Data structures and algorithms

Part 11

Searching, mainly via Hash tables

Petr Felkel

26.1.2007

Topics

Searching

Hashing

- Hash function
- Resolving collisions
  - Hashing with chaining
  - Open addressing
    - Linear Probing
    - Double hashing

Dictionary

Many applications require:

- dynamic set
- with operations: Search, Insert, Delete

= dictionary

Ex. Table of symbols in a compiler

identifier	type	address
sum	int	0xFFFFDC09
...	...	...

Searching

Comparing the keys $\Omega(\log n)$

– Found when key of data item = searched key

– Ex: Sequential search, BST,...

associative

Indexing by the key (direct access) $\Theta(1)$

–The key value is the memory address of the item

– keys scope ~ indices scope

Hashing **on average** $\Theta(1)$

– The item address is computed using the key

address search

Hashing

= tradeoff between the speed and the memory usage

- ∞ time - sequential search
- ∞ memory - direct access (indexing by the key)

- few memory and few time:
  - Hash table
  - table size influences the search time

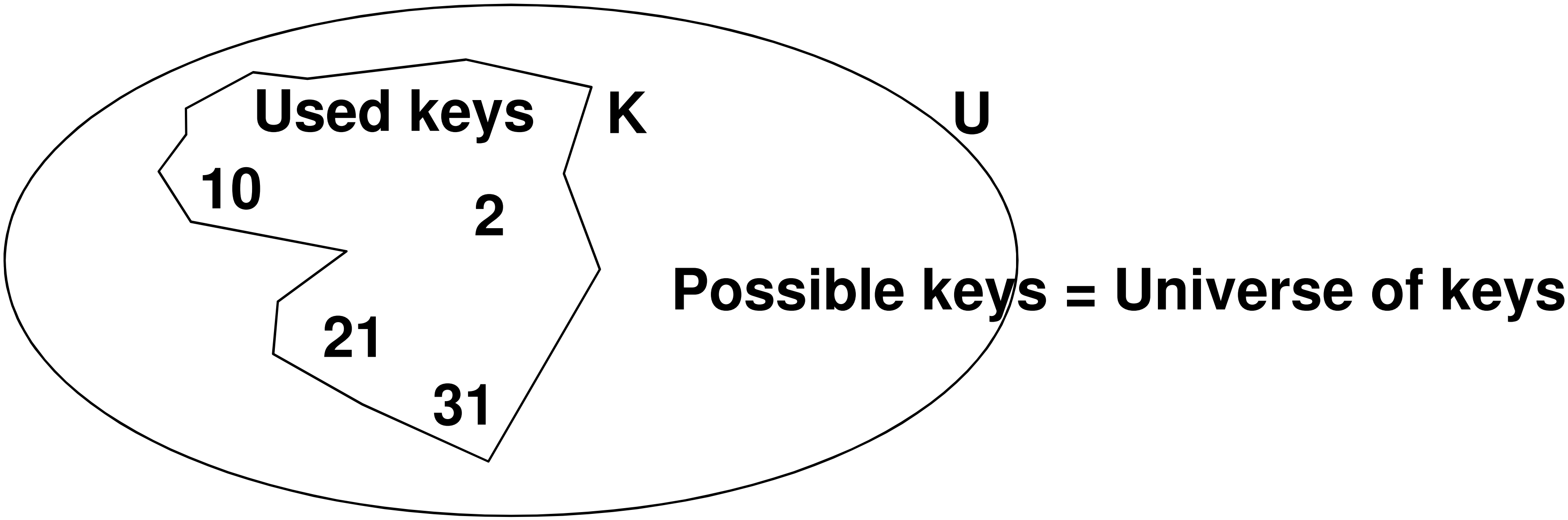
Hashing

Constant expected time of operations *search* and *insert* !!!

Tradeoff:

- Operation time ~ key length
- Hashing is not suitable for operations *select a subset* and *sort*

Rozptylování

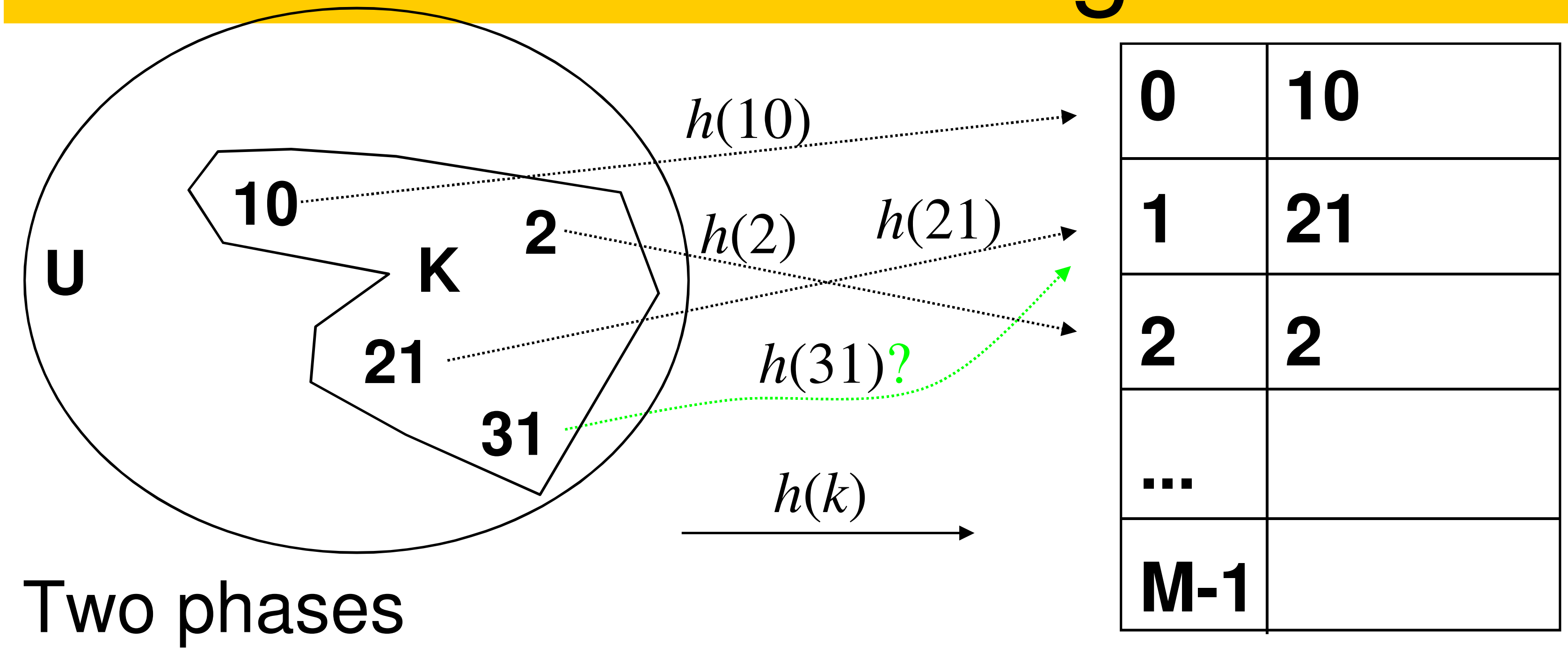


Hashing applicable when |K| << |U|

K Set of really used keys

U Universe of keys -- all possible (thinkable) keys, even if unused

hashing



Two phases

1. Compute hash function *h*(*k*)
   (*h*(*k*) produces item address based on the key value
2. Resolving collisions

h(31) ..... collision: index 1 is already occupied

1. Compute hash function $h(k)$

Hash function $h(k)$

Maps

set of keys $K_j \in U$

into the interval of addresses $A = <a_{min}, a_{max}>$,

usually into $<0,M-1>$

Synonyms: $k_1 \neq k_2$, $h(k_1) = h(k_2)$

= collision!!

Hash function $h(k)$

Depends very strongly on key properties and the memory representation of the keys

Ideally:

- simple calculation -- fast
- approximates well a random distribution
- exploits uniformly address space in memory
- generates minimum number of collisions
- Therefore: It uses all components of a key

Hash function $h(k)$ - examples

Examples of $h(k)$ for different key types

- Real (float) values
- integers
- bit strings
- strings

Hash function *h*(*k*) - examples

Real values from <0, 1>

– multiplicative: **h(k,M) = round(k * M)**

(does not separate the clusters of similar values)

M = table size

Hash function *h*(*k*) - examples

For *w-bit integers*

- multiplicative: (M is a prime)
  - **h(k,M) = round(k /2^{w} * M)**
- modular:
  - **h(k,M) = k % M**
- combined:
  - **h(k,M) = round(c * k) % M, c ∈ <0,1>**
  - **h(k,M) = (int)(0.616161 * (float) k) % M**
  - **h(k,M) = (16161 * (unsigned) k) % M**

Hash functions *h*(*k*) - examples

Fast but depends a lot on keys representation:

h(k) = k & (M-1) **for M = 2^x (not a prime) ,**
& = bit product

Hash function *h*(*k*) - examples

For *strings:*

```
int hash( char *k, int M )
{
    int h = 0, a = 127;
    for( ; *k != 0; k++ )
        h = ( a * h + *k ) % M;
    return h;
}
```

Horner scheme:

$$k_2 * a^2 + k_1 * a^1 + k_0 * a^0 =$$

$$((k_2 * a) + k_1) * a + k_0$$

Hash function *h*(*k*) - examples

For strings: (pseudo-) randomized

```
int hash( char *k, int M )
{   int h = 0, a = 31415; b = 27183;
    for( ; *k != 0; k++, a = a*b % (M-1) )
        h = ( a * h + *k ) % M;
    return h;
}
```

Universal hash function

- collision probability = 1/M
- different random constants applied to different positions in the string

Hash function $h(k)$ - flaws

Frequent flaw: $h(k)$ returns often the same value

- wrong type conversion
- works but generates many similar addresses
- therefore it produces many collisions

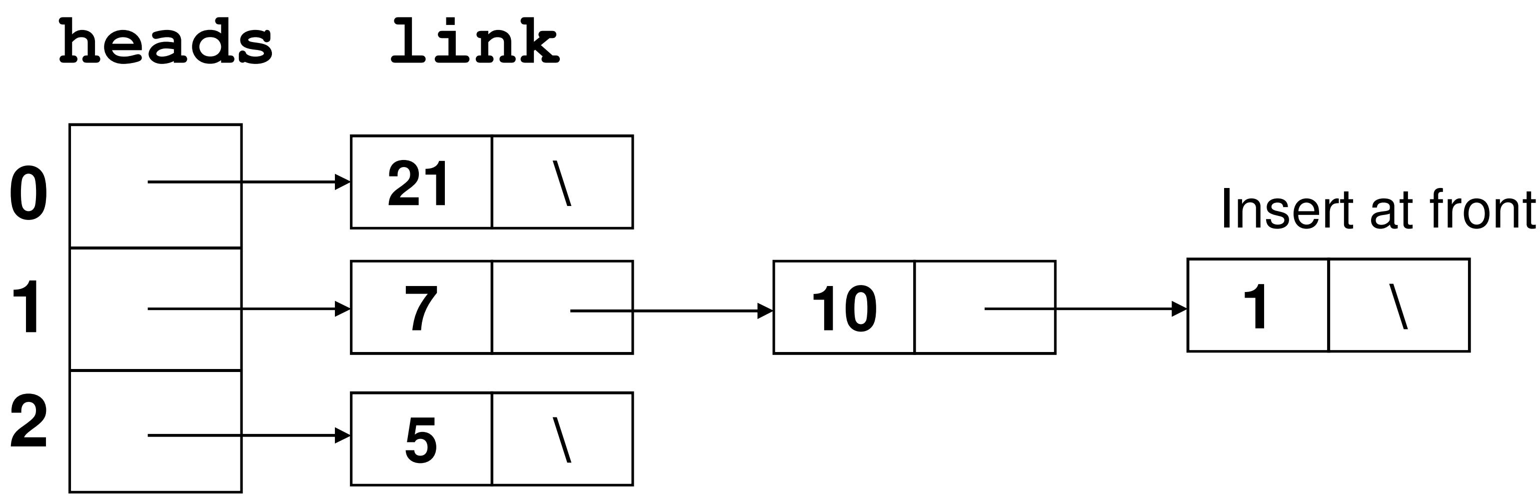
=> *the application is extremely slow*

2. Collision resolving

a) Chaining 1/5

$h(k) = k \bmod 3$

sequence: 1, 5, 21, 10, 7



lists of synonyms

a) Chaining 2/5

```
private:
  link* heads; int N,M;   [Sedgewick]

public:
  init( int maxN )          // initialization
  {
    N=0;                    // No.nodes
    M = maxN / 5;           // table size
    heads = new link[M];    // table with pointers
    for( int i = 0; i < M; i++ )
       heads[i] = null;
  }
  ...
```

a) Chaining 3/5

```
Item search( Key k )
{
  return searchList( heads[hash(k, M)], k );
}

void insert( Item item )                    // insert at front
{
  int i = hash( item.key(), M );
  heads[i] = new node( item, heads[i] );
  N++;
}
```

a) Chaining 4/5

synonyms chain has ideally length

$\alpha = n/m, \alpha > 1$ (load factoor)

(n = no of elems, m = table size, m<n)

Highly improbable outcome

			on average
Insert	**$I(n) = t_{hash} + t_{link} = O(1)$**		
Search	**$Q(n) = t_{hash} + t_{search}$**		
	$= t_{hash} + t_c{}^{*} n/(2m)$	**$= O(n)$**	**$O(1 + \alpha)$**
Delete	**$D(n) = t_{hash} + t_{search} + t_{link}$**	**$= O(n)$**	**$O(1 + \alpha)$**

for small α (and big m) it is close to O(1) !!!

for big α (and small m) m-times faster than sequential search

a) Chaining 5/5

Practical use:

choose $m = n/5 \dots n/10$ => load factor $\alpha = 5 \dots 10$

- sequential search in the chain is fast
- not many unused table slots

Pros & cons:

+ exact value of *n* needs not to be known in advance
– needs dynamic memory allocation
– needs additional memory for chain (list) pointers

b) Open-address hashing

The approximate number of elements is known

No additional pointers

=> Use 1D array

Hash function *h*(*k*) is tied with collision resolving

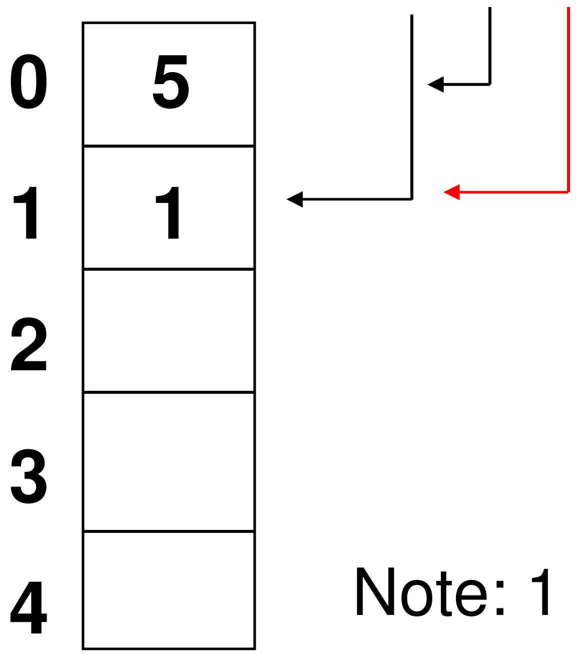
1. linear probing
2. double hashing

0	**5**
1	**1**
2	**21**
3	**10**
4	

b) Open-address hashing

$h(k) = k \bmod 5$ ($h(k) = k \bmod m$, m is array size)

sequence: 1, 5, 21, 10, 7



Problem:

collision - 1 already occupies the space for 21

1. linear probing
2. double hashing

Note: 1 and 21 are synonyms. The position is often occupied by a key which is not a synonym. Collision does not distinguish between synonyms and non-synonyms.

Probing

= check what is in the table at the position given by the hash function

- search hit = key found
- search miss = empty position, key not found
- else = position occupied by another key, continue searching

b) Open-address hashing

Methods of collision resolving

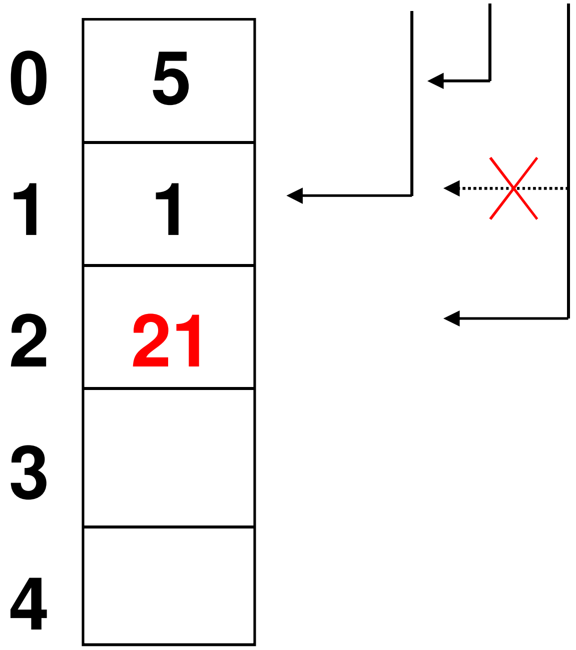
b1) Linear probing

b2) Double hashing

b1) Linear probing

$h(k)$ = [(k mod 5) + i] mod 5 = (k + i) mod 5

sequence: 1, 5, 21, 10, 7



collision!

=> 1. linear probing

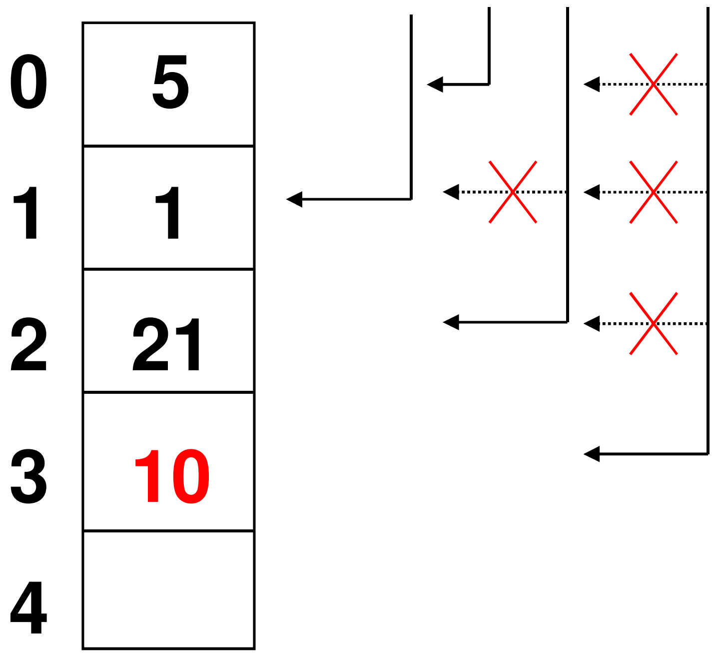
move forward

by one position (i++ => i = 1)

b1) Linear probing

h(*k*) = (k + i) mod 5

sequence: 1, 5, 21, 10, 7



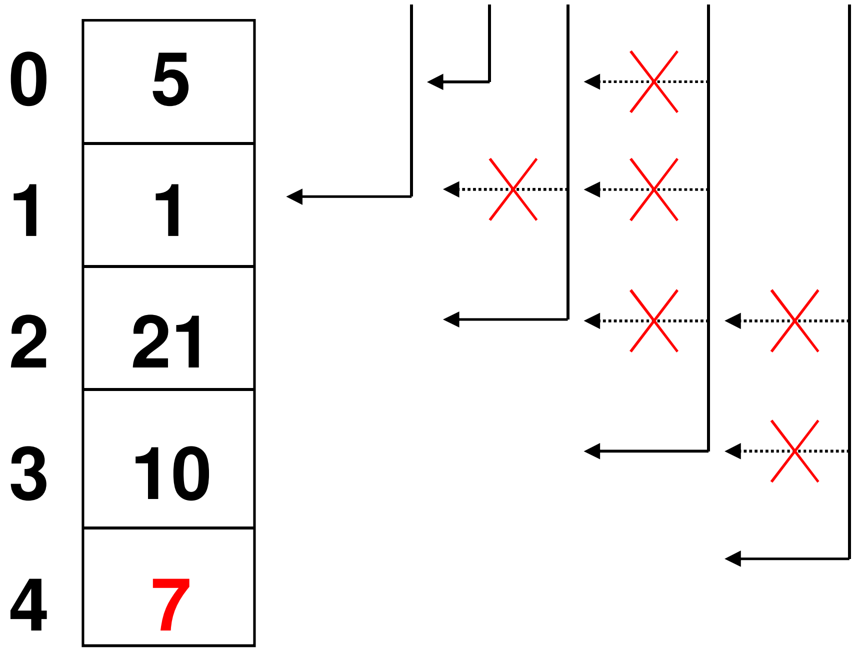
1. collision with 5 - move on
2. collision with1 - move on
3. collision with 21 - move on

Inserted 3 positions further in the table (i = 3)

b1) Linear probing

$h(k) = (k + i) \bmod 5$

sequence: 1, 5, 21, 10, 7



1. collision with 21 (i++)
2. collision with 10 (i++)

Inserted 3 positions further in the table (i = 2)

b1) Linear probing

$h(k) = (k + i) \bmod 5$

sequence: 1, 5, 21, 10, 7

0	**5**	i = 0
1	**1**	i = 0
2	**21**	i = 1
3	**10**	i = 3
4	**7**	i = 2

b1) Linear probing

```
private:
  Item *st; int N,M;  [Sedgewick]
  Item nullItem;
public:
  init( int maxN )        // initialization
  {
    N=0;                  // Number of stored items
    M = 2*maxN;           // load_factor < 1/2
    st = new Item[M];
    for( int i = 0; i < M; i++ )
        st[i] = nullItem;
  }...
```

b1) Linear probing

```
void insert( Item item )
{
  int i = hash( item.key(), M );

  while( !st[i].null() )
      i = (i+1) % M; // Linear probing

  st[i] = item;
  N++;
}
```

b1) Linear probing

```
Item search( Key k )
{
  int i = hash( k, M );

  while( !st[i].null() ) { // !cluster end
                           // sentinel
     if( k == st[i].key() )
          return st[i];
     else
          i = (i+1) % M; // Linear probing
    }
     return nullItem;
}
```

b) Open-address hashing

Methods of collision resolving

b1) Linear probing

b2) Double hashing

b2) Double hashing

Hash function $h(k) = [h_1(k) + i.h_2(k)] \bmod m$

$h_1(k) = k \bmod m$ // initial position

$h_2(k) = 1 + (k \bmod m')$ // offset

Both depend on k

=>

Each key has different probe sequence

m = prime number or m = power of 2

m' = slightly less m' = odd

If d = greatest common divisor => search 1/d slots only

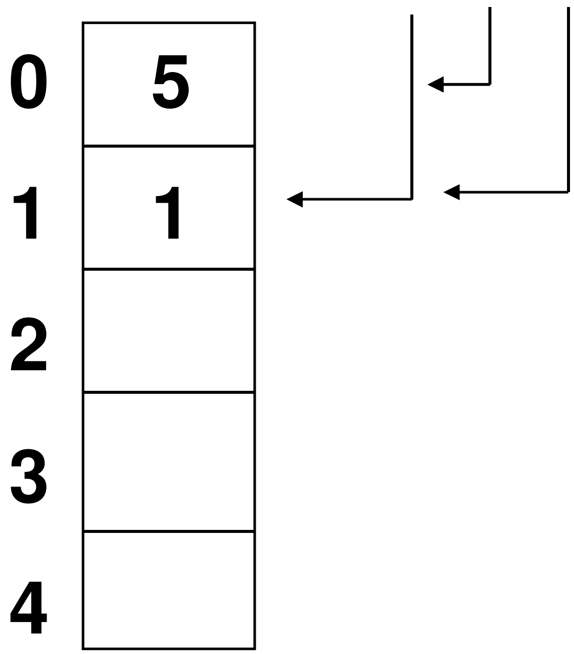
Ex: $k = 123456$, $m = 701$, $m' = 700$

$h_1(k) = 80$, $h_2(k) = 257$ Starts at 80, and every 257 % 701

b2) Double hashing

$h(k) = k \bmod 5$

sequence: 1, 5, 20, 25, 18



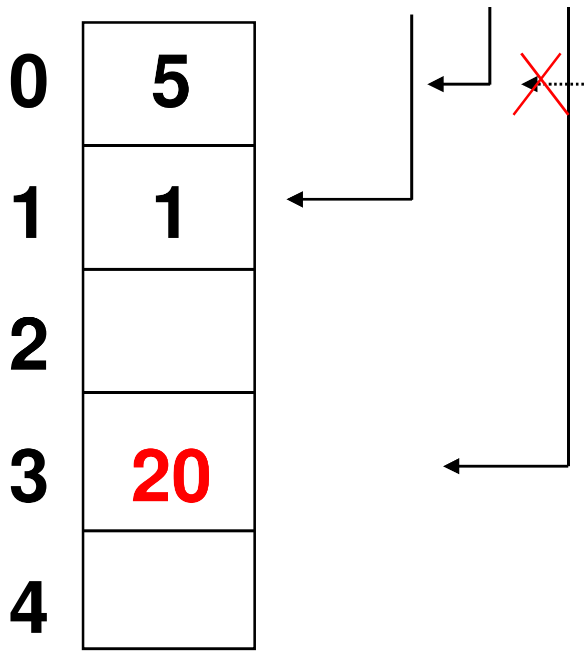
collision

=> 2. double hashing

b2) Double hashing

$h(k)$ = [(k mod 5) + i.h_2(k)] mod 5, $h_2(k)$ = 1 + k mod 3

sequence: 1, 5, 20, 25, 18



collision,

$h_2(20)$ = 1 + 20 mod 3 = 3,

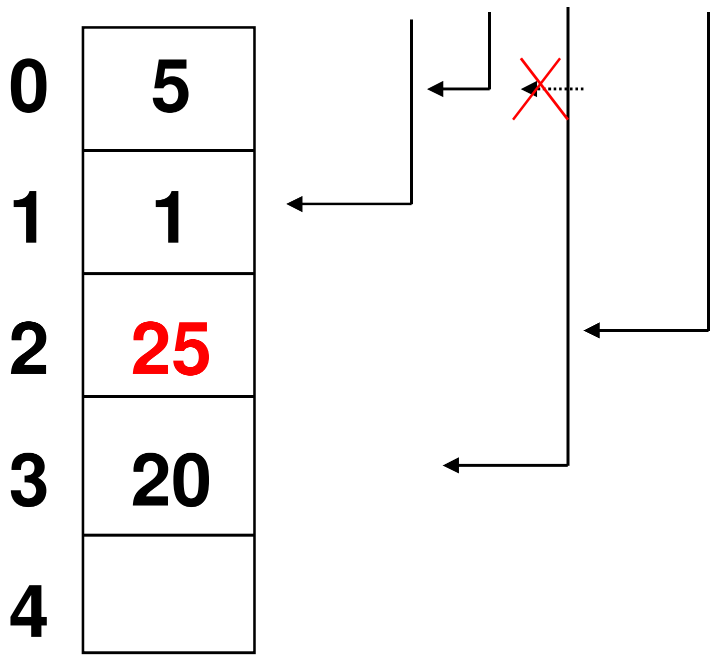
store 20 at position

0 + 3

b2) Double hashing

$h(k) = [(k \bmod 5) + i.h_2(k)] \bmod 5$, $h_2(k) = 1 + k \bmod 3$

sequence: 1, 5, 20, 25, 18



collision,

$h_2(25) = 1 + 25 \bmod 3 = 2$,

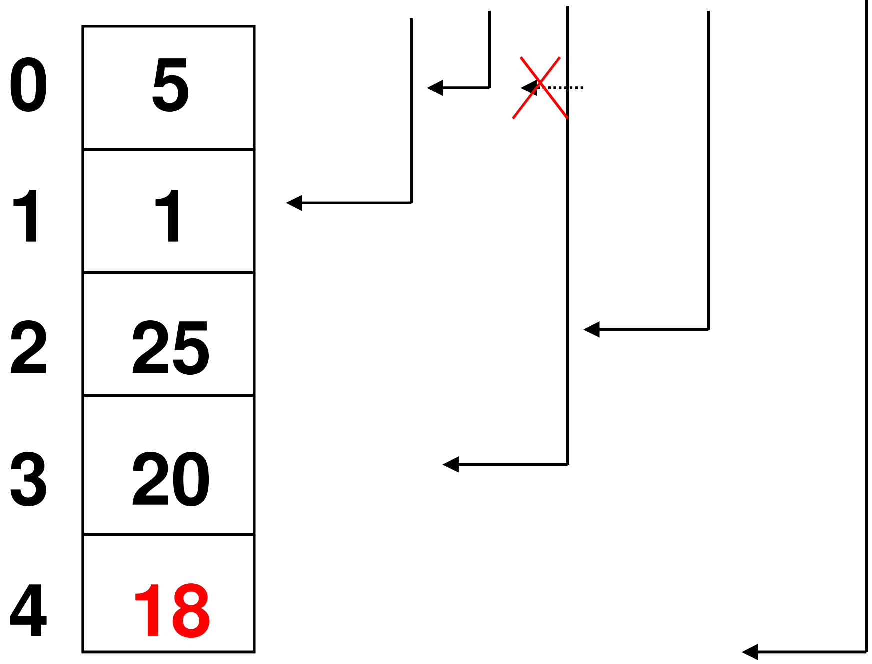
store 25 at position

0 + 2

b2) Double hashing

$h(k) = [(k \bmod 5) + i.h_2(k)] \bmod 5$, $h_2(k) = 1 + k \bmod 3$

sequence: 1, 5, 20, 25, 18



collision,

$h_2(18) = 1 + 18 \bmod 3 = 1$,

store 18 at position

$3 + 1 = 4$

b2) Double hashing

$h(k) = [(k \bmod 5) + i.h_2(k)] \bmod 5$, $h_2(k) = 1 + k \bmod 3$

sequence: 1, 5, 20, 25, 18

0	**5**	i = 0
1	**1**	i = 0
2	**25**	i = 0
3	**20**	i = 1
4	**18**	i = 1

Linear probing x Double hashing

$h(k) = (k + i) \bmod 5$

0	**5**	i = 0
1	**1**	i = 0
2	**21**	i = 1
3	**10**	i = 3 !
4	**7**	i = 2

long clusters

$h(k) = [(k \bmod 5) + i.h_2(k)\ ] \bmod 5,$

$h_2(k) = 1 + k \bmod 3$

0	**5**	i = 0
1	**1**	i = 0
2	**25**	i = 1
3	**20**	i = 1
4	**18**	i = 1

mixed probe sequences

b2) Double hashing

```
void insert( Item item )
{
  Key k = item.key();
  int i = hash( k, M ),
      j = hashTwo( k, M );// different for k1 != k2

  while( !st[i].null() )
     i = (i+j) % M; //Double Hashing

  st[i] = item; N++;
}
```

b2) Double hashing

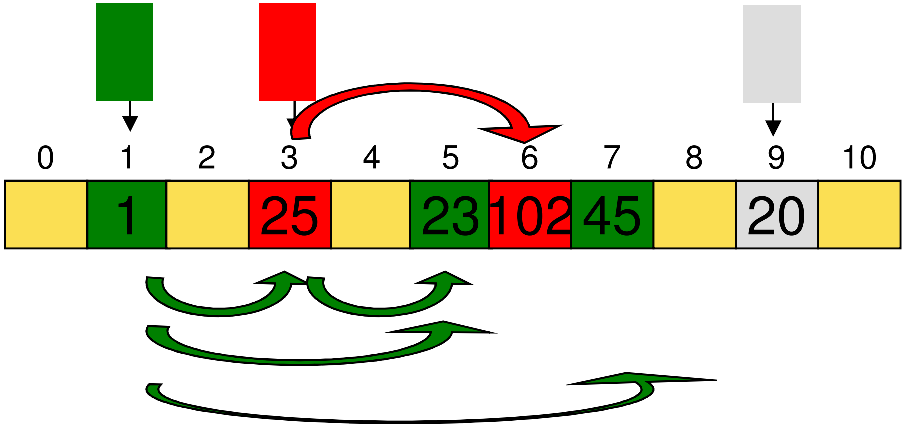
```
Item search( Key k )
{
  int i = hash( k, M ),
      j = hashTwo( k, M );// different for k1 != k2

  while( !st[i].null() )
  {
     if( k == st[i].key() )
          return st[i];
     else
          i = (i+j) % M; // Double Hashing
   }
   return nullItem;
}
```

Double hashing - example

b2) Double hashing $h(k) = [h_1(k) + i.h_2(k)] \bmod m$

Input	$h_1(k)=$ $k\,\%11$	$h_2(k)=$ $1+k\,\%10$	i	$h(k)$
1	1	2	0	1
25	3	6	0	3
23	1	4	0,1	1,5
45	1	6	0,1	1,7
102	3	3	0,1	3,6
20	9	1	0	9



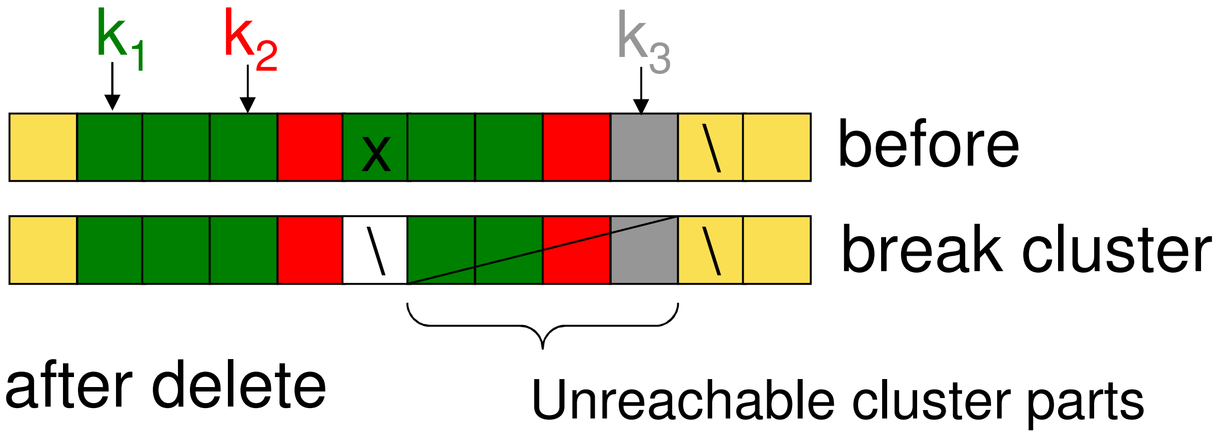
$h_1(k) = k\ \%\ 11$

$h_2(k) = 1 + (k\ \%\ 10)$

Item removal (delete)

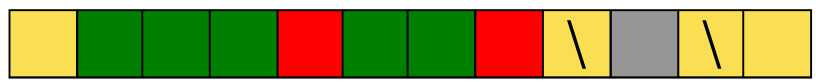
Item ‘x’ removal

- x replaced by null
- null breaks cluster(s) !!!
- => do not leave the hole after delete



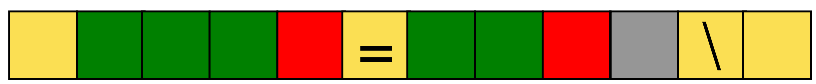
Correction different for linear probing and double hashing

- b1) in linear probing



  => reinsert the items after x (to the first null = to cluster end)

- b2) in double hashing

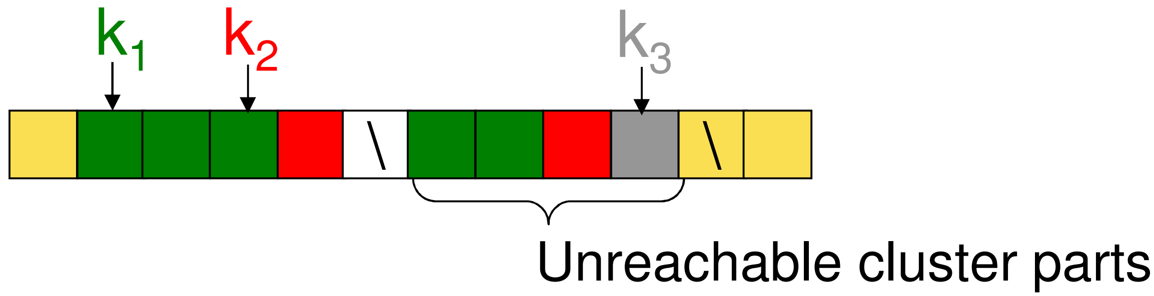


  => fill the hole up by a special sentinel

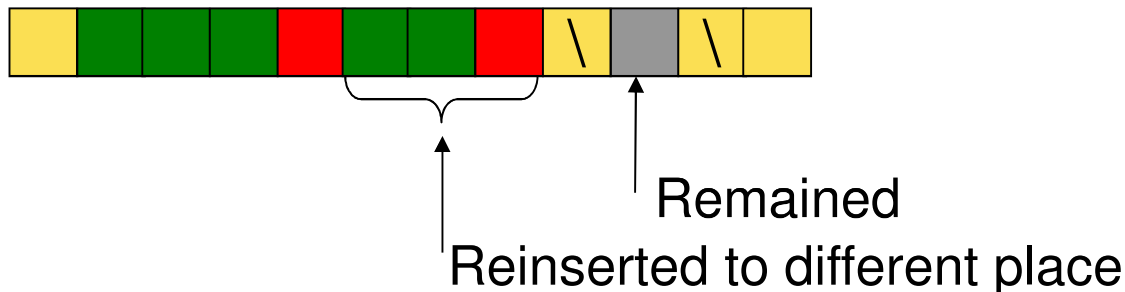
  skipped by search, replaced by insert

Item removal (delete)

b1) in linear probing

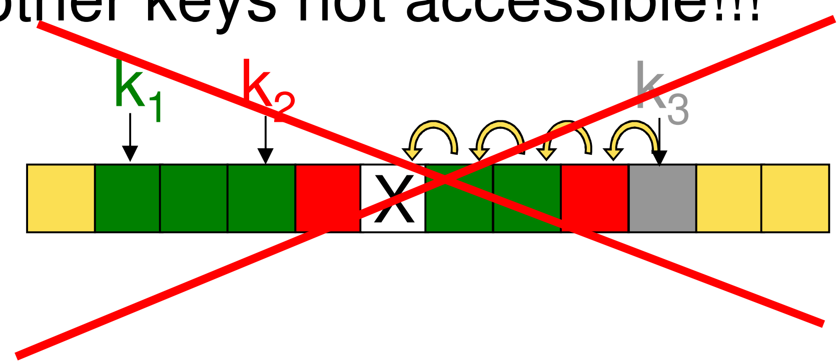


=> reinsert the items behined the cluster break (to the null)



=> avoid simple move of cluster tail

it can make other keys not accessible!!!

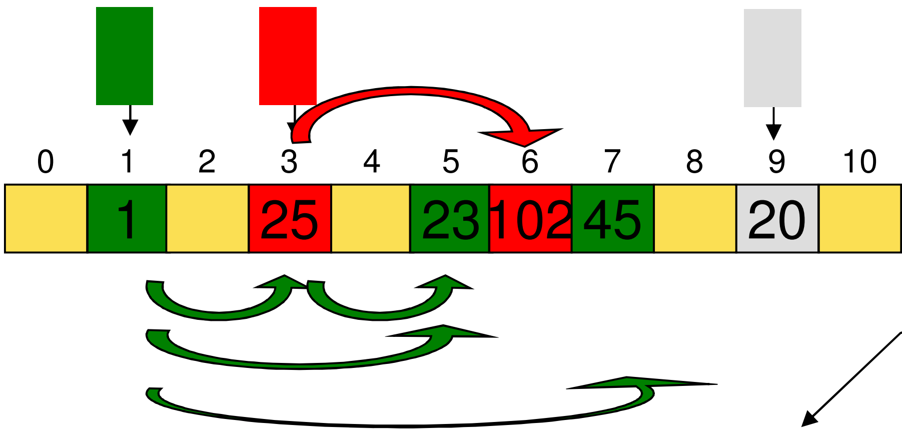


Linear-probing Item Removal

```
// do not leave the hole - can break a cluster
void remove( Item item )
{ Key k = item.key();
  int i = hash( k, M ), j;
  while( !st[i].null() )// find item to remove
      if( item.key() == st[i].key() ) break;
      else i = (i+1) % M;
  if( st[i].null() ) return; // not found
  st[i] = nullItem; N--;      //delete,reinsert
  for(j = i+1; !st[j].null(); j=(j+1)%M, N--)
  {   Item v = st[j]; st[j] = nullItem;
      insert(v);  //reinsert elements after deleted
  }
}
```

Item removal (delete)

b2) Double hashing $h(k) = [h_1(k) + i.h_2(k)]$ mod m

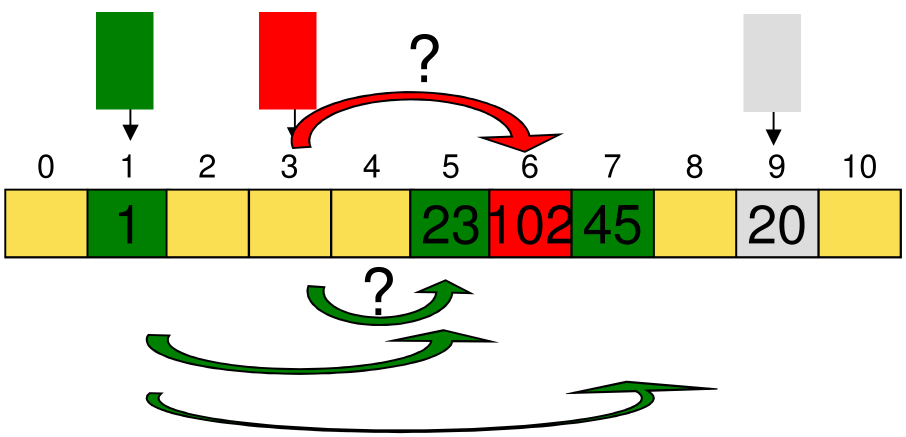


$h_1(k) = k$ % 11

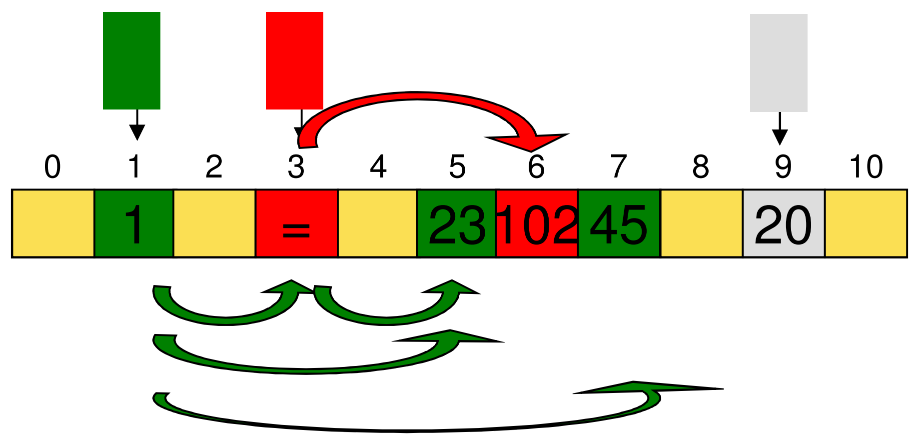
$h_2(k) = 1 + (k$ % 10)

Remove 25

null – breaks paths to 23 and 102



Sentinel is correct



Double-hashing Item Removal

```
// Double Hashing – overlapping search sequences
//      – fill up the hole by sentinel
//   – skipped by search, replaced by insert
void remove( Item item )
{ Key k = item.key();
  int i = hash( k, M ), j = hashTwo( k, M );
  while( !st[i].null() ) // find item to remove
     if( item.key() == st[i].key() ) break;
     else i = (i+j) % M;
  if( st[i].null() ) return; // not found
  st[i] = sentinelItem; N--; // "delete" = replace
}
```

b) Open-addressing hashing

α = ***load factor of the table***

α = **n/m,** $\alpha \in \langle$**0,1**$\rangle$

n = ***number of items in the table***

m = ***table size,*** **m>n**

b) Open-addressing hashing

Average number of probes [Sedgewick]

Linear probing:

Search hits	$0.5 (1 + 1 / (1 - \alpha))$	**found**
Search misses	$0.5 (1 + 1 / (1 - \alpha)^2)$	**not found**

Double hashing:

Search hits	$(1 / \alpha) \ln (1 / (1 - \alpha)) + (1 / \alpha)$
Search misses	$1 / (1 - \alpha)$

$\alpha = n/m, \alpha \in \langle 0,1 \rangle$

b) Expected number of tests

Linear probing:

load factor α	1/2	2/3	3/4	9/10
Search hit	1.5	2.0	3.0	5.5
Search miss	2.5	5.0	8.5	55.5

Double hashing:

load factor α	1/2	2/3	3/4	9/10
Search hit	1.4	1.6	1.8	2.6
Search miss	1.5	2.0	3.0	5.5

Table can be more loaded before the effectivity starts decaying.

Same effectivity can be achieved with smaller table.

References

[Cormen]

Cormen, Leiserson, Rivest: Introduction to Algorithms, Chapter 12, McGraw Hill, 1990