Data structures and algorithms

Part 11

Searching, mainly via Hash tables

Petr Felkel
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

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>int</td>
<td>0xFFFFFDC09</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Searching

Comparing the keys
- Found when key of data item = searched key
- Ex: Sequential search, BST, ...

Ω(\log n)

Indexing by the key (direct access)
- The key value is the memory address of the item
- keys scope ~ indices scope

Θ(1)

Hashing
- The item address is computed using the key

Θ(1)

on average
Hashing

= tradeoff between the speed and the memory usage

- $\infty$ time - sequential search
- $\infty$ 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
Hashing applicable when $|K| << |U|$  

$K$  Set of really used keys  
$U$  Universe of keys -- all possible (thinkable) keys, even if unused
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 $A = \langle a_{\text{min}}, a_{\text{max}} \rangle$,

*usually into* $\langle 0, M-1 \rangle$

**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: $h(k,M) = \text{round}(k \times M)$
  
  (does not separate the clusters of similar values)

  $M = \text{table size}$
Hash function $h(k)$ - examples

For $w$-bit integers

- multiplicative: (M is a prime)
  - $h(k,M) = \text{round}( k / 2^w \times M )$
- modular:
  - $h(k,M) = k \mod M$
- combined:
  - $h(k,M) = \text{round}( c \times k ) \mod M$, $c \in \langle 0,1 \rangle$
  - $h(k,M) = (\text{int})(0.616161 \times (\text{float}) k) \mod M$
  - $h(k,M) = (16161 \times (\text{unsigned}) k) \mod M$
Fast but depends a lot on keys representation:

\[ h(k) = k \& (M-1) \quad \text{for } M = 2^x \text{ (not a prime)}, \]
\[ \& = \text{bit product} \]
Hash function $h(k)$ - examples

For *strings*:

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

**Horner scheme:**  
$$k_2 \cdot a^2 + k_1 \cdot a^1 + k_0 \cdot a^0 =$$
$$((k_2 \cdot a) + k_1)\cdot a + k_0$$
Hash function $h(k)$ - examples

For strings: (pseudo-) randomized

```c
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

$\Rightarrow$ the application is extremely slow
2. Collision resolving
a) Chaining 1/5

$h(k) = k \mod 3$

sequence: 1, 5, 21, 10, 7

heads  link

0      21  \  

1      7  10  1  \  

2      5  \  

lists of synonyms

Insert at front
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

\[ \alpha = \frac{n}{m}, \; \alpha > 1 \quad \text{(load factor)} \]

\( (n = \text{no of elems, } m = \text{table size, } m < n) \)

\[
\begin{align*}
\text{Insert:} & \quad I(n) = t_{\text{hash}} + t_{\text{link}} = O(1) \\
\text{Search:} & \quad Q(n) = t_{\text{hash}} + t_{\text{search}} \\
& = t_{\text{hash}} + t_c \cdot \frac{n}{2m} = O(n) \\
\text{Delete:} & \quad D(n) = t_{\text{hash}} + t_{\text{search}} + t_{\text{link}} = O(n)
\end{align*}
\]

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

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

Highly improbable outcome
a) Chaining 5/5

Practical use:

choose \( m = n/5 \ldots n/10 \) \( \Rightarrow \) load factor \( \alpha = 5 \ldots 10 \)

- 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

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
b) Open-address hashing

\[ h(k) = k \mod 5 \quad \text{(} h(k) = k \mod m, \ m \ \text{is array size)} \]

sequence: 1, 5, 21, 10, 7

Problem:
1. Collision - 1 already occupies the space for 21
2. linear probing
3. 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

\[ h(k) = \left( (k \mod 5) + i \right) \mod 5 = (k + i) \mod 5 \]

sequence: 1, 5, 21, 10, 7

collision!

=> 1. linear probing

move forward

by one position \((i++ \Rightarrow i = 1)\)
b1) Linear probing

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

sequence: 1, 5, 21, 10, 7

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

Inserted 3 positions further in the table (i = 3)
b1) Linear probing

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

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

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

sequence: 1, 5, 21, 10, 7

<table>
<thead>
<tr>
<th>i</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>
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

```c
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 $h(k) = [h_1(k) + i.h_2(k)] \mod m$

$h_1(k) = k \mod m$ \quad // initial position
$h_2(k) = 1 + (k \mod m')$ \quad // offset

$m = \text{prime number or } m = \text{power of 2}$
$m' = \text{slightly less}$ \quad $m' = \text{odd}$

If $d = \text{greatest common divisor} \Rightarrow \text{search } 1/d \text{ slots only}$

Ex: $k = 123456$, $m = 701$, $m' = 700$
$h_1(k) = 80$, $h_2(k) = 257$ \quad \text{Starts at 80, and every } 257 \% 701
b2) Double hashing

\[ h(k) = k \mod 5 \]

sequence: \[ 1, 5, 20, 25, 18 \]

\[ \text{collision} \]

\[ \Rightarrow 2. \text{ double hashing} \]
b2) Double hashing

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

sequence: 1, 5, 20, 25, 18

collision,

\[ h_2(20) = 1 + 20 \mod 3 = 3, \]

store 20 at position 0 + 3
b2) Double hashing

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

sequence: 1, 5, 20, 25, 18

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

collision,
\[ h_2(25) = 1 + 25 \mod 3 = 2, \]
store 25 at position 0 + 2
b2) Double hashing

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

sequence: 1, 5, 20, 25, 18

\[
\begin{array}{c|c}
0 & 5 \\
1 & 1 \\
2 & 25 \\
3 & 20 \\
4 & 18
\end{array}
\]

collision,
\[ h_2(18) = 1 + 18 \mod 3 = 1, \]
store 18 at position
\[ 3 + 1 = 4 \]
b2) Double hashing

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

sequence: \quad 1, 5, 20, 25, 18

\begin{array}{|c|c|c|c|c|}
\hline
i & 0 & 1 & 2 & 3 & 4 \\
\hline
h(k) & 5 & 1 & 25 & 20 & 18 \\
\hline
\end{array}
**Linear probing x Double hashing**

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

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

\[ h_2(k) = 1 + k \mod 3 \]

<table>
<thead>
<tr>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>21</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>18</td>
<td>12</td>
</tr>
</tbody>
</table>

long clusters

mixed probe sequences
b2) Double hashing

```c
void insert( Item item )
{
    Key k = item.key();
    int i = hash( k, M ),
           j = hashTwo( k, M );  // Double Hashing!

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

    st[i] = item; N++;
}
```
b2) Double hashing

```java
Item search( Key k )
{
    int i = hash( k, M ),
    j = hashTwo( k, M ); // Double Hashing

    while( !st[i].null() )
    {
        if( k == st[i].key() )
            return st[i];
        else
            i = (i+j) % M; // Double Hashing
    }
    return nullItem;
}
```
b2) Double hashing $h(k) = [h_1(k) + i.h_2(k)] \mod m$

<table>
<thead>
<tr>
<th>Input</th>
<th>$h_1(k) = k \mod 11$</th>
<th>$h_2(k) = 1 + k \mod 10$</th>
<th>$i$</th>
<th>$h(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>4</td>
<td>0,1</td>
<td>1,5</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>6</td>
<td>0,1</td>
<td>1,7</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>3</td>
<td>0,1</td>
<td>3,6</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

$h_1(k) = k \mod 11$

$h_2(k) = 1 + (k \mod 10)$
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 behind the cluster break (to the null)

=> avoid simple move of cluster tail

it can make other keys not accessible!!!
// 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
    }
}
b2) Double hashing \( h(k) = [h_1(k) + i.h_2(k)] \mod m \)

\[ h_1(k) = k \mod 11 \]
\[ h_2(k) = 1 + (k \mod 10) \]

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

\[ \alpha = \textit{load factor of the table} \]
\[ \alpha = n/m, \quad \alpha \in \langle 0, 1 \rangle \]

\[ n = \textit{number of items in the table} \]
\[ m = \textit{table size}, \quad m > n \]
Average number of probes [Sedgewick]

Linear probing:
- Search hits: \(0.5 \left( 1 + \frac{1}{(1 - \alpha)} \right)\) found
- Search misses: \(0.5 \left( 1 + \frac{1}{(1 - \alpha)^2} \right)\) not found

Double hashing:
- Search hits: \(\frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right) + \frac{1}{\alpha}\)
- Search misses: \(\frac{1}{(1 - \alpha)}\)

\(\alpha = \frac{n}{m}, \alpha \in \langle 0,1 \rangle\)
### b) Expected number of tests

#### Linear probing:

<table>
<thead>
<tr>
<th>load factor $\alpha$</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search hit</strong></td>
<td>1.5</td>
<td>2.0</td>
<td><strong>3.0</strong></td>
<td>5.5</td>
</tr>
<tr>
<td><strong>Search miss</strong></td>
<td>2.5</td>
<td><strong>5.0</strong></td>
<td>8.5</td>
<td>55.5</td>
</tr>
</tbody>
</table>

#### Double hashing:

<table>
<thead>
<tr>
<th>load factor $\alpha$</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search hit</strong></td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td><strong>2.6</strong></td>
</tr>
<tr>
<td><strong>Search miss</strong></td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td><strong>5.5</strong></td>
</tr>
</tbody>
</table>

Table can be more loaded before the effectivity starts decaying. Same effectivity can be achieved with smaller table.
References

[Cormen]