Algorithms and Data Structures

Open Hashing

Ulf Leser

Repetition

- Universe, Hash function, fill degree, synonyms, collision
- Hash tables, in general, provide a way to insert, delete and find in average O(1) time
- So why still use binary trees?
  - Complex operations are expensive (no order on the elements in a hash table); e.g. extract min, max, etc.
  - Fixed storage can be a problem
  - We might/will have collisions
    - One solution: separate/direct chaining
    - Alternative? Today …

Open Hashing

- Open Hashing: Store all values inside hash table A
- General framework
  - No collision: Business as usual
  - Collision: Chose another index and probe again (is it “open”?)
  - As second index might be full as well, probing must be iterated
- Many suggestions on how to chose the next probe index
- In general, we want a strategy (probe sequence) that
  - … ultimately visits any index in A (and few twice before)
  - … is deterministic – when searching, we must follow the same order of indexes (probe sequence) as for inserts

Reaching all Indexes of A

- Definition
  Let A be a hash table, |A|=m, over universe U and h a hash function for U into A. Let I={0, …, m-1}. A probe sequence is a deterministic, surjective function s: UxI→I
- Remarks
  - We use j to denote elements of the sequence: Where to jump after j-1 probes
  - s need not be injective – a probe sequences may cross itself
    - But it is better if it doesn't
  - We typically use s(k, j) = (h(k) – s'(k, j)) mod m for a properly chosen function s'
- Example: s'(k, j) = j ,hence s(k, j) = (h(k)–j) mod m

Searching

```
func int search(k int) {
  j := 0;
  first := h(k);
  repeat
    pos := (first-s'(k, j)) mod m;
    j := j+1;
  until (A[pos]=k) or
        (A[pos]=null) or
        (j=m)
  if (A[pos]=k) then
    return pos;
  else
    return -1;
  end if;
}
```

- Let s'(k, 0) := 0
- We assume that s cycles through all indexes of A
  - In whatever order
- Probe sequences longer than m-1 usually make no sense, as they necessarily look into indexes twice
  - But beware of non-injective functions

Deletions

- Deletions are a problem
  - Assume h(k)= k mod 11 and s(k, j) = (h(k) + 3*j) mod m)

	0	1	2	3	4	5	6	7	8	9	10
ins(1); ins(6)		**1**					**6**				
ins(23)		**1**			**23**		**6**				
ins(12)		**1**			**23**		**6**	**12**			
del(23)		**1**					**6**	**12**			
search(12)		**1**			**?**		**6**	**12**			

Remedies

- Leave a mark (tombstone)
  - During search, jump over tombstones
  - During insert, tombstones may be replaced
- Re-organize list
  - Keep pointer p to index where a key should be deleted
  - Walk to end of probe sequence (first empty entry)
  - Move last non-empty entry to index p
  - Requires to completely run through the probe sequence for every deletion (otherwise only n/2 on average)
  - Not compatible with strategies that keep probe sequences sorted
    - See later

Open versus External collision handling

- Pro
  - We do not need more space than reserved – more predictable
  - A typically is filled more homogeneously – less wasted space
- Contra
  - More complicated
  - Depending on method, we get worse average-case / worst-case complexities for insertion/deletion/sort
    - Especially deletions have overhead
  - A gets full; we cannot go beyond $\alpha=1$
  - If A gets very large, we can elegantly store overflow chains on external memory

Overview

- We will look into three strategies
  - Linear probing: s(k, j) := (h(k) – j) mod m
  - Double hashing: s(k, j) := (h(k) – j*h'(k)) mod m
  - Ordered hashing: Any s; values in probe sequence are kept sorted
- Others
  - Quadratic hashing: s(k, j) := (h(k) – $floor(j/2)^2*(-1)^j$) mod m
    - Less vulnerable to local clustering then linear hashing
  - Uniform hashing: s is a random permutation of I dependent on k
    - High administration overhead, guarantees shortest probe sequences

Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Ordered Hashing

Linear Probing

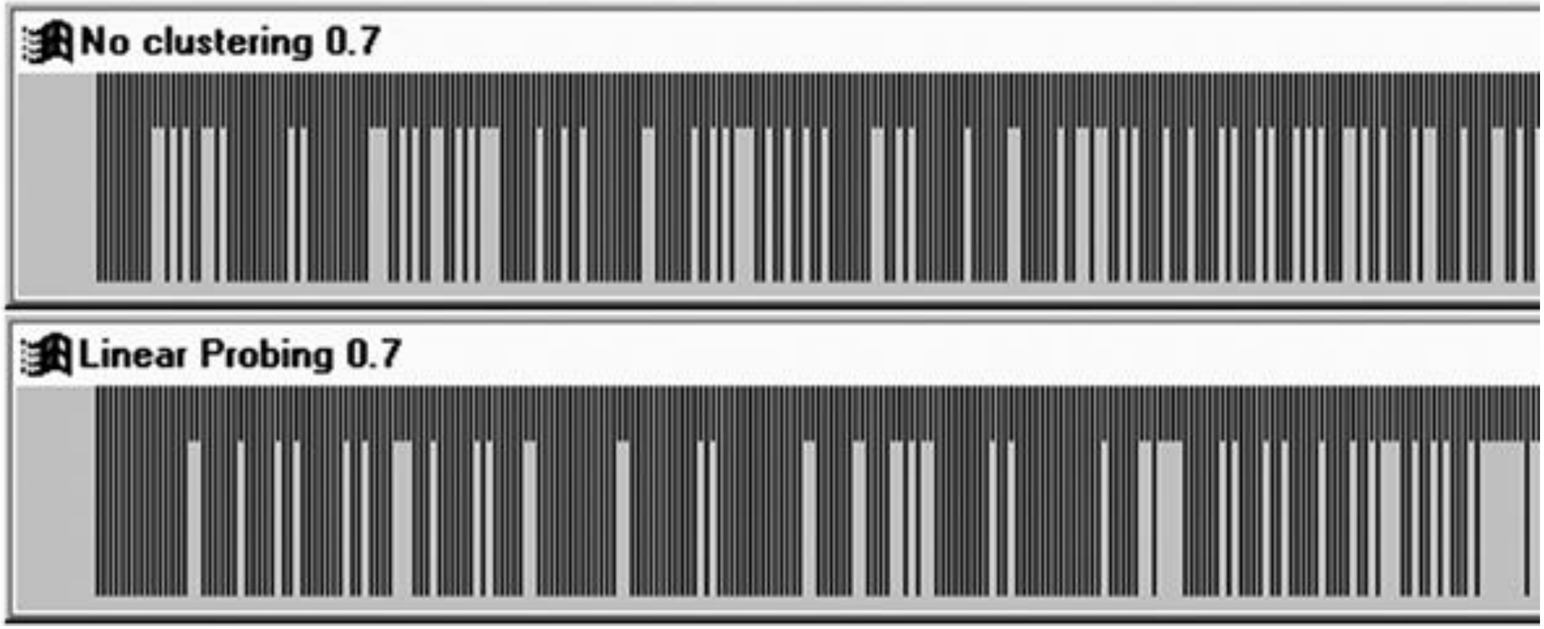
- Probe sequence function: s(k, j) := (h(k) – j) mod m
  - Assume h(k)= k mod 11

	0	1	2	3	4	5	6	7	8	9	10
ins(1); ins(7); ins(13)		**1**	**13**					**7**			
ins(23)	**23**	**1**	**13**					**7**			
ins(12)	**23**	**1**	**13**					**7**			**12**
ins(10)	**23**	**1**	**13**					**7**		**10**	**12**
ins(24)	**23**	**1**	**13**					**7**	**24**	**10**	**12**

Analysis

- The longer a chain …
  - the more different values of h(k) it covers
  - the higher are the chances to produce more collisions
  - the faster it will grow, the faster it will merge with other chains
- Assume an empty position p left of a chain of length n and an empty position q with an empty cell to the right
  - Also assume h is uniform
  - Chances to fill q with next insert: 1/m
  - Chances to fill p with the next insert: n/m
- Linear probing tends to quickly produce long, completely filled stretches of A with high collision probabilities

Analysis



In Numbers

- Scenario: Some inserts, then many searches
  - Expected number of probes per search are most important

erfolgreiche Suche:

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

erfolglose Suche:

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

α	C_n (erfolgreich)	C'_n(erfolglos)
0.50	1.5	2.5
0.90	5.5	50.5
0.95	10.5	200.5
1.00	–	–

Source: S. Albers / [OW93]

Quadratic Hashing

erfolgreiche Suche:

$$C_n \approx 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{(1-\alpha)}\right)$$

erfolglose Suche:

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{(1-\alpha)}\right)$$

α	C_n (erfolgreich)	C'_n (erfolglos)
0.50	1.44	2.19
0.90	2.85	11.40
0.95	3.52	22.05
1.00	-	-

Source: S. Albers / [OW93]

Discussion

- Disadvantage of linear (and quadratic) hashing: Problems with the original hash function h are preserved
  - Probe sequence only depends on h(k), not on k
    - s'(k, j) ignores k
  - All synonyms k, k' will create the same probe sequence
    - Two keys that form a collision are called synonyms
  - Thus, if h tends to generate clusters (or inserted keys are non-uniformly distributed in U), also s tends to generate "clusters" (i.e., sequences filled from multiple keys)

Double Hashing

- Double Hashing: Use a second hash function h'
  - $s(k, j) := (h(k) - j*h'(k)) \bmod m$ (with $h'(k) \neq 0$)
  - Further, we want that $\neg h'(k)|m$ (done if m is prime)
- h' should spread h-synonyms
  - If $h(k)=h(k')$, then hopefully $h'(k) \neq h'(k')$
    - Otherwise, we preserve problems with h
  - Optimal case: h' statistically independent of h, i.e.,

$$p(h(k)=h(k') \wedge h'(k)=h'(k')) = p(h(k)=h(k'))*p(h'(k)=h'(k'))$$

- Example: If $h(k)= k \bmod m$, then $h'(k)=1+k \bmod (m-2)$

Example (Linear Probing produced 9 collisions)

- h(k) = k mod 11;   h‘(k)= 1+ k mod 9

ins(1); ins(7); ins(13)		**1**	**13**					**7**			
ins(23) h(k)=1; h‘(k)=6 s(k, 1)=5		**1**	**13**			**23**		**7**			
ins(12) h(k)=1; h‘(k)=4 s(k, 1)=3		**1**	**13**	**12**		**23**		**7**			
ins(10)		**1**	**13**	**12**		**23**		**7**			**10**
ins(24) h(k)=2; h‘(k)=7 s(k, 1)=5 s(k, 2)=1 s(k, 3)=8		**1**	**13**	**12**		**23**		**7**	**24**		**10**
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

Analysis

- Would need a lengthy proof

$$C'_n \leq \frac{1}{1-\alpha}$$

$$C_n \approx \frac{1}{\alpha} * \ln\left(\frac{1}{(1-\alpha)}\right)$$

α	C_n (erfolgreich)	C'_n(erfolglos)
0.50	1.39	2
0.90	2.56	10
0.95	3.15	20
1.00	-	-

Another Example

	0	1	2	3	4	5	6	7	8	9	10
ins(23); ins(13)		**23**	**13**								
ins(34) h(k)=1; h‘(k)=8 s(k, 1)=7		**23**	**13**					**34**			
ins(12) h(k)=1; h‘(k)=4 s(k, 1)=3		**23**	**13**	**12**				**34**			
ins(10)		**23**	**13**	**12**				**34**			**10**
ins(24) h(k)=2; h‘(k)=7 s(k, 1)=5		**23**	**13**	**12**		**24**		**34**			**10**

Observation

- We change the order of insertions (and nothing else)

ins(34); ins(13)		**34**	**13**								
ins(23) h(k)=1; h‘(k)=6 s(k, 1)=5		**34**	**13**			**23**					
ins(12) h(k)=1; h‘(k)=4 s(k, 1)=3		**34**	**13**	**12**		**23**					
ins(10)		**34**	**13**	**12**		**23**					**10**
ins(24) h(k)=2; h‘(k)=7 s(k, 1)=5 s(k, 2)=1 s(k, 3)=8		**34**	**13**	**12**		**23**			**24**		**10**

Observation

- The number of collisions depends on the order of inserts
  - Because h' spreads h-synonyms differently for different values of k
- We cannot change the order of inserts, but …
- Observe that when we insert k' and there already was a k with h(k)=h(k'), we actually have two choices
  - Until now we always looked for a new place for k'
  - Why not: set A[h(k')]=k' and find a new place for k?
  - If s(k',1) is filled but s(k,1) is free, then the second choice is better
  - Insert is faster, searches will be faster on average

Brent’s Algorithm

- Brent’s algorithm:
  Upon collision, propagate key for which the next index in probe sequence is free; if both are occupied, propagate k’
- Improves only successful searches
  - Otherwise we have to follow the chain to its end anyway
- One can show that the average-case probe length for successful searches now is constant (~2.5 accesses)
  - Even for relatively full tables

Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Ordered Hashing

Idea

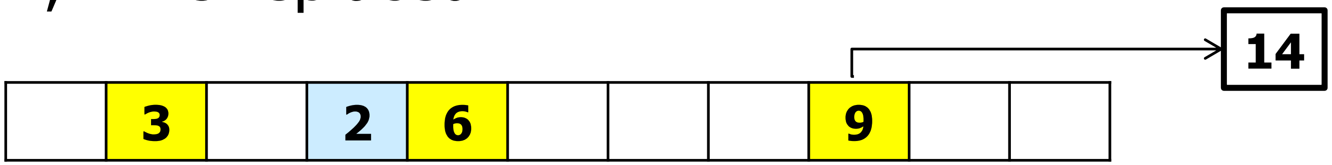
- Can we do something to improve unsuccessful searches?
  - Recall overflow hashing: If we keep the overflow chain sorted, we can stop searching after n/2 comparisons on average
- Transferring this idea: We must keep the keys in any probe sequence ordered
  - We have seen with Brent's algorithm that we have the choice which key to propagate whenever we have a collision
  - Thus, we can also choose to always propagate the smaller of both keys – which generates a sorted probe sequence
- Result: Unsuccessful are as fast as successful searches
  - Note: This trick cannot be combined with Brent's algorithm – conflicting rules

Details

- In Brent‘s algorithm, we only replace a key if we can insert the replaced key directly into A
- Now, we must replace keys even if the next slot in the probe sequence is occupied
  - We run through probe sequence until we meet a key that is larger
  - We insert the new key here
  - All subsequent keys must be replaced (moved in probe sequence)
- Note that this doesn’t make inserts slower than before
  - Without replacement, we would have to search the first free slot
  - Now we replace until the first free slot

Critical Issue

- Imagine ins(6) would first probe position 1, then 4
- Since 6<9, 9 is replaced; imagine the next slot would be 8
- Since 9<14, 14 is replaced



- Problem
  - 14 is not a synonym of 9 – two probe sequences cross each other
  - Thus, we don't know where to move 14 – the next position in general requires to know the "j", i.e., the number of hops that were necessary to get from h(14) to slot 8

Solution

- Ordered hashing only works if we can compute the next offset without knowing j
  - E.g. linear hashing (offset -1) or double hashing (offset –h‘(k))
- But – is the method still correct?
  - Yes (for formal proof, see [OW93])
  - The critical points are where probe sequences cross

Wrap-Up

- In general, complexity of hash-operations depends basically on fill-degree, not on the size of the hash table!
- Open hashing can be a good alternative to overflow hashing even if the fill grade approaches 1
  - Very little average-case cost for look-ups with double hashing and Brent's algorithm or using ordered hashing
    - Depending which types of searchers are more frequent
- Open hashing suffers from having only static place, but guarantees to not request more space once A is allocated
  - Less memory fragmentation
- Open hashing is often used if only few deletions occur

Dynamic Hashing

- Dynamic Hashing adapts the size of the hash table
  - Once fill degree exceeds (falls under) a threshold, increase (decrease) table size
- Used a lot in databases
  - Hash table in main memory, all synonyms in one disc block
  - We increase hash table when synonym block overflows
- Main problem: Avoid rehashing
  - Even if |A| increases, our original hash function (using m) will never address the new slots
  - Undesirable: Create new hash function and rehash all values