

# Fundamental Algorithms

## Chapter 5: Hash Tables

Harald Räcke

Winter 2015/16

# Generalised Search Problem

## Definition (Search Problem)

**Input:** a sequence or set  $A$  of  $n$  elements  $\in \mathcal{A}$ , and an  $x \in \mathcal{A}$ .

**Output:** Index  $i \in \{1, \dots, n\}$  with  $x = A[i]$ , or NIL, if  $x \notin A$ .

- complexity depends on data structure
- complexity of operations to set up data structure? (insert/delete)

## Definition (Generalised Search Problem)

- Store a set of objects consisting of a key and additional data:

```
Object := (  
    key: Integer, .  
    record: Data );
```

- search/insert/delete objects in this set

# Direct-Address Tables

## Definition (table as data structure)

- similar to array: access element via index
- usually contains elements only for some of the indices

## Direct-Address Table:

- assume: limited number of values for the keys:  
 $U = \{0, 1, \dots, m - 1\}$
- allocate table of size  $m$
- use keys directly as index

## Direct-Address Tables (2)

```
DirAddrInsert(T:Table , x:Object) {  
    T[x.key] := x;  
}
```

```
DirAddrDelete(T:Table , x:Object){  
    T[x.key] := NIL;  
}
```

```
DirAddrSearch(T:Table , key:Integer){  
    return T[key];  
}
```

## Direct-Address Tables (3)

### Advantage:

- very fast: search/delete/insert is  $\Theta(1)$

### Disadvantages:

- $m$  has to be small, or otherwise, the table has to be very large!
- if only few elements are stored, lots of table elements are unused (waste of memory)
- all keys need to be distinct (they should be, anyway)

# Hash Tables

## Idea: compute index from key

Wanted: function  $h$  that

- maps a given key to an index,
- has a relatively small range of values, and
- can be computed efficiently,

## Definition (hash function, hash table)

Such a function  $h$  is called a **hash function**.

The respective table is called a **hash table**.

# Hash Tables – Insert, Delete, Search

```
HashInsert(T:Table , x:Object) {  
    T[h(x.key)] := x;  
}
```

```
HashDelete(T:Table , x:Object) {  
    T[h(x.key)]:= NIL;  
}
```

```
HashSearch(T:Table , x:Object) {  
    return T[h(x.key)];  
}
```

# So Far: Naive Hashing

## Advantages:

- still very fast: search/delete/insert is  $\Theta(1)$ , if  $h$  is  $\Theta(1)$
- size of the table can be chosen freely, provided there is an appropriate hash function  $h$

## Disadvantages:

- values of  $h$  have to be distinct for all keys
- however: impossible to find a hash function that produces distinct values for any set of stored data

## ToDo: deal with **collisions**:

objects with different keys that share a common hash value have to be stored in the same table element



# Resolve Collisions by Chaining

## Idea:

- use a table of **containers**
- containers can hold an arbitrarily large amount of data
- using (linked) lists as containers: **chaining**

```
ChainHashInsert(T:Table , x:Object) {  
    insert x into T[h(x.key)];  
}
```

```
ChainHashDelete(T:Table , x:Object) {  
    delete x from T[h(x.key)];  
}
```

# Resolve Collisions by Chaining

```
ChainHashSearch(T:Table , x:Object) {  
    return ListSearch(x, T[h(x.key)] );  
    ! result: reference to x or NIL, if x not found;  
}
```

## Advantages:

- hash function no longer has to return distinct values
- still very fast, if the lists are short

## Disadvantages:

- delete/search is  $\Theta(k)$ , if  $k$  elements are in the accessed list
- worst case: all elements stored in one single list (very unlikely).

# Chaining – Average Search Complexity

## Assumptions:

- hash table has  $m$  slots (table of  $m$  lists)
- contains  $n$  elements  $\Rightarrow$  **load factor**:  $\alpha = \frac{n}{m}$
- $h(k)$  can be computed in  $O(1)$  for all  $k$
- all values of  $h$  are equally likely to occur

## Search complexity:

- on average, the list corresponding to the requested key will have  $\alpha$  elements
- unsuccessful search: compare the requested key with all objects in the list, i.e.  $O(\alpha)$  operations
- successful search: requested key last in the list;  
 $\Rightarrow$  also  $O(\alpha)$  operations

Expected: Average complexity:  $O(1 + \alpha)$  operations

# Hash Functions

A good hash function should:

- satisfy the assumption of even distribution:  
each key is equally likely to be hashed to any of the slots:

$$\sum_{k: h(k)=j} (P(\text{key} = k)) = \frac{1}{m} \quad \text{for all } j = 0, \dots, m-1$$

- be easy to compute
- be “non-smooth”: keys that are close together should not produce hash values that are close together (to avoid clustering)

**Simplest choice:**  $h = k \bmod m$  ( $m$  a prime number)

- easy to compute; even distribution if keys evenly distributed
- however: **not** “non-smooth”

# The Multiplication Method for Integer Keys

Two-step method

1. multiply  $k$  by constant  $0 < \gamma < 1$ , and extract fractional part of  $k\gamma$
2. multiply by  $m$ , and use integer part as hash value:

$$h(k) := \lfloor m(\gamma k \bmod 1) \rfloor = \lfloor m(\gamma k - \lfloor \gamma k \rfloor) \rfloor$$

## Remarks:

- value of  $m$  uncritical; e.g.  $m = 2^p$
- value of  $\gamma$  needs to be chosen well
- in practice: use fix-point arithmetics
- non-integer keys: use encoding to integers (ASCII, byte encoding, ...)

# Open Addressing

## Definition

- no containers: table contains objects
- each slot of the hash table either contains an object or NIL
- to resolve collisions, more than one position is allowed for a specific key

**Hash function:** generates **sequence** of hash table indices:

$$h: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$$

## General approach:

- store object in the first empty slot specified by the probe sequence
- empty slot in the hash table guaranteed, if the probe sequence  $h(k, 0), h(k, 1), \dots, h(k, m-1)$  is a permutation of  $0, 1, \dots, m-1$

## Open Addressing – Algorithms

```
OpenHashInsert(T:Table, x:Object) : Integer {  
  for i from 0 to m-1 do {  
    j := h(x.key, i);  
    if T[j]=NIL then { T[j] := x; return j; }  
  }  
  cast error "hash_table_overflow"  
}
```

```
OpenHashSearch(T:Table, k:Integer) : Object {  
  i := 0;  
  while T[h(k,i)]  $\diamond$  NIL and i < m {  
    if k = T[h(k,i)].key then return T[h(k,i)];  
    i := i+1;  
  }  
  return NIL;  
}
```

# Open Addressing – Linear Probing

**Hash function:**  $h(k, i) := (h_0(k) + i) \bmod m$

- first slot to be checked is  $T[h_0(k)]$
- second probe slot is  $T[h_0(k) + 1]$ , then  $T[h_0(k) + 2]$ , etc.
- wrap around to  $T[0]$  after  $T[m - 1]$  has been checked

**Main problem: clustering**

- continuous sequences of occupied slots (“clusters”) cause lots of checks during searching and inserting
- clusters tend to grow, because all objects that are hashed to a slot inside the cluster will increase it
- slight (but minor) improvement:  $h(k, i) := (h_0(k) + ci) \bmod m$

**Main advantage: simple and fast**

- easy to implement
- cache efficient!



# Open Addressing – Quadratic Probing

**Hash function:**  $h(k, i) := (h_0(k) + c_1 i + c_2 i^2) \bmod m$

- how to choose constants  $c_1$  and  $c_2$ ?
- objects with identical  $h_0(k)$  still have the same sequence of hash values  
("secondary clustering")

**Idea: double hashing**  $h(k, i) := (h_0(k) + i \cdot h_1(k)) \bmod m$

- if  $h_0$  is identical for two keys,  $h_1$  will generate different probe sequences

# Open Addressing – Double Hashing

$$h(k, i) := (h_0(k) + i \cdot h_1(k)) \pmod{m}$$

## How to choose $h_0$ and $h_1$ :

- range of  $h_0$ :  $U \rightarrow \{0, \dots, m - 1\}$  (cover entire table)
- $h_1(k)$  must never be 0 (no probe sequence generated)
- $h_1(k)$  should be prime to  $m$  for all  $k$   
→ probe sequence will try all slots
- if  $d$  is the greatest common divisor of  $h_1(k)$  and  $m$ , only  $\frac{1}{d}$  of the hash slots will be probed

## Possible choices:

- $m = 2^M$  and let  $h_1$  generate odd numbers, only
- $m$  a prime number, and  $h_1: U \rightarrow \{1, \dots, m_1\}$  with  $m_1 < m$

# Collisions and Clustering

## Scenarios for Collisions:

- keys share the same primary hash value:  $h(k_1, 0) = h(k_2, 0)$   
→ same sequence of hash values for linear and quadratic probing
- keys share a value of the hash sequence:  $h(k_1, i) = h(k_2, j)$   
→ same sequence of hash values for linear probing  
→ different hash values for next try:  $h(k_1, i + 1) \neq h(k_2, j + 1)$

## Example:

- multiple keys that share the same has values
- linear hashing will cause primary cluster
- cluster will also grow by all keys mapped to a hash value within this cluster

# Open Addressing – Deletion

## Problem remaining: how to delete?

- search entry, remove it
  - does not work:
    - insert 3, 7, 8 having same hash-value, then delete 7
    - how to find 8?
- ⇒ do not delete, just mark as deleted

## Next problem:

- searching stops if first empty entry found
- after many deletions: lots of unnecessary comparisons!

# Open Addressing – Deletion (2)

## Deletion general problem for open hashing

- only “solution”: new construction of table after some deletions
- hash tables therefore commonly don't support deletion

## Inserting

- inserting efficient, but too many inserts  $\Rightarrow$  not enough space
- $\Rightarrow$  if ratio  $\alpha$  too big, new construction of table with larger size

## Still...

- searching faster than  $O(\log n)$  possible