Algorithms and Data Structures

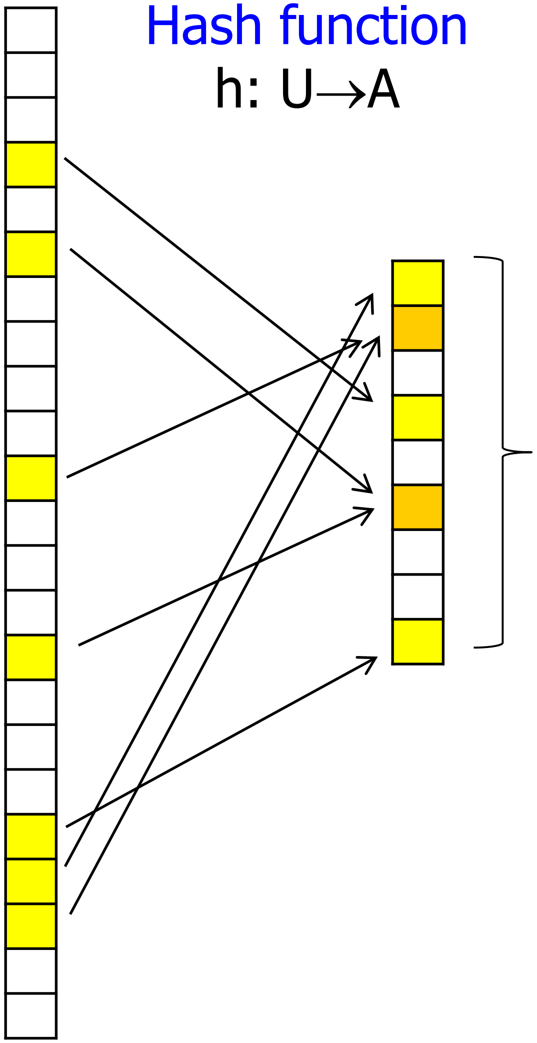
Open Hashing

Ulf Leser

Recall: Hashing

Hash function
h: $U \rightarrow A$

Actual values
of k in S



Hash table A
with collisions

Collision Handling

- 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
- Dynamic hashing: A may grow/shrink
  - Not covered here – see Databases II

Open Hashing

- Open Hashing: Store all values inside hash table A
- Inserting values
  - No collision: As usual
  - Collision: Chose another index and "probe" again
    - And … again …
    - Until free slots is found; otherwise ERROR
- Many suggestions on how to chose the next index to probe
- In general, we want a strategy (probe sequence) that
  - … ultimately visits any index in A (and none 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|=a, over universe U and h a hash function for U into A. Let I={0, …, a-1}. A probe sequence is a deterministic, surjective function s: UxI→I
- Remarks
  - We use j to denote elements of the probe sequence: Where to look next after j-1 probes
  - s need not be injective – a probe sequence may "cross" itself
    - But it is better if it doesn't
- We typically use s(k, j) = (h(k) – s'(k, j)) mod a
  - Of course, s' must be chosen carefully
  - Example: s'(k, j) = j ,hence s(k, j) = (h(k)–j) mod a

Searching

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

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

Example

- A sequence of insertions
  - Assume h(k)= k mod 11 and s(k, j) = (h(k) + 3*j) mod a)

	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**			

But: Deletions

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

	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**			

Remedy

- We can not simply move last element of probe sequence to fill the hole
  - Because we don't know the probe sequence of the deleted element
  - Could be from a cross
- Solution: Leave a mark (tombstone)
  - During search, jump over tombstones
  - During insert, tombstones may be replaced
- Creates longer sequences
- Ultimately, α becomes useless to estimate complexity
- Practical hint: Avoid open hashing when deletions are frequent

Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
    - Brent's Algorithm
  - Ordered Hashing

Open Hashing: Overview

- We will look into three strategies in more detail
  - Linear probing: s(k, j) := (h(k) – j) mod a
  - Double hashing: s(k, j) := (h(k) – j*h'(k)) mod a
  - 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 a
    - Less vulnerable to local clustering then linear hashing
  - Uniform hashing: s is a random permutation of I dependent on k
    - Permutations must be created and stored for each k
    - High administration overhead, guarantees shortest possible probe sequences on average
  - Coalesced hashing
  - …

Linear Probing

- Probe sequence function: s(k, j) := (h(k) – j) mod a
  - 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

Linear Probing

- Probe sequence function: s(k, j) := (h(k) – j) mod a
  - 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**

- Often creates local chains: Full subarrays

Analysis

- The longer a chain …
  - the more different values of h(k) it covers
  - the higher the chances to produce further collisions
  - the faster it grows
- The faster a chain grows, the faster it merges 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/a
  - Chances to fill p with the next insert: (n+1)/a
- Linear probing tends to quickly produce long full stretches of A with high collision probabilities

In Numbers (Derivation of Formulas Skipped)

- Scenario: Some inserts (leading to fill degree α), then many searches
  - Estimating the expected number C_n of probes per search

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) and j, not on k
    - s'(k, j) ignores k
  - All synonyms k, k' will create the same probe sequence
    - Synonym: Two keys that form a collision
  - 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 being filled from multiple keys)

Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Brent's Algorithm
  - Ordered Hashing

Double Hashing

- Double Hashing: Use a second hash function h'
  - s(k, j) := (h(k) – j*h'(k)) mod a (with $h'(k) \neq 0$)
  - Further, we don't want that h'(k)|a (done if a 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.,

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

    - If both are uniform: p(h(k)=h(k')) = p(h'(k)=h'(k')) = 1/a
- Example: If h(k)= k mod a, chose h'(k)=1+k mod (a-2)

Example (Linear Probing produced 9 collisions)

h(k) = k mod 11;   h‘(k)= 1+k mod 9; s(k,j) := (h(k)– j*h’(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) h(k)=1; h‘(k)=6 s(k, 1)=6		1	13				23	7			
ins(12) h(k)=1; h‘(k)=4 s(k, 1)=8		1	13				23	7	12		
ins(10)		1	13				23	7	12		10
ins(24) h(k)=2; h‘(k)=7 s(k, 1)=6 s(k, 2)=10 s(k, 3)=3		1	13	24			23	7	12		10

Analysis

- Please see [OW93]

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

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

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

Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Brent's Algorithm
  - Ordered Hashing

Another Example

$h(k) = k \bmod 11$; $h`(k) = 1 + k \bmod 9$;
$s(k,j) := (h(k) - j * h'(k)) \bmod 11$

	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)=4		**23**	**13**		**34**						
ins(12) h(k)=1; h`(k)=4 s(k, 1)=8		**23**	**13**		**34**				**12**		
ins(10)		**23**	**13**		**34**				**12**		**10**
ins(15) h(k)=4; h`(k)=7 s(k, 1)=8 s(k, 2)=1 s(k, 3)=5		**23**	**13**		**34**	**15**			**12**		**10**

Observation

- Let's change the order of insertions (and nothing else)

ins(23); ins(13)

	23	**13**								

ins(15)
h(k)=4; h‵(k)=6

	23	**13**		**15**						

ins(12)
h(k)=1; h‵(k)=4
s(k, 1)=8

	23	**13**		**15**				**12**		

ins(10)

	23	**13**		**15**				**12**		**10**

ins(34)
h(k)=1; h‵(k)=8
s(k, 1)=4
s(k, 2)=7

	23	**13**		**15**			**34**	**12**		**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' (in step j')
  - Why not: set A[h(k')]=k' and find a new place for k?
    - But for the present key k we don't know j (which step in its sequence?)
    - Use open hashing scheme where next offset is independent of j
    - E.g. linear hash. (constant offset -1), double hash. (offset –h'(k))
    - If s(k',j') is filled but s(k,?) is free, then the second choice is better

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'
  - Brent, R. P. (1973). "Reducing the Retrieval Time of Scatter Storage Techniques.". Communications of the ACM
- Insert is faster, searches will be faster on average
  - Improves only successful searches - otherwise we have to follow the chain to its end anyway
  - Average-case probe length for successful searches becomes almost constant (~2.5 accesses) even for high fill degrees

Content of this Lecture

- Open Hashing
  - Linear Probing
  - Double Hashing
  - Brent's Algorithm
  - Ordered Hashing

Idea

- Can we also improve unsuccessful searches?
  - Recall overflow hashing: If we keep the overflow chain sorted, we can stop searching after $\alpha/2$ comparisons on average
- Transferring this idea: Keep keys sorted in probe sequence
  - We have seen with Brent's algorithm that we have a choice which key to propagate whenever we have a collision
  - Thus, we can also choose to always propagate the larger of both keys – which generates a sorted probe sequence
- Result: Unsuccessful searchers become as fast as successful searches - $\alpha/2$ on average

Details

- In Brent‵s algorithm, we 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 smaller
  - 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
- Careful with tombstones – see [OW93]

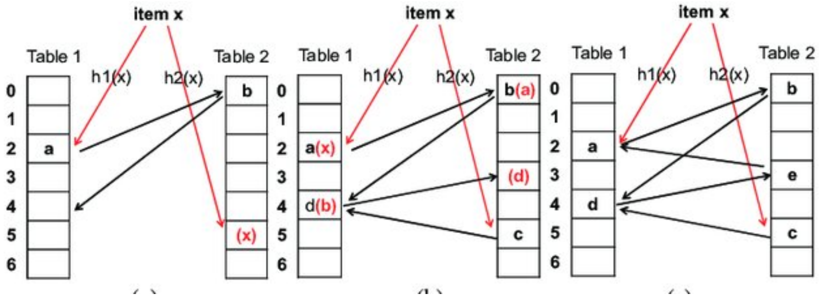
Open versus External collision handling

- Pro Open Hashing
  - We do not need more space than reserved – more predictable
  - A typically is filled more – less wasted space
  - With double hashing and "nice" hash functions and "nice" insert sequences – very low expected work for search / insert
  - Less memory fragmentation, no pointer chasing
  - Many "local" memory accesses (same block) - fast
- Contra
  - More complicated
  - Deletions are a problem
  - A can get full; we cannot go beyond $\alpha=1$
    - That's the ERROR in the insert

Outlook: Cuckoo Hashing

Pagh & Rodler (2004). "Cuckoo hashing." *Journal of Algorithms* 51.2

- Use two tables A_1, A_2 with different hash functions h_1, h_2
- When inserting k
  - if $A_1[h_1[k]]$ free: Insert k into A_1
  - Else if $A_2[h_2[k]]$ free: Insert k into A_2
  - Else get $k' = A_1[h_1[k]]$, set $A_1[h_1[k]] = k$, and try to insert k' into A_2
  - Repeat until free slot found or a cycle is detected
  - If cycle: Rebuilt A_1 and A_2 with new has functions
    - Expensive - but very rare



Li et al. (2019). Multi-copy cuckoo hashing. ICDE

- Properties (assuming certain properties of h_1, h_2)
  - Search and deletion in O(1) – guaranteed
  - Insertion in O(1) on average (amortized analysis)

Exemplary Questions

- Create a hash table step-by-step using open hashing with double probing and hash functions h(k)=k mod 13 and h'(k)=3+k mod 9 when inserting keys 17,12,4,1,36,25,6
- Use the same list for creating a hash table with double hashing and Brent's algorithm
- Use the same list for creating a hash table with ordered linear probing (linear probing such that the probe sequences are ordered).
- Analyze the WC complexity of searching key k in a hash table with direct chaining using a sorted linked list when (a) k is in A; (b) k is not in A.