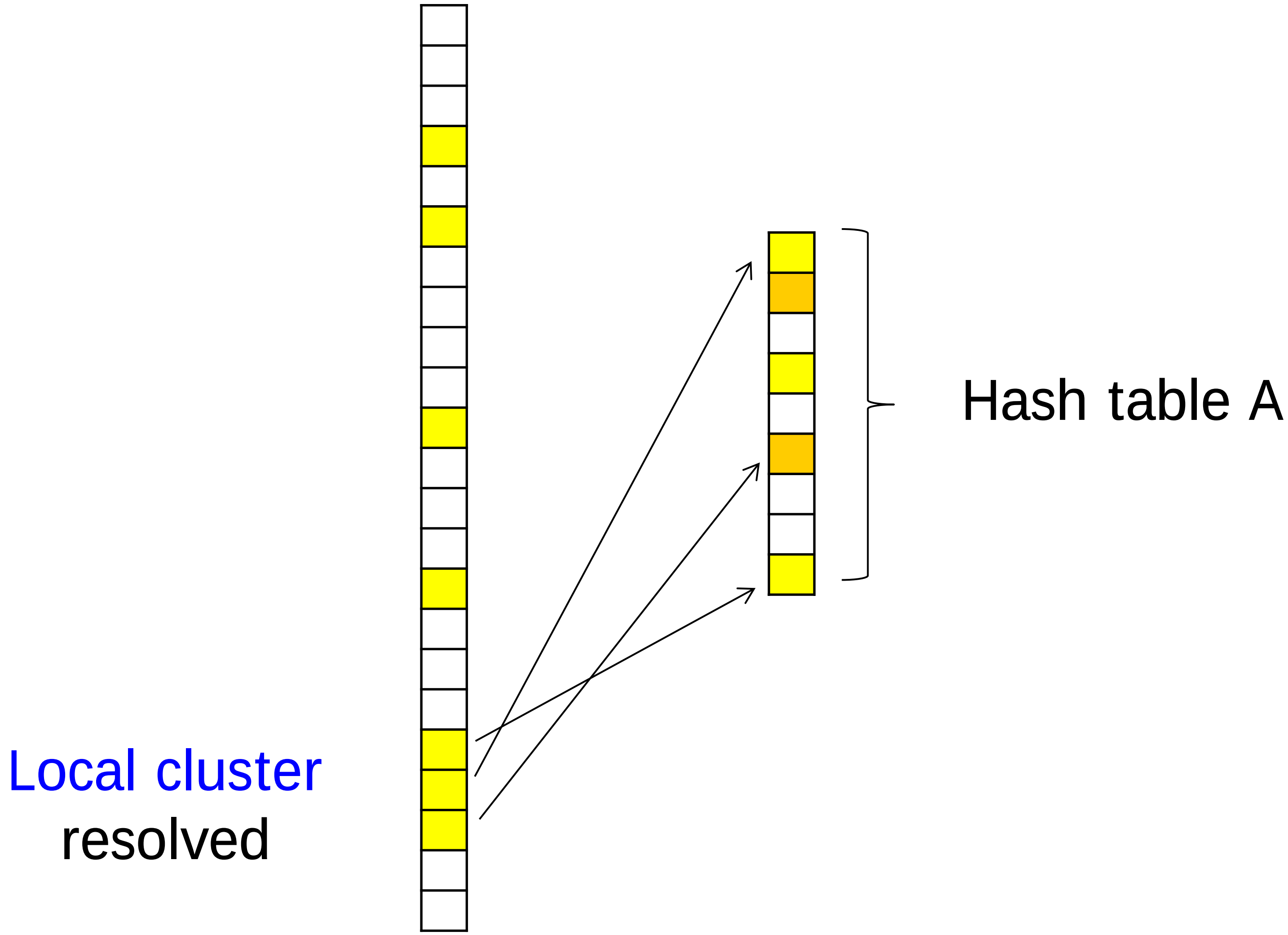
Illustration



Illustration



Illustration



Topics

- We want hash functions with as few collisions as possible
  - Without knowing S, but knowing U
- Hash functions should be computed quickly
  - No option: 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: n should be as small as possible
- Note: Order-preserving hash functions are rare
  - Hashing is bad for range queries

Example

- We usually have $a >> |S|$ yet $a << |U|$
- If k is a number (or can be turned into a number): A simple and surprisingly good hash function:
  $h(k) := k \bmod a$ for $a=|A|$ 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 a positions in A with equal probability
- Given |S| and a – how big are the chances to produce collisions?

Two Cakes a Day?

- Group 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 piece of cake on one day?
- Birthday paradox
  - Obviously, there are 365 chances to eat two pieces
  - 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

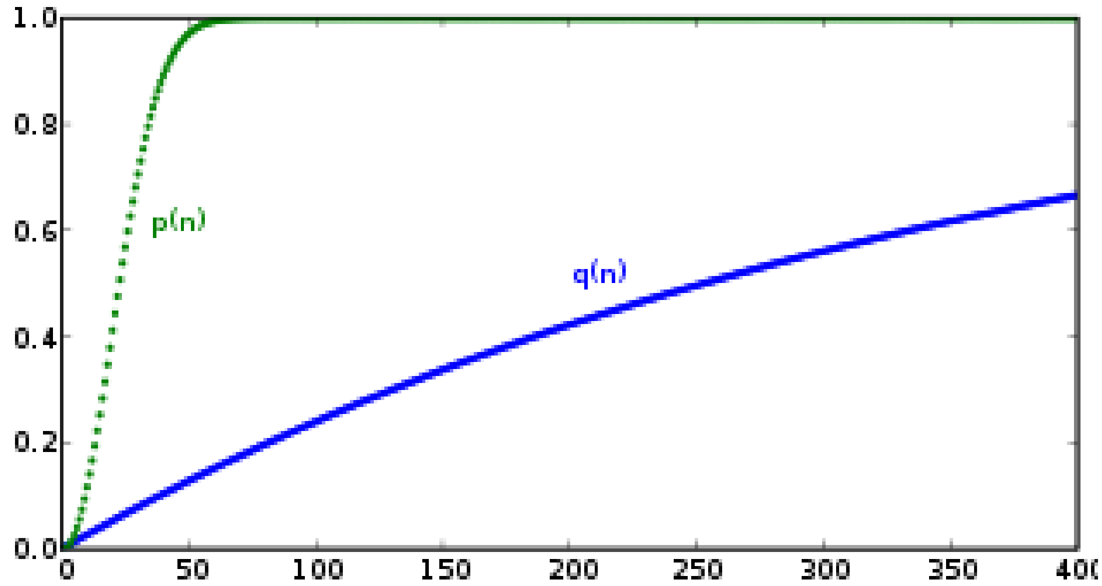
- Abstract formulation: Urn with 365 balls
  - We draw 32 times and place the ball back after drawing
  - What is the probability p(32, 365) to draw any ball at least twice?
- Complement of the chance to draw no ball more than once
  - p(32, 365) = 1 – q(32,365)
- This means we only draw different balls
- We draw a first ball. Then
  - Chance that the second is different is 364/365
  - Chance that the 3rd is different from 1st and 2nd (which must be different) 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) means p(n,365)
- q(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

Three Fundamental Methods

- Overflow hashing: Collisions are stored outside A
  - We need additional storage
  - Solves the problem of A having a fixed size 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

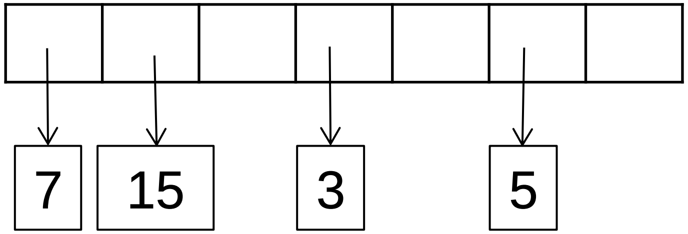
Content of this Lecture

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

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 (key, p), where p is a pointer to a list storing all keys except the first one mapped to i
    - 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 (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

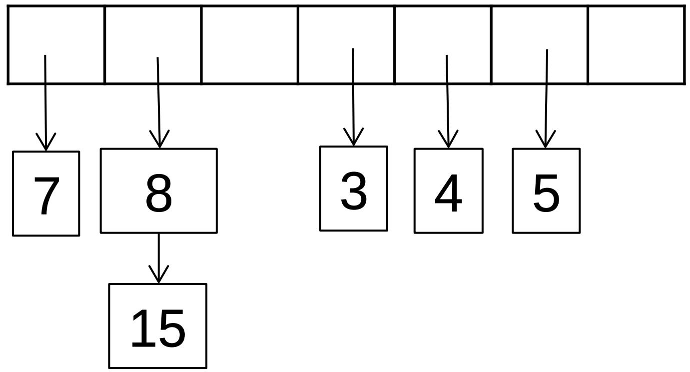
3

7

8

4

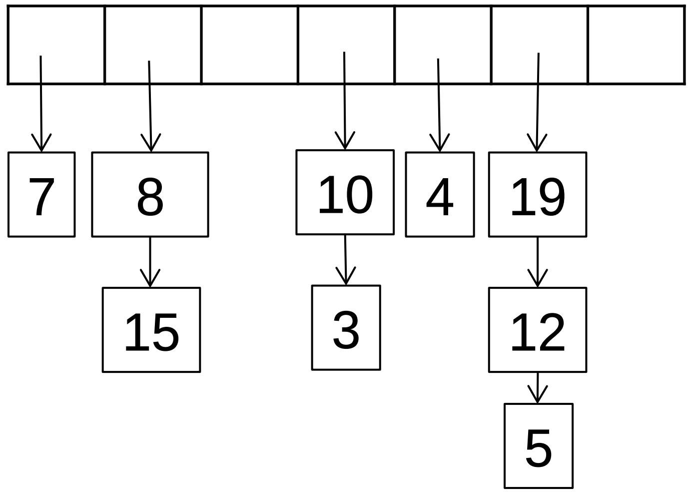
12



- 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: O(a+n) (a lists, n list elements)
- Time (worst-case)
    - Insert: O(1)
    - Search: O(n) – worst case, all keys map to the same cell
    - Delete: O(n) – we first need to search

Average Case Complexities

- Assume h uniform
- After having inserted n values, every overflow list has $\alpha \sim n/a$ elements
  - α is also called the fill degree of the hash table
- How long does the $n+1^{st}$ operation take on average?
  - insert: O(1)
  - search: If $k \in L$: $\alpha/2$ comparisons; else α comparisons
  - delete: Same as search

Improvement

- We may keep every overflow list sorted
  - If stored in a (dynamic) array, binsearch requires $\log(\alpha)$
  - 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
  - If we first have many inserts (build dictionary), then mostly searches, it may pay off to first build unsorted overflows and then sort all overflow lists in a separate phase
- We may use – a second (smaller) hash table with a different hash function
  - Especially if some overflow lists grow very large
  - See Double Hashing

But ...

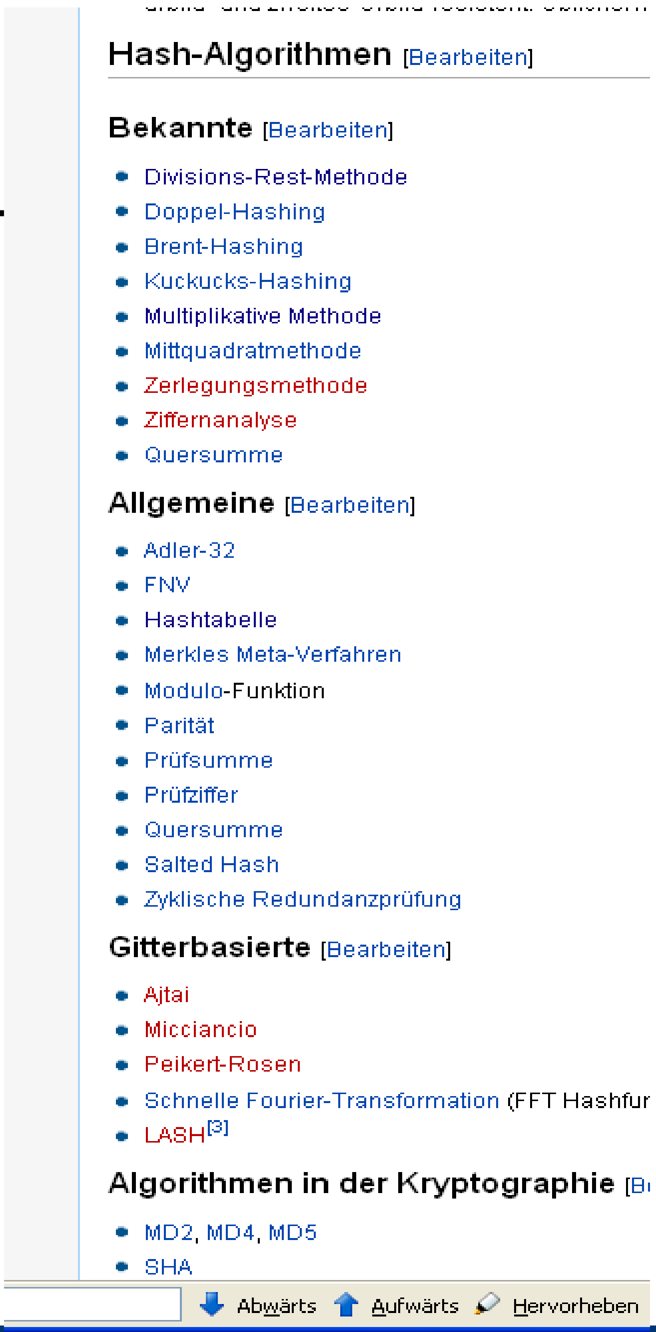
- Searching with $\sim\alpha/2$ comparisons on average doesn't seem very attractive
- But: One typically uses hashing in cases where α is small
  - Usually, $\alpha<1$ – search on average takes only constant time
  - $1\leq\alpha\leq 10$ – search takes only ~5 comparisons
- For instance, let |S|=n=10.000.000 and a=1.000.000
  - Hash table: ~5 comparisons
  - Sorted list: log(1E7)~23 comparisons
- But: In many situations values in S are highly skewed; 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

- Recall Requirements
  - Should be computed quickly
  - Should spread keys equally over A even if local clusters exist
  - Should reach 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)



Why Prime?

- We want hash functions that use the entire key
- Empirical observation from many examples
  - Often keys have an internal structure
    - key= leftstr(firstName,3)+leftstr(lastName, 3)+year(birthday)+sex
  - Think of representation of k as bitstring
  - If a is even, then h(k) is even iff k is even
    - Males get 50%, females get 50% of A – no adaptation
  - If $a=2^i$, h(k) only uses last i bits of any key
    - Which usually are not equally distributed
  - ...
  - a being prime is often a good idea

Other Hash Functions

- "Multiplikative Methode": h(k) = floor(a*(k*x–floor(k*x)))
  - Multiply k with x, remove the integer part, multiply with a and round to the next smaller integer value
  - x: any real number; best distribution on average for $x=(1+\sqrt{5})/2$ - Goldene 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()

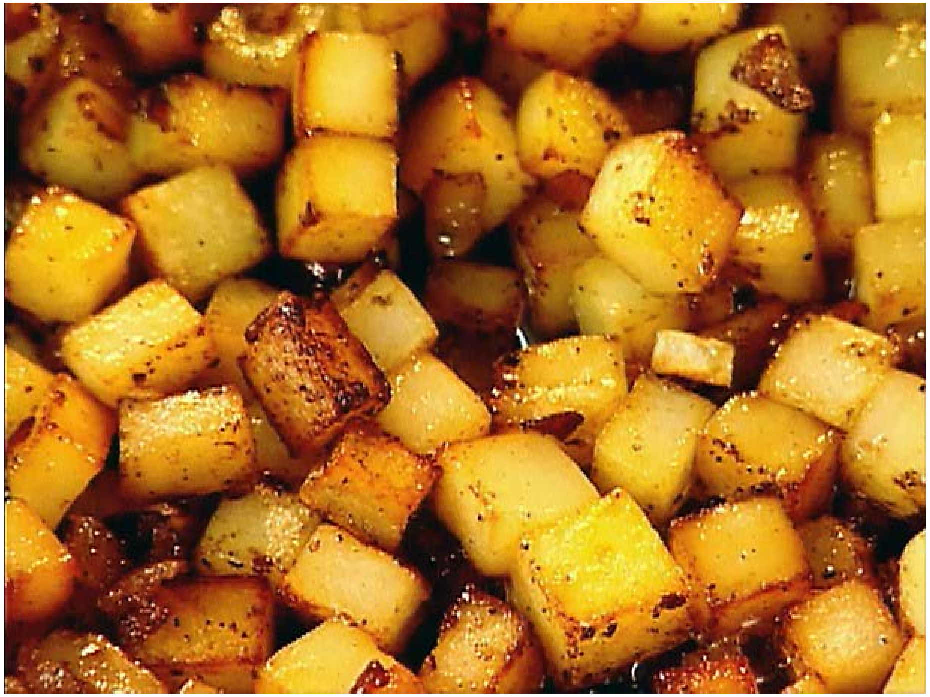
```
/** * Returns a hash code for this string. The hash code for a
* <code>String</code> object is computed as
* <blockquote><pre>
* s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
* </pre></blockquote>
* using <code>int</code> arithmetic, where <code>s[i]</code> is the
* <i>i</i>th character of the string, <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) in BC / AC, but worst case O(n)
- No preparatory work
- App/domain specific hash functions and strategies
- Additional 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?
- 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
- 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 Filter: Simple

- Create a bitarray A with |A|=a=1E6*32
  - We fully exploit our memory; A is always kept in memory
- Choose a uniform hash functions h
- Initialize A (offline): ∀k∈S: A[h(k)]=1
- Searching a given k (online)
  - ∀j: Test A[h(k)] in memory
  - If A[h(k)]=0, we know that k∉S (with 0 IO)
  - If A[h(k)]=1, we need to search k in S (e.g., binsearch)
- We can also use the complement of A (depending on our expectations): Will we have more searches for values in S than for values not 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 a given k (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 (e.g., binsearch)

Analysis

- Assume $k \notin S$
  - Let denote C_n the cost of such a (negative) search
  - We only access disk if all $A[h_j(k)]=1$ by chance – 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[hj(k)]=1 but we don't know if this is by chance of not
  - Thus, $C_p = 20$
    - If S is kept sorted on disk

Chances for a False Positive

- For one $k \in S$ and one hash function, the chance for a given position in A to be 0 is 1-1/a
- For j hash functions, chance that all remain 0 is $(1-1/a)^j$
- For j hash functions and n values, the chance to remain 0 is $q=(1-1/a)^{j*n}$
- The probability that a given bit is set to 1 is 1-q
- Now let's look at a search for key k, which tests j bits
- Chance 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