Algorithms and Data Structures 2

18 – Collision resolution

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Recap**
- **Collision resolution by chaining**
  - Operations
  - Analysis
- **Collision resolution by open addressing**
  - Linear probing
  - Quadratic probing
  - Double hashing
- **Perfect hashing**

Hash tables

- **Data structure to store key/value pairs invented by H. P. Luhn in 1953**
  - Generalisation of direct-address tables

- **It consists of two main components**
  1. An array T[0,..,m - 1] of fixed size (often called hash table or bucket array)
  2. A hash function $h: U \rightarrow \{0,1,\ldots,m - 1\}$ mapping keys to slots in T, where $m \ll |U|$

- **When two keys are mapped to the same position in array T, we have a hash collision**
  - Ideally hash function is easy to compute and no collisions occur

- **Hash tables support INSERT, DELETE and SEARCH operations in O(1) time on average**

Collision resolution by chaining

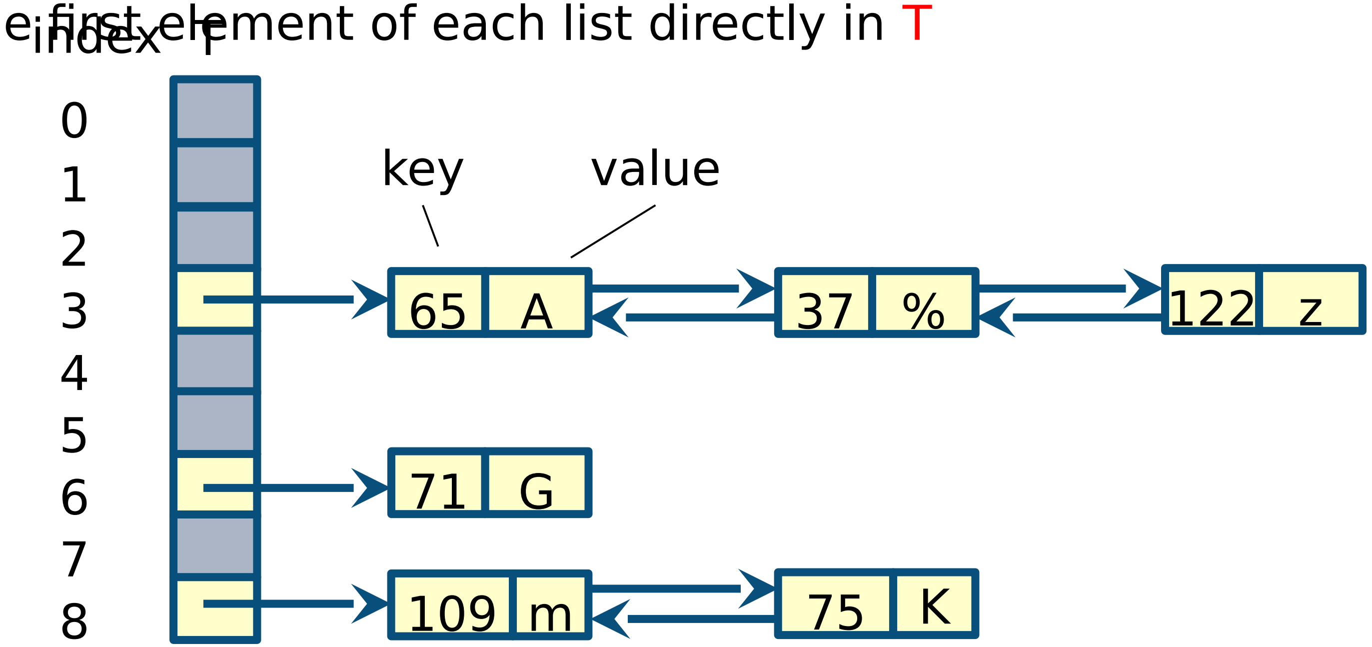
- **All elements that hash to the same slot are stored in a linked list (called chain)**
  - Doubly linked list to support fast deletion
  - Usually, no check is performed to prevent the insertion of elements with duplicate keys
  - Some implementations store the first element of each list directly in T

U = {0, …,127}
h: U → {0,…,8}

h(65) = h(37) = h(122) = 3

h(71) = 6

h(109) = h(75) = 8



Operations

- **Based on the standard operations for doubly linked lists**

```
CHAINED-HASH-INSERT(T,x)
  insert x at the head of list T[h(x.key)]
```

```
CHAINED-HASH-SEARCH(T,k)
  search for an element with key k in list T[h(k)]
```

```
CHAINED-HASH-DELETE(T,x)
  delete x from list T[h(x.key)]
```

Example

- **We want to store the ASCII table in a hash table with chaining**
  - 128 pairs (int, char)
- **Assume we can tolerate up to three elements in each chain**
- **Hashing function h implements hashing by division: $h(k) = k \bmod m$**

- **What is a good choice for m in this scenario?**

Example

- **We want to store the ASCII table in a hash table with chaining**
  - 128 pairs (int, char)
- **Assume we can tolerate up to three elements in each chain**
- **Hashing function h implements hashing by division: h(k) = k mod m**

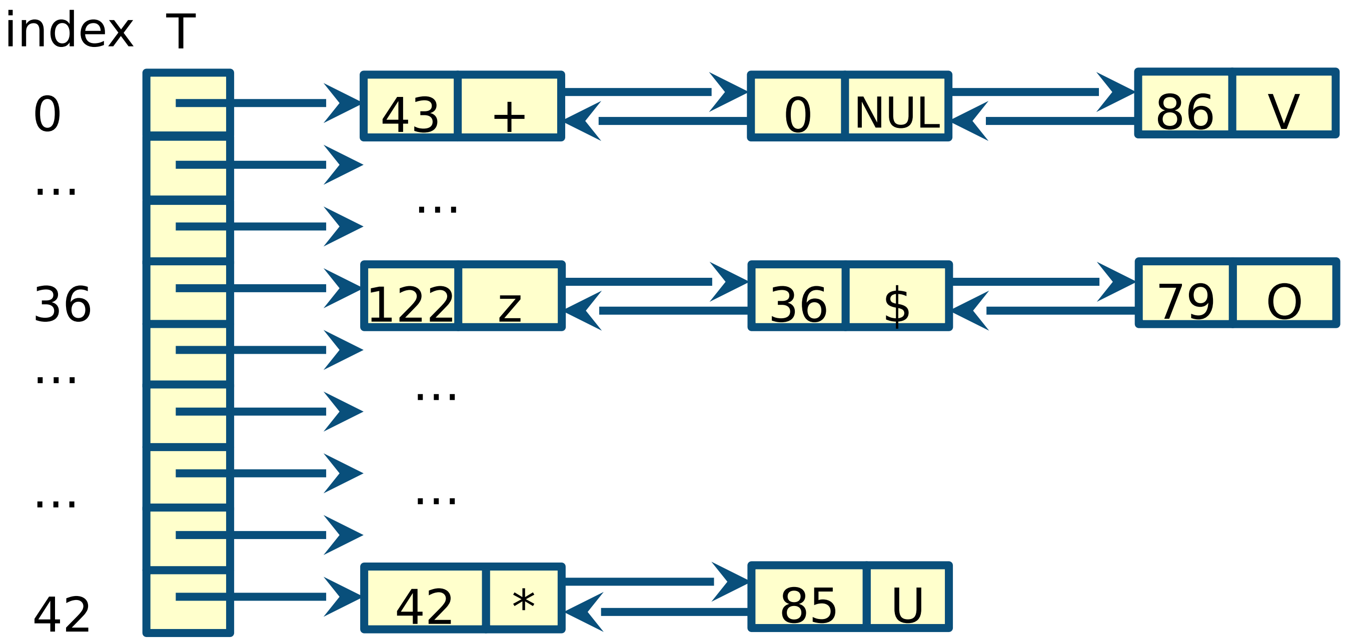
- **What is a good choice for m in this scenario?**
  - Pick a prime greater than 128/3 and not near a power of 2: 43

Example

- **h(k) = k mod 43**
  - Only 3 chains shown

All chains have length ≤ 3

The chain at T[42] is the only chain of length 2



Analysis

- **The worst case running time for insertion is O(1)**
  - Searching is required if we check for duplicates before insertions
- **Deletion is O(1) if the list in doubly linked and the input of the procedure is x**
  - Searching is required if the input is key k
- **Searching takes O(n) in the worst case**
  - All n keys hash to the same slot creating a list of length n
- **The average case running time for searching depends on how well h distributes the keys over the m slots**

Simple Uniform Hashing Assumption (SUHA)

- **We assume that hash function h evenly distributes items into the m slots**
  - Each item has an equal probability of being placed into a slot, regardless of the other elements already in the table

- **Simplifies the mathematical analysis of hash tables**

- **We also assume that it takes $O(1)$ time to compute $h(k)$**

- **Load factor for T with m slots is $\alpha = n/m$**
  - Also the average number of elements stored in a chain
  - Previous example $128/43 \approx 2.97$

Average case: unsuccessful search

- **Length of chain $T[j] = n_j$**
  - $n = n_0 + n_1 + \ldots + n_{m-1}$
  - Expected value of n_j is $E[n_j] = n/m = \alpha$

- **Under SUHA, the expected time to search unsuccessfully for a key k is the expected time to scan entirely list T[h(k)]**
  - T[h(k)] has expected length $E[n_{h(k)}] = \alpha$
  - Computing h(k) is O(1)

- **Running time is $O(1 + \alpha)$**

Average case: successful search

- **Probability that a chain is searched is proportional to the number of elements it contains**
  - Each chain is not equally likely to be searched

- **Assume the element we are searching x is equally likely to be in any of the n elements stored in the table**
  - We need to examine all the elements before x in the chain, plus one

- **The expected number of elements examined in a successful search is the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list**

Average case: successful search (cont.)

- x_i denotes the **i-th** element inserted into the table with $i = 1,2,..,n$ and let $k_i = x_i.key$
- **Define indicator random variable** $X_{ij} = I\{h(k_i)=h(k$

A random variable that has the value 1 or 0, according to whether a specified event occurs or not

- **Under SUHA, we have** $P\{h(k_i)=h(k_j)\} = 1/m$
  - This implies $E[X_{ij}] = 1/m$

n

(n - 1) + (n - 2) + … + (n - n)

=

Average case: successful search (cont.)

- x_i denotes the i-th element inserted into the table with $i = 1,2,..,n$ and let $k_i = x_i.key$
- Define indicator random variable $X_{ij} = I\{h(k_i)=h(k$
- Under SUHA, we have $P\{h(k_i)=h(k_j)\} = 1/m$
  - This implies $E[X_{ij}] = 1/m$

A random variable that has the value 1 or 0, according to whether a specified event occurs or not

=

=

Average case: successful search (cont.)

- x_i denotes the **i-th** element inserted into the table with $i = 1,2,..,n$ and let $k_i = x_i.key$
- Define indicator random variable $X_{ij} = I\{h(k_i)=h(k$
- Under SUHA, we have $P\{h(k_i)=h(k_j)\} = 1/m$
  - This implies $E[X_{ij}] = 1/m$

A random variable that has the value 1 or 0, according to whether a specified event occurs or not

=

=

=

$n/m = \alpha$ $m = n/\alpha$

Average case: successful search (cont.)

- x_i denotes the i-th element inserted into the table with $i = 1,2,..,n$ and let $k_i = x_i.key$
- Define indicator random variable $X_{ij} = I\{h(k_i)=h(k$
- Under SUHA, we have $P\{h(k_i)=h(k_j)\} = 1/m$
  - This implies $E[X_{ij}] = 1/m$

A random variable that has the value 1 or 0, according to whether a specified event occurs or not

=

=

=

- Running time is $O(1 + \alpha)$

Average case

- **If the number of slots is proportional to the number of elements in the table, we have $n = O(m)$**

- **Then $\alpha = n/m = O(m)/m = 1$**

- **Searching takes constant time on average in hash tables with chaining**

Collision resolution by open addressing

- **If a collision occurs, alternative cells are tried (or probed) until an empty cell is found**
  - Does not use linked lists to resolve hash collisions
  - Pointers are avoided (more space available for slots)

- **To determine which slots to probe, the hash function is extended to include the probe number (starting from 0) as a second input: $h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$**

- **With open addressing, we require that every hash-table position is eventually considered as a slot for a new key as the table fills up**

- **Method preferred when memory limited (small handheld device or sensor network)**

Insertion

- **The HASH-INSERT procedure takes as input a hash table T and a key k**
  - It either returns the slot number where it stores k or raise an error because the T is already full

```
HASH-INSERT(T,k)
  i := 0
  while i < m
    j := h(k,i)
    if T[j] = NIL
       T[j] := k
       return j
    else i := i + 1
  error “hash table overflow”
```

Example

- **We will see later three methods to compute probing sequences**
  - For now, assume slot $i + 1$ is probed after slot i (wrapping around when $i = m$)

```
HASH-INSERT(T,k)
  i := 0
  while i < m
    j := h(k,i)
    if T[j] = NIL
       T[j] := k
       return j
    else i := i + 1
  error “hash table overflow”
```

index	T
0	
1	113
2	
3	83
4	
5	109
6	
7	71

- INSERT(T, 65)
- h(65) = 65 mod 8 = 1
- Collision

Example

- **We will see later three methods to compute probing sequences**
  - For now, assume slot $i + 1$ is probed after slot i (wrapping around when $i = m$)

```
HASH-INSERT(T,k)
  i := 0
  while i < m
    j := h(k,i)
    if T[j] = NIL
       T[j] := k
       return j
    else i := i + 1
  error "hash table overflow"
```

index	T
0	
1	113
2	65
3	83
4	
5	109
6	
7	71

- INSERT(T, 65)
- h(65) = 65 mod 8 = 1
- Insert element in slot 2

Example

- **We will see later three methods to compute probing sequences**
  - For now, assume slot i + 1 is probed after slot i (wrapping around when i = m)

```
HASH-INSERT(T,k)
  i := 0
  while i < m
    j := h(k,i)
    if T[j] = NIL
       T[j] := k
       return j
    else i := i + 1
  error “hash table overflow”
```

index	T
0	
1	113
2	65
3	83
4	
5	109
6	
7	71

- INSERT(T, 57)
- h(57) = 57 mod 8 = 1
- Collision

Example

- **We will see later three methods to compute probing sequences**
  - For now, assume slot i + 1 is probed after slot i (wrapping around when i = m)

```
HASH-INSERT(T,k)
  i := 0
  while i < m
    j := h(k,i)
    if T[j] = NIL
       T[j] := k
       return j
    else i := i + 1
  error “hash table overflow”
```

index	T
0	
1	113
2	65
3	83
4	57
5	109
6	
7	71

- INSERT(T, 57)
- h(57) = 57 mod 8 = 1
- Insert element in slot 4

Search

- **The HASH-SEARCH procedure takes as input a hash table T and a key k**
  - It either returns the slot number where k is stored, NIL if k is not stored in T

```
HASH-SEARCH(T,k)
  i := 0
  j := h(k,i)
  while T[j] = NIL or i < m
    if T[j] = k
      return j
    i := i + 1
    j := h(k,i)
  return NIL
```

Deletion

- **Simply deleting a key from slot i would break the retrieval of any key k during whose insertion we had probed slot i and found it occupied**

- **This problem is solved by storing in the slot the special value DELETED**
  - Modify HASH-INSERT to treat DELETED slots as if they were empty
  - HASH-SEARCH is not modified, since it will pass over DELETED values while searching

- **When using DELETED, search times no longer depend on the load factor**
  - Use chaining when keys must be deleted

Linear probing

- **Use a hash function of the form $h(k,i) = (h'(k) + i) \bmod m$ where h' is an ordinary hash function**
  - Probe sequence in previous example

- **Since the initial probe determines the entire probe sequence, there are only m distinct probe sequences**

- **Linear probing is easy to implement, but it suffers from primary clustering**
  - Long runs of occupied slots build up, increasing the average search time

Quadratic probing

- **Use a hash function of the form $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ where h' is an ordinary hash function and c_1, c_2 constants**

- **Much better than linear probing**
  - To make full use of the hash table, the values of c_1, c_2, and m are constrained

- **If two keys have the same initial probe position, then their probe sequences are the same**
  - This leads to a milder form of clustering, called secondary clustering

Double hashing

- **Use a hash function of the form $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$ where h_1 and h_2 are ordinary hash functions**

- **The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched**

1. Let m be a power of 2 and design h_2 to always produce an odd number
2. Let m be prime and design h_2 to always return a positive integer less than m

- **One of the best methods available for open addressing**

Average case analysis

- **We analyse the expected number of probes under SUHA**

- **Unsuccessful search and insertion require at most $1/(1-\alpha)$ probes**
  - Recall $1/(1 - \alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$
  - First probe + probability α the first slot is occupied + probability α^2 the first two slots are occupied + …

- **If α is constant, unsuccessful search and insertion run in $O(1)$ time**
  - If table is half full ($\alpha = .5$) the average number of probes is $1/(1 - .5) = 2$
  - If table is 90% full ($\alpha = .9$) the average number of probes is $1/(1 - .9) = 10$

Average case analysis (cont.)

- **Successful search require at most $1/\alpha \ln(1/(1 - \alpha))$ probes**

- **If α is constant, successful search run in $O(1)$ time**
  - If table is half full ($\alpha = .5$) the average number of probes is less than 1.387
  - If table is 90% full ($\alpha = .9$) the average number of probes is less than 2.559

Perfect hashing

- **Applicable when the set of keys is static**
  - Once the keys are stored in the table, they never change

- **Perfect hashing is a hashing technique that does not generate collisions**
  - Search is performed in constant time in all cases

- **We use two levels of hashing, with universal hashing at each level**
  1. Same as for hashing with chaining: we hash the n keys into m slots using a universal hash function
  2. Small secondary hash tables with an associated hash function h_j in place of the linked lists

- **By choosing each h_j carefully, we can guarantee that there are no collisions at the secondary level**

Summary

- **Collision resolution by chaining**
  - Operations
  - Analysis
- **Collision resolution by open addressing**
  - Linear probing
  - Quadratic probing
  - Double hashing
- **Perfect hashing**