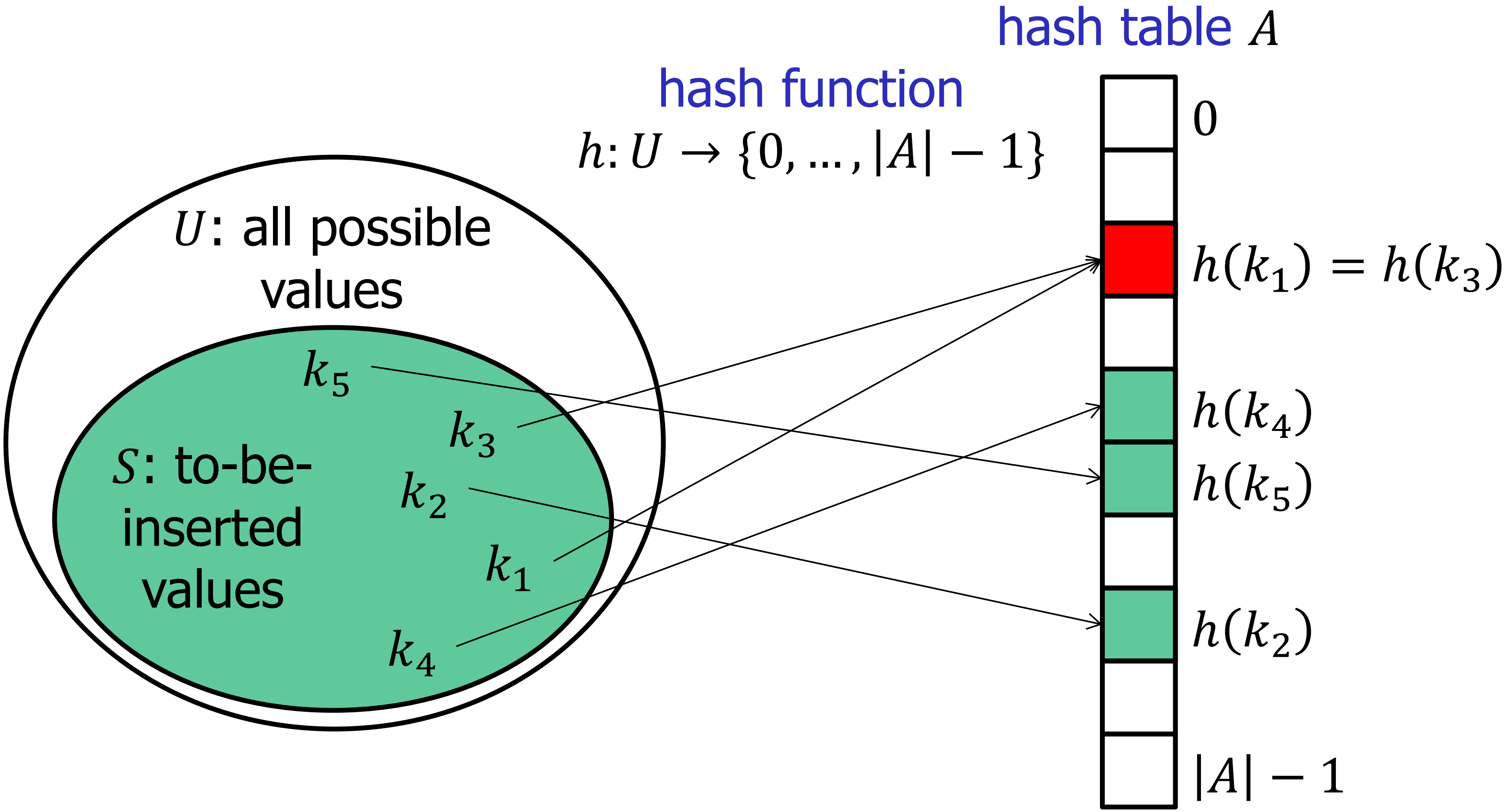
Algorithms and Data Structures

Open Hashing

Marc Bux, Ulf Leser

Recall: Hashing



Recall: Collision Handling

- hash table
  - data structure
  - average-case complexity $O(1)$ for search, insert, delete
  - (assuming a uniform hash function & sufficient remaining space)
- last week: overflow hashing
  - collisions are stored outside A
  - we need additional storage
  - solves the problem of A having a fixed size
- today: open hashing
  - collisions are managed inside A
  - no additional storage
  - $|A|$ is upper bound to the amount of data that can be stored

Content of this Lecture

1. Open Hashing
   a) Linear Probing
   b) Double Hashing
   c) Ordered Hashing

Content of this Lecture

1. Open Hashing
   a) Linear Probing
   b) Double Hashing
   c) Ordered Hashing

Open Hashing

- open hashing: store all values inside hash table A [OW93]
  - also known as: open addressing, closed hashing, …
- inserting values
  - no collision: business as usual
  - collision: choose another index and probe again
  - as second index might be full as well, probing must be iterated
- many suggestions on how to select the next index to probe
- generally, we want a strategy (probe sequence) that
  - … ultimately visits every index in A
  - … rarely (if ever) visits the same index twice
  - … differs from probe sequences for other values
  - … is deterministic, such that we can find our inserted value later

Reaching all Indexes of A

- Definition: Let A with $|A| = m$ be a hash table over universe U. Let $I \coloneqq \{0, \dots, |A| - 1\}$ and let $h: U \rightarrow I$ be a hash function. A probe sequence is a deterministic, surjective function $s: U \times I \rightarrow I$.
- for a given value k, $s(k, i)$ denotes what index to probe next after i unsuccessful probings (starting with $i = 0$)

- we typically use $s(k, i) = \big(h(k) - s'(k, i)\big) \bmod m$ for a properly chosen function s'
- example: $s'(k, i) = i$, hence $s(k, i) = (h(k) - i) \bmod m$
- s need not be injective – a probe sequences may cross itself (but it is better if it doesn't)

Searching

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

- let $s'(k,0) := 0$
- we assume that s probes 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

Deleting

- deletions are a problem
  - assume $h(k) = k \bmod 11$ and $s(k,i) = (h(k) + 3 * i) \bmod m$

	0	1	2	3	4	5	6	7	8	9	10
insert(1); insert(6)		**1**					**6**				
insert(23)		**1**			**23**		**6**				
insert(12)		**1**			**23**		**6**	**12**			
delete(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
  - disadvantage: likelihood of collisions increases beyond fill degree α
- re-organize table
  - keep pointer to index i where a key should be deleted
  - walk to end of probe sequence (first empty entry)
  - move last non-empty entry to index i
  - disadvantages:
    - requires to always probe until the end of the probe sequence
    - not compatible with strategies in which $s'(k,i)$ depends on k
    - not compatible with strategies that keep probe sequences sorted (see later)

Open Hashing versus Overflow Hashing

- pro
  - we do not need more space than reserved – more predictable
  - A typically is filled more homogeneously – less wasted space

- contra
  - more complicated
  - generally, we get worse WC/AC complexities
    - tombstone collisions during search & deletion
    - necessity to walk to the end of probe sequences during deletion
  - A can get full; we cannot go beyond fill degree $\alpha = 1$

Open Hashing: Probing Strategies

- we will look into three strategies
  1. linear probing: $s(k,i) \coloneqq (h(k) - i) \bmod m$
  2. double hashing: $s(k,i) \coloneqq \big(h(k) - \ i \cdot h'(k)\big) \bmod m$
  3. ordered hashing: any s; values in probe sequence are kept sorted
- many other strategies exist:
  - quadratic probing: $s(k,i) \coloneqq \left(h(k) - \left\lceil \frac{i}{2} \right\rceil^2 \cdot (-1)^i\right) \bmod m$
    - $s(k,0) = h(k)$, $s(k,1) = h(k) + 1$, $s(k,2) = h(k) - 1$, $s(k,2) = h(k) + 4$
    - less vulnerable to local clustering than linear probing
  - uniform hashing: s is a random permutation of I dependent on k
    - high administration overhead, guarantees shortest probe sequences
  - coalesced hashing: s arbitrary; entries are linked by add. pointers
    - like overflow hashing, but overflow chains are in A
    - needs additional space for links

Content of this Lecture

1. Open Hashing
    a) Linear Probing
    b) Double Hashing
    c) Ordered Hashing

Linear Probing

- probe sequence function: $s(k, i) \coloneqq (h(k) - i) \bmod m$
  - assume $h(k) \coloneqq k \bmod 11$

	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 the chances to produce more collisions, and,
  - thus, the faster it grows
- the faster it 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 right of a chain
  - also assume h is uniform
  - probability to fill q with next insert: $\frac{1}{m}$
  - probability to fill p with the next insert: $\frac{n+1}{m}$
- linear probing tends to quickly produce long, completely filled stretches of A with high collision probabilities

In Numbers

- scenario:
  - some inserts, then many searches
  - expected number of probings per search are most important
- successful search: $C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
- unsuccessful search: $C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$
- (derivation of furmulae omitted)

α	C_n	C'_n
0.5	1.5	2.5
0.9	5.5	50.5
0.95	10.5	200.5
1	–	–

Source: [OW93]

Quadratic Hashing (in Comparison)

- successful search: $C_n \approx 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{1-\alpha}\right)$
- unsuccessful search: $C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$

α	C_n	C'_n
0.5	1.44	2.19
0.9	2.85	11.4
0.95	3.52	22.05
1	–	–

Source: [OW93]

Discussion

- advantages of linear (and quadratic) hashing:
  - straightforward to implement
  - table can be re-organized after deletion (see slide 10)

- disadvantage of linear (and quadratic) hashing: problems with the original hash function h are preserved
  - $s'(k,j)$ ignores k, i.e., probe sequence only depends on $h(k)$, not on k
  - all synonyms k, k' with $h(k) = h(k')$ will create the same probe sequence (two keys that form a collision are called synonyms)
  - if h tends to generate clusters (or inserted keys are non-uniformly distributed in U), s also tends to generate clusters

Content of this Lecture

1. Open Hashing
   a) Linear Probing
   b) Double Hashing
   c) Ordered Hashing

Double Hashing

- idea: use a second hash function h'
- probe sequence function:
  - $s(k,i) := \big(h(k) - i \cdot h'(k)\big) \bmod m$ with $h'(k) \neq 0$
  - also, we don't want that $h'(k)|m$ (given 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.,

$$p\Big(\big(h(k) = h(k')\big) \wedge \big(h'(k) = h'(k')\big)\Big) =$$
$$p\big(h(k) = h(k')\big) \cdot p\big(h'(k) = h'(k')\big)$$

  - if both are uniform: $p\big(h(k) = h(k')\big) = p\big(h'(k) = h'(k')\big) = \frac{1}{m}$
- example: $h(k) = k \bmod m$, $h'(k) = 1 + k \bmod (m-2)$

Example (Linear Probing produced 9 collisions)

$$h(k) = k \bmod 11,\ h'(k) = 1 + k \bmod 9,\ s(k,i) \coloneqq \big(h(k) - i \cdot h'(k)\big) \bmod 11$$

	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

- successful search: $C_n \leq \frac{1}{1-\alpha}$
- unsuccessful search: $C'_n \approx \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$

α	C_n	C'_n
0.5	1.39	2
0.9	2.56	10
0.95	3.15	20
1	–	–

Source: [OW93]

Another Example

$$h(k) = k \bmod 11,\ h'(k) = 1 + k \bmod 9,\ s(k,i) \coloneqq \big(h(k) - i \cdot h'(k)\big) \bmod 11$$

	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

we change the order of insertions (and nothing else)

	0	1	2	3	4	5	6	7	8	9	10
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 insertions
  - reason: 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
  - so far, 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, R. P. (1973). "Reducing the Retrieval Time of Scatter Storage Techniques." CACM

- Brent's algorithm:
  - when inserting k, upon collision with k', propagate key for which the next index in probe sequence is free
  - if the next indexes for k and k' are both occupied, propagate k
- improves successful searches
  - for unsuccessful searches, we have to follow the chain to its end anyway
- the average case probe length for successful searches is now < 2.5 (even for relatively full tables)

Content of this Lecture

1. Open Hashing
   a) Linear Probing
   b) Double Hashing
   c) Ordered Hashing

Motivation

- can we do something to improve unsuccessful searches?
  - recall overflow hashing: if we keep the overflow list sorted, we can stop searching after $\frac{\alpha}{2}$ comparisons on average

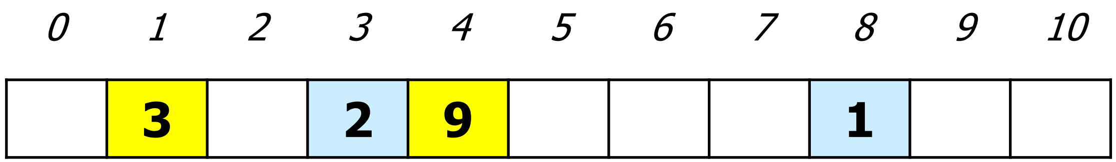
- transferring this idea: keep keys sorted in probe sequence of open hashing
  - we have seen with Brent's algorithm that we have the choice which key to propagate whenever we have a collision
  - thus, we can also choose to always propagate the smaller of both keys
  - this generates a sorted probe sequence
- result: unsuccessful searches are as fast as successful searches

Details

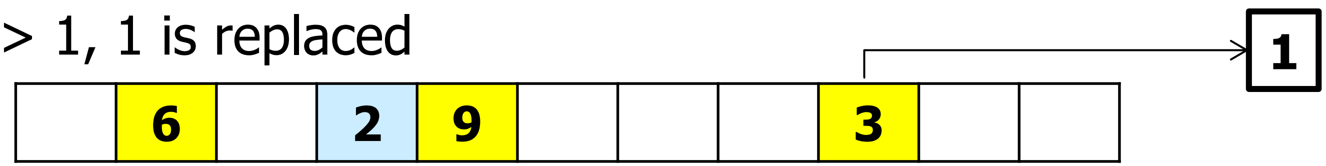
- in Brent‘s algorithm, we only replace a key k' if we can insert the replaced key k' directly into A
- now, we must replace keys even if the next slot in the probe sequence is occupied
  - we walk through probe sequence until we meet a key that is smaller
  - we insert the new key here
  - all subsequent keys must be replaced (moved in probe sequence)

- 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 4, then 1
- since 6 > 3, 3 is replaced; imagine the next slot would be 8
- since 3 > 1, 1 is replaced



- problem
  - 1 is not a synonym of 3 – two probe sequences cross each other
  - thus, we don’t know where to move 1
- ordered hashing only works if we can compute the next position without knowing i (i.e., the number of probings that were necessary to get from $h(1)$ to slot 8)
  - e.g., linear hashing (offset -1) or double hashing (offset $-h'(k)$)

Wrap-Up

- open hashing can be a good alternative to overflow hashing even if the fill grade approaches 1
  - very little average case cost for searching using double hashing and Brent's algorithm or ordered hashing
  - average case complexity of search depends on its success
- open hashing suffers from having only static space, but guarantees to not request more space once A is allocated
  - less memory fragmentation

Exemplary Questions

1. Create a hash table of size 13 step by step using open hashing with double probing and hash functions $h(k) = k \bmod 13$ and $h'(k) = 1 + k \bmod 11$ when inserting keys 17, 12, 4, 1, 36, 25, 6.
2. Create the hash table as in 1. using Brent's algorithm for collision resolution.
3. Create the hash table as in 1. using ordered hashing.
4. What are the advantages / disadvantages of using open hashing over using overflow hashing?
5. For collision resolution in open hashing, what are the advantages / disadvantages of using double hashing over using quadratic hashing?