Hashing
Lecture #5 of Algorithms, Data structures and Complexity

Joost-Pieter Katoen, Ed Brinksma
Formal Methods and Tools Group
E-mail: katoen@cs.utwente.nl

September 24, 2002
Overview

⇒ Introduction

• Direct addressing

• Hashing
  – Collision resolution using chaining
  – Complexity analysis of chaining

• Open addressing
  – Probing strategies
  – Complexity analysis of open addressing

• Hash functions
Introduction

• A dictionary ADT stores information that can be retrieved at any time
  – the set of items stored is dynamic
  – items have a key and information associated with that key
  – example: symbol table for a compiler where keys are strings (i.e., identifiers)

• A dictionary $d$ supports the following operations:
  – $\text{search}(k)$ looks up the information stored under key $k$ in $d$
  – $\text{insert}(e)$ stores information object $e$ into $d$
  – $\text{delete}(e)$ deletes information object $e$ from $d$; requires $e$ to be in $d$

• Which data structure is appropriate to implement a dictionary?
  – a heap: insertion and deletion are efficient, but how about search?
  – ordered array/list: insertion is linear in worst case
  – red-black tree: all operations are logarithmic in worst case

under reasonable assumptions a hash table takes $O(1)$ on average for all operations
Overview

- Introduction

  - Direct addressing

- Hashing
  - Collision resolution using chaining
  - Complexity analysis of chaining

- Open addressing
  - Probing strategies
  - Complexity analysis of open addressing

- Hash functions
Direct addressing

- Allocate an array that has a **position for each possible key**

- Each array element contains a pointer to the stored information
  - for simplicity we omit the information associated to keys in this lecture
    ⇒ the techniques and analysis results remain valid

- For universe $U = \{ 0, 1, \ldots, n-1 \}$ of keys we have:
  - a direct-address table $T[0 \ldots n-1]$ with $T[k]$ corresponding to key $k$
  - $\text{search}(k)$: return $T[k]$
  - $\text{insert}(e)$: boils down to $T[\text{key}[e]] = e$
  - $\text{delete}(e)$: simply means $T[\text{key}[e]] = \text{nil}$

- Runtime for each of the operations is $\Theta(1)$ in worst case
Direct addressing

universe of keys

actual keys

key direct address table
Check for duplicates in linear time

assume all elements are positive integers of at most $k$

```c
bool checkDuplicates(int [1..n] E) {
    int [1..k] Count; // direct-address table for $E[i]
    for (i = 1; i ≤ k; i++) Count[i] = 0; // initialize Count
    for (i = 1; i ≤ n; i++) {
        if (Count[E[i]] > 0) return true; // duplicate found
        else Count[E[i]]++; // count occurrence of $E[i$
    }
    return false; // no duplicate found
}
```
Counting sort

assume all elements are positive integers of at most $k$

```c
void countSort(int [1..n] E) {
    int [1..k] Count, int i, j, l = 0;
    for (i = 1; i $\leq$ k; i++) Count[i] = 0;
    for (i = 1; i $\leq$ n; i++) Count[E[i]]++;
    for (i = 1; i $\leq$ n; i++) {
        for (j = Count[i] + l; j $>$ l; j--) E[j] = i;
        l = Count[i] + l;
    }
}
```
Counting sort: example

<table>
<thead>
<tr>
<th>k</th>
<th>Count</th>
<th>input array E</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>2 0 0 1 2 1 1 0 0 1</td>
</tr>
<tr>
<td>after 1 iteration</td>
<td>2</td>
<td>2 0 0 1 2 1 1 0 0 1</td>
</tr>
<tr>
<td>after 2 iterations</td>
<td>2</td>
<td>2 0 0 1 2 1 1 0 0 1</td>
</tr>
<tr>
<td>after 3 iterations</td>
<td>2</td>
<td>2 0 0 1 2 1 1 0 0 1</td>
</tr>
<tr>
<td>after 4 iterations</td>
<td>3</td>
<td>2 0 0 1 2 1 1 0 0 1</td>
</tr>
<tr>
<td>after 5 iterations</td>
<td>5</td>
<td>2 0 0 1 2 1 1 0 0 1</td>
</tr>
</tbody>
</table>
Counting sort

- Note that we now sort with worst-case complexity $\Theta(n)$
  - compare this to the lower-bound of $\Theta(n \cdot \log n)$ that we obtained earlier
  - but this algorithm is incomparable to quicksort, heapsort and the like
  $\Rightarrow$ it is not based on element-wise comparisons, but counts occurrences

- Why does this trick work: exploit direct addressing

- Insertion, deletion and searching takes $\Theta(1)$ in worst case

- Main complication: excessive space consumption (size of array = $|U|$)
  - e.g., if keys are strings of 20 symbols, we need about $2^{100}$ array entries

  *can we avoid this huge memory consumption while remaining efficient?*

  *yes! by using hashing*
Overview

- *Introduction*

- *Direct addressing*

  ⇒ *Hashing*

  - Collision resolution using chaining
  - Complexity analysis of chaining

- *Open addressing*

  - Probing strategies
  - Complexity analysis of open addressing

- *Hash functions*
Hashing

• In practice only a small fraction of keys is used, i.e., $|K| \ll |U|$
  \[\Rightarrow\] with direct addressing most of the direct address table $T$ is wasted

• The aim of hashing is:
  – map an extremely large key space onto a reasonable small range (of integers)
  – such that it is unlikely that two keys are mapped onto the same integer

• A hash function maps a key onto an index in the hash table $T$:
  \[h : U \rightarrow \{0, 1, \ldots, m-1\}\] where $m$ is the table-size and $|U| = n$

• Hash collisions, i.e., $h(k) = h(k')$ for $k \neq k'$, raise the issues:
  – how to obtain a hash function that is cheap to evaluate and minimizes collisions?
  – how to treat hash collisions when they occur?
Hashing

universe of keys

hash function

hash table

hash collision

© JPK
Hash collisions: the birthday paradox

No matter how good our hash function is, we better be prepared for collisions

- This is due to the birthday paradox:
  - the probability that your neighbor has the same birthday is \( \frac{1}{365} \approx 0.027 \)
  - if you ask 23 people, this probability raises to \( \frac{23}{365} \approx 0.063 \)
  - but, if there are 23 people in a room, two of them have the same birthday

  with probability: 
  \[
  1 - \left( \frac{365}{365} \frac{364}{365} \frac{363}{365} \ldots \frac{343}{365} \right) \approx 0.5
  \]

- Applying this to hashing yields:
  - the probability of no collisions after \( k \) insertions into an \( m \)-element table:

    \[
    \frac{m \cdot m-1 \cdot \ldots \cdot m-k+1}{m \cdot m \cdot \ldots \cdot m} = \prod_{i=0}^{k-1} \frac{m - i}{m}
    \]

    - for \( m = 365 \) and \( k \geq 50 \) this probability goes to 0
Hash collisions: the birthday paradox

Probability of no collision

Number of insertions $n$
Collision resolution by chaining

concept: put all keys that hash to the same integer in a linked list

[Luhn 1953]
Collision resolution by chaining

- Dictionary operations when using chaining:
  - `search(k)`: search for an element with key `k` in the list `T[h(k)]`
  - `insert(e)`: put element `e` at the front of list `T[h(key[e])]`
  - `delete(e)`: delete element `e` from list `T[h(key[e])]`

- Worst-case complexity of these operations:
  - assuming computing `h(k)` is rather efficient, say $\Theta(1)$
  - searching: proportional to the length of the list `T[h(k)]`
  - insertion: in constant time (note: no check whether element `e` is already present)
  - deletion: proportional to the length of the list `T[h(k)]`

- In worst case all keys are hashed onto the same slot
  - searching and deletion have same complexity as for lists! $\Theta(n)$

*The average case complexity of hashing with chaining is efficient, though*
Average case analysis of chaining (I)

- **Assumptions:**
  - we have \( n \) possible keys and \( m \) hash-table entries \( n \gg m \)
  - **uniform hashing:** each key is equally likely hashed to any integer
  - the hash value \( h(k) \) can be computed in constant time
- **The filling degree** of hash table \( T \) is \( \alpha(n, m) = \frac{n}{m} \)
  - note that the average length of list \( T[j] \) is also \( \alpha \)
- **What is the expected # elts examined in** \( T[h(k)] \) **to search key** \( k \)?
  - distinguish between **unsuccessful** and **successful** search (like in lecture #1)
- **Technical point:**
  - extend definition of \( O, \Theta \) and \( \Omega \) for functions with two parameters (like \( \alpha \))
  - e.g., \( g \in O(f) \) if \( \exists c > 0, n_0, m_0 \) such that

\[
\forall n \geq n_0, m \geq m_0 : 0 \leq g(n, m) \leq c \cdot f(n, m)
\]
Average case analysis of chaining (II)

- An unsuccessful search takes $\Theta(1+\alpha)$ time on average
  - expected time to search for key $k$ = expected time to search list $T[h(k)]$
  - this list has expected length $\alpha$
  - the computation of $h(k)$ takes a single time unit
  $\Rightarrow$ together this yields $1+\alpha$ time units on average

- A successful search also takes $\Theta(1+\alpha)$ time on average
  - let $k_i$ be the $i$-th inserted key and $A(k_i)$ be the expected time to search $k_i$:
    \[
    A(k_i) = 1 + \text{average # of keys inserted in } T[h(k_i)] \text{ after } k_i \text{ was inserted}
    \]
    - using the uniform hashing assumption this reduces to: $A(k_i) = 1 + \sum_{j=i+1}^{n} \frac{1}{m}$
    - take the average over all $n$ insertions into the hash-table $\frac{1}{n} \sum_{i=1}^{n} A(k_i)$
Average case analysis of chaining (III)

The expected number of elements examined in a successful search is

\[
\frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} \frac{1}{m} \right)
\]

\[
= \ (* \ calculus \ *)
\]

\[
\frac{1}{n} \sum_{i=1}^{n} 1 + \frac{1}{nm} \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1
\]

\[
= \ (* \ calculus \ *)
\]

\[
1 + \frac{1}{nm} \sum_{i=1}^{n} (n - i)
\]

\[
= \ (* \ calculus \ *)
\]

\[
1 + \frac{1}{nm} \left( n^2 - \frac{n(n+1)}{2} \right)
\]

\[
= \ (* \ calculus \ *)
\]

\[
1 + \frac{n - 1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \text{ and thus in } \Theta(1+\alpha)
\]
Complexity of dictionary operations using chaining

- Assume the number \( m \) of entries is (at least) proportional to \( n \)

- Then filling degree \( \alpha(n, m) = \frac{n}{m} \in O(\frac{m}{m}) = O(1) \)

- Then all dictionary operations take \( O(1) \) time on average

- This includes searching, so we can sort in \( O(n) \) on average!
Overview

- Introduction

- Direct addressing

- Hashing
  - Collision resolution using chaining
  - Complexity analysis of chaining

⇒ Open addressing
  - Probing strategies
  - Complexity analysis of open addressing

- Hash functions
Collision resolution by open addressing

- Unlike chaining all elements are stored in the hash table itself
  \[ \Rightarrow \text{ at most } n \text{ keys can be stored, i.e., } \alpha(n, m) = \frac{n}{m} \leq 1 \quad \text{[Amdahl 1954]} \]
- Since no memory is used for pointers, more data can be stored
  \[ \Rightarrow \text{ this helps to reduce the number of hash collisions} \]
- Insertion of a key \( k \):
  - probe the entries of the hash table until an empty slot is found
  - sequence of slots probed depends on key \( k \) to be inserted
  - the hash function depends on the key \( k \) and the probe number:
    \[ h : U \times \{ 0, 1, \ldots, m-1 \} \rightarrow \{ 0, 1, \ldots, m-1 \} \]
    - hash function \( h \) should eventually consider every entry in the hash table
#5: Hashing

## Insertion using open addressing

```c
void hashInsert(int T, key k) {
    int i = 0, j; // i is probe number
    repeat
        j = h(k, i); // compute (i+1)-st probe
        if T[j] == nil { // free entry found
            T[j] = k; return ; } // store key k and stop
        else i = i+1;
    until (i == T.length); // check entire table
    return hashtable overflow; // no free entry left
}
```
Searching using open addressing

```c
int hashSearch(int T, key k) {
    int i = 0, j; // i is probe number
    repeat
        j = h(k, i); // compute (i + 1)-st probe
        if T[j] == k return j; // key k found
    else i = i + 1;
    until (i == T.length || T[j] == nil);
        // check entire table or find an empty slot
    return nil; // key k has not been found
}
```
Deletion using open addressing

- Deleting key $k$ from slot $i$ by $T[i] = \text{nil}$ is inappropriate
  - if at insertion of $k$ slot $i$ was occupied we cannot retrieve $k$ anymore

- Solution: mark $T[i]$ as special value DELETED (or “obsolete”)
  - $hashInsert$ needs to be adapted to treat such slots as empty
  - $hashSearch$ remains unchanged as DELETED slots are ignored

- Search times now no longer depend on filling degree $\alpha$ only
  - If keys are to be deleted, chaining is more commonly used
How to select the next probe?

- How to generate the **probing sequence** for a given key $k$:
  \[
  \langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle
  \]
  - which is a permutation of $\langle 0, \ldots, m-1 \rangle$ for each key $k$
  $\Rightarrow$ this guarantees that all slots are eventually considered

- Ideally we have **uniform hashing**
  - i.e. each of the $m!$ permutations is equally likely as probing sequence
  - only used for analysis, in practice too expensive and approximated

- Different policies exist to select the next probe
  - we consider **linear probing**, **quadratic probing** and **double hashing**
  - quality is indicated by the number of distinct probing sequences generated
Linear probing

- Uses the hash function \( h(k, i) = (h'(k) + i) \mod m \) (for \( i < m \))
  - where \( h' \) is an auxiliary hash function

- Subsequent probed slots are offset by a linear dependence on \( i \)

- Initial probe determines the entire probe sequence
  \( \Rightarrow \) \( m \) distinct probe sequences can be generated

- Suffers from clustering, i.e., long sequences of occupied slots
  - an empty slot preceded by \( i \) full slots gets filled next with probability \( \frac{i+1}{m} \)
  \( \Rightarrow \) long sequences of occupied slots tend to get longer
Linear probing: example

\[ h'(k) = k \mod 11 \]

\[ h(k, i) = (h'(k) + i) \mod 11 \]
Quadratic probing

- Uses the hash function $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$ (for $i < m$)
  - where $h'$ is an auxiliary hash function and non-zero constants $c_1, c_2$
- Subsequent probed slots are offset by a quadratic dependence on $i$
- Initial probe determines the entire probe sequence
  
  $\Rightarrow m$ distinct probe sequences can be generated (like for linear probing)
  - . . . . . . provided the values of $m$ and constants $c_1$ and $c_2$ are appropriately chosen

- Suffers from secondary clustering
  - $h(k, 0) = h(k', 0)$ implies $h(k, i) = h(k', i)$ for all $i$
  - but avoids the clustering appearing with linear probing
Quadratic probing: example

\[ h'(k) = k \mod 11 \]

\[ h(k, i) = (h'(k) + i + 3i^2) \mod 11 \]
Double hashing

- Uses the hash function \( h(k) = (h_1(k) + i \cdot h_2(k)) \mod m \) (for \( i < m \))
  - where \( h_1 \) and \( h_2 \) are auxiliary hash functions

- Subsequent probed slots are offset by the amount \( h_2(k) \)
  - the initial probe does not determine the probe sequence
  - this yields a better distribution of keys in the hash table
  - approximates the uniform hashing strategy

- If \( h_2(k) \) and \( m \) are relatively prime, the entire hash table is searched
  - e.g., choose \( m = 2^k \) and \( h_2 \) such that it produces an odd number

- Each possible pair \( h_1(k) \) and \( h_2(k) \) yields a distinct probe sequence
  - double hashing generates \( m^2 \) distinct permutations
## Double hashing: example

$h_1(k) = k \mod 11$

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

$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod 11$
Practical efficiency of double hashing

- Hash table with 538 051 entries (final filling 99.95%)

- *Mean* number of collisions per insertion into hash table:

![Graph showing the mean number of collisions per insertion into the hash table as the usage of the hash table increases. The x-axis represents the usage of the hash table in percentage, ranging from 0 to 100, and the y-axis represents the mean number of collisions, ranging from 0 to 6.]
Efficiency of open addressing

Under the assumption of uniform hashing we have:

- An unsuccessful search takes $O\left(\frac{1}{1-\alpha}\right)$ time on average
  - if hash table is half full, 2 probes are necessary on average
  - if hash table is 90% full, 10 probes are necessary on average

- A successful search takes $O\left(\frac{1}{\alpha \cdot \ln \frac{1}{1-\alpha}}\right)$ time on average
  - if hash table is half full, about 1.39 probes are necessary on average
  - if hash table is 90% full, about 2.56 probes are necessary on average

- Recall that for chaining this was $\Theta(1+\alpha)$ for both cases
Analyzing unsuccessful search (I)

\[ \Pr\{\# \text{ probes} \geq i\} = \begin{array}{l}
\quad (*) \quad A_i \text{ is the event that there is an } i\text{-th probe and it is to an occupied slot } (*)
\quad \Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}
\quad = \begin{array}{l}
\quad (*) \quad \text{probability theory } (*)
\quad \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \ldots \Pr\{A_i \mid A_1 \cap \ldots \cap A_{i-1}\}
\quad = \begin{array}{l}
\quad (*) \quad \text{there are } n \text{ elements and } m \text{ slots } (*)
\quad \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-i+2}{m-i+2}
\quad \leq \begin{array}{l}
\quad (*) \quad \text{bound to above } (*)
\quad \left(\frac{n}{m}\right)^{i-1}
\quad = \begin{array}{l}
\quad (*) \quad \text{definition of } \alpha \ (*)
\quad \alpha^{i-1}
\end{array}
\end{array}
\end{array}
\end{array} \]
Analyzing unsuccessful search (II)

the expected number of probes

\[ \sum_{i=1}^{\infty} \Pr\{\# \text{ probes} \geq i\} \]

\[ \leq \sum_{i=1}^{\infty} \alpha^i \]

\[ = \sum_{i=0}^{\infty} \alpha^i \]

\[ = \frac{1}{1 - \alpha} \]
Analyzing successful search (I)

average number of probes in a successful search

\[
\frac{1}{n} \cdot \sum_{i=0}^{n-1} \text{average number of probes for } (i+1)-\text{st inserted key}
\]

\[\leq (* \text{ average number of probes for } (i+1)-\text{st inserted key is at most } \frac{m}{m-i} *)\]

\[
\frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i}
\]

\[= (* \text{ calculus } *)\]

\[
\frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i}
\]
Analyzing successful search (II)

\[ \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m - i} \]

\[ = \quad (* \text{calculus} *) \]

\[ \frac{1}{\alpha} \cdot \left( \sum_{k=m}^{n+1} \frac{1}{k} \right) \]

\[ \leq \quad (* \text{approximate summation by integral (cf. Example 1.7)} *) \]

\[ \frac{1}{\alpha} \cdot \int_{m-n}^{m} \frac{1}{x} \, dx \]

\[ = \quad (* \text{integral calculus} *) \]

\[ \frac{1}{\alpha} \ln \left( \frac{m}{m-n} \right) \]

\[ = \quad (* \text{definition of } \alpha *) \]

\[ \frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right) \]
#5: Hashing

Overview

- *Introduction*

- *Direct addressing*

- *Hashing*
  - Collision resolution using chaining
  - Complexity analysis of chaining

- *Open addressing*
  - Probing strategies
  - Complexity analysis of open addressing

⇒ *Hash functions*
Hash functions

- A hash function maps a key onto an integer (i.e., an index)
  - the hash function $h(k)$ should be cheap to evaluate
  - it should be surjective on the range $0 \ldots m-1$
  - it should tend to use all indexes with uniform frequency
  - it should tend to put similar keys in different parts of the hash table

- Three major techniques to obtain a “good” hash function:
  - the division method
  - the multiplication method
  - universal hashing
Division method

- Uses the hash scheme $h(k) = k \mod m$ (for $i < m$)

- Using this method, the value of $m$ should be chosen with care
  - if $m = 2^p$, then $k \mod m$ amounts to select the $p$ least significant bits of $k$

- Practical good choice: $m$ is prime and not too close to power of 2
  - example: consider 2,000 character strings
  - allow on average about 3 probes for an unsuccessful search
  - choose $m = \frac{2000}{3} \rightarrow 701$
Multiplication method

- Uses the hash scheme $h(k) = \lfloor m \cdot (k \cdot c \mod 1) \rfloor$ (for $i < m$)
  - with constant $0 < c < 1$ (Knuth suggests $c \approx (\sqrt{5} - 1)/2 \approx 0.62$)
  - note that $k \cdot c \mod 1$ is the fractional part of $k \cdot c$
  ⇒ the value of $m$ is not critical here

- Usual scheme take $m = 2^p$ and $c = \frac{s}{2^w}$ where $0 < s < 2^w$ and then:
  - first compute $k \cdot s = k \cdot c \cdot 2^w$
  - divide by $2^w$, use only the fractional part
  - multiply by $2^p$ and use only the integer part

\[ h(k) = \lfloor m \cdot (k \cdot c \mod 1) \rfloor \]
Universal hashing

- Greatest problem with hashing:
  - there is always an adversarial sequence of keys all mapped onto the same slot
- Choose randomly a hash function from a given small set $H$
  - that is independent of the keys which are going to be used
- For $k, k'$ the fraction of functions in $H$ such that $k$ and $k'$ collide is $\frac{|H|}{m}$
  - probability that $k, k'$ collide is $\frac{1}{|H|} \cdot \frac{|H|}{m} = \frac{1}{m}$

- Example: define the elements of the class of hash functions by:

$$h_{a,b}(k) = ((a \cdot k + b) \mod p) \mod m$$

  - where $p$ is a prime number such that $p > m$ and $p >$ largest key
  - integers $a$ ($1 \leq a < p$) and $b$ ($0 \leq b < p$) are chosen at execution time