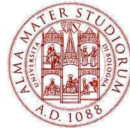
Algorithms and Data Structures 2010-2011

Lesson 4: *Sets, Dictionaries and Hash Tables*

Luciano Bononi
<bononi@cs.unibo.it>
http://www.cs.unibo.it/~bononi/
(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)



International Bologna Master in Bioinformatics

University of Bologna

29/04/2011, Bologna

Outline of the lesson

- **Sets**
  - Definition
  - Abstract Data Type (ADT)
  - Implementions
- **Dictionaries**
  - Definition
  - Hash tables

Sets: **definition**

- **DEFINITION:** a **Set** is a collection (or family) of elements that are of the same type
- The elements in a set are also called **components** or **members**
- Sets are one of the most fundamental concepts in mathematics
- The elements of a set are **homogeneous**
- There is **no notion of order** among elements
- Multiple instances of the same element are **not allowed**

Set: **Abstract Data Type**

- **DEFINITION**:
  - a set **S** is an unordered collection of elements from a **universe**
  - an element cannot appear more than once in **S**
- The **cardinality** of S is the number of elements in S
  - an **empty set** is a set whose cardinality is **zero**
  - a **singleton** is a set whose cardinality is **one**

Set ADT: some **operations**

- **S, T = sets** **e = element**
- **Creates** a new Set — Create(S)
- Given **(e)** returns **{e}** — Singleton(e)
- Returns "S **union** T" — Union(S, T)
- Returns "S **intersection** T" — Intersection(S, T)
- Returns the "**complement** of S" — Complement(S)
- Returns True if e **is in** S, otherwise returns False — ElementOf(e, S)
- Returns the **cardinality** of S — Cardinality(S)

...

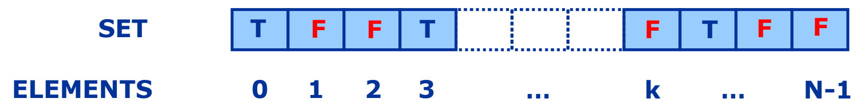
Set ADT: **implementation**

- As usual we have many possible implementations that are suitable for the same ADT: each one with PROs and CONs
  - **CHOICE 1**: Boolean vector
  - **CHOICE 2**: Unordered list
  - **CHOICE 3**: Ordered List

Set ADT: implementation with **Boolean vectors**

- **CHOICE 1: Boolean vector**

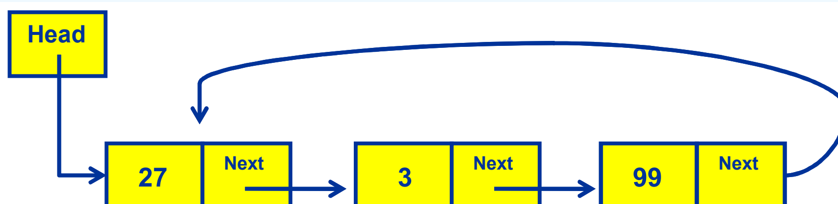
  each Boolean element in the vector (true / false) represents an element of the set



- **Advantages and disadvantages:**
  - Memory footprint?
  - Implementation of ADT operations
  - Cost of the operations?

Set ADT: implementation with **unsorted lists**

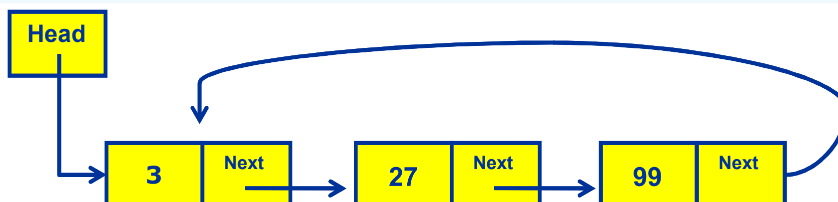
- **CHOICE 2: Unsorted List**
  - Singly or doubly linked?



- **Advantages and disadvantages:**
  - Memory footprint?
  - Implementation of ADT operations
  - Cost of the operations?

Set ADT: implementation with **sorted lists**

- **CHOICE 3: Sorted List**



- **Advantages and disadvantages:**
  - Memory footprint?
  - Implementation of ADT operations
  - Cost of the operations?

Dictionary: **definition**

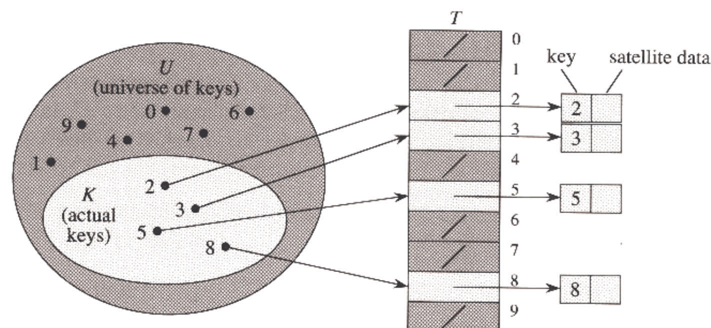
- **DEFINITION:** a **Dictionary** is a **special kind of set**
- Only few operations are allowed:
  - **Find** an element
  - **Insert** a new element
  - **Delete** an existing element
- The elements of a dictionary are usually called **keys**
- **Many different implementations are suitable, in the following we'll see the hash tables**

Dictionary: **goal**

- **GOAL**: to define a data structure that permits an efficient implementation of the find(), insert() and delete() operations
- The universe cardinality is often huge
- It often not known "a priori" the number of elements to accommodate in the dictionary
- Dictionaries are very often used in software, for example: compilers and databases

Hash tables: **unrealistic** solution

- The hash table **T** is an **array** of "pointers"
- Each position (**slot**) in the hash table (**T**) corresponds to a key in the **universe of keys**
- **T[k]** corresponds to an element with key **k**
- If the set contains no element with key **k**, then **T[k]=NULL**



Hash tables: **unrealistic** solution

- **Insert**(), **Delete**() and **Find**() all take **O(1)** (worst-case) time
- **PROBLEM:**
  - The scheme wastes **too much space** if the universe is large compared with the actual number of elements to be stored
  - In many real world cases this solution **is impossible to implement** (i.e. each time the cardinality of the universe is quite large)

Hash tables: the concept of **hashing**

$\{K_0, K_1, \ldots, K_{N-1}\}$ possible keys

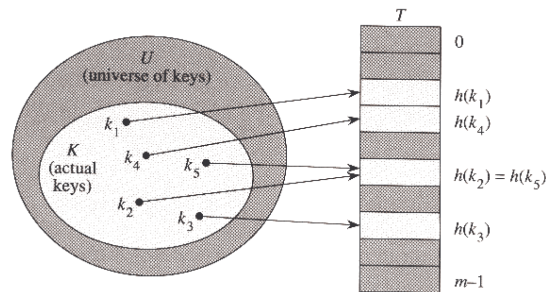
hash function h

$T[0, \ldots, m-1]$
hash table

- Usually, **m << N**
- **h(K_i)** = an integer in [0, …, m-1] called the **hash value** of K_i

Hash tables: **hash function definition**

- With hashing, an element having key **k** is stored in **T[h(k)]**
- **h: hash function definition**
  - maps the universe U of keys into the slots of a hash table **T[0,1,…,m-1]**
  - an element of key **k** hashes to slot **h(k)**
  - **h(k)** is the hash value of key **k**



Hash tables: **collisions**

- **PROBLEM**: **collision**
  - two keys may hash **to the same slot**
  - can we ensure that any two distinct keys get different cells?
    - **No**, if |U|>m, where m is the size of the hash table

1. **Design a good hash function**
   - that is **fast to compute** and
   - **minimize** the number of **collisions**
2. Design a method to **resolve the collisions** when they occur

Hash tables: **hash function examples**

- **The division method**
  - **h(k) = k mod m** e.g. m=12, k=100, h(k)=4
  - Requires only a single division operation (quite fast)
  - *Certain values of m should be avoided*
    - e.g. if m=2p, then h(k) is just the p lowest-order bits of k; the hash function does not depend on all the bits
  - Similarly, if the keys are decimal numbers, should not set m to be a power of 10
  - It's a good practice to set the table size m to be a prime number
  - Good values for m: primes not too close to exact powers of 2
    - e.g. the hash table is to hold 2000 numbers, and we don't mind an average of 3 numbers being hashed to the same entry
      - choose m=701

Hash tables: **hash function examples**

- Can the keys be strings?
  - Most hash functions assume that the keys are natural numbers
  - If keys are not natural numbers, a way must be found to interpret (translate) them as natural numbers
- **EXAMPLE**
  - Add up the ASCII value of the characters in the string
  - **Problems**:
    - different permutations of the same set of characters would have the same hash value
    - the keys are well distributed in the table?

Hash tables: **open addressing**

- Open addressing hash tables store the records directly **within the array** (also called **closed hashing**)
- **Open addressing**:
  - relocate the key K to be inserted if it collides with an existing key. That is, we store K at an entry different from T[h(K)]
- **Two issues arise**:
  - what is the relocation scheme?
  - how to search for K later?
- Three common methods for resolving a collision in open addressing:

1) **linear probing**, 2) **quadratic probing**, 3) **double hashing**

Hash tables: **open addressing**

- To insert a key **K**, compute $\mathbf{h_0(K)}$
  - if $\mathbf{T[h_0(K)]}$ is empty, insert it there
- If collision occurs, probe *alternative cell* $\mathbf{h_1(K)}$, $\mathbf{h_2(K)}$, ... **until an empty cell is found**
- $\mathbf{h_i(K) = (hash(K) + f(i)) \bmod m}$, with $f(0) = 0$
- The function **f()**: is called **collision resolution strategy**

Hash tables: **open addressing, linear probing**

- **f(i) =i**
  - cells are probed **sequentially** (with wraparound)
  - **$h_i(K)$ = (hash(K) + i) mod m**
- **Insertion**:
  - Let K be the new key to be inserted. We compute hash(K)
  - for i = 0 to m-1
    - compute L = (hash(K) + i) mod m
    - if T[L] is empty, then we put K there and stop
- **If we cannot find an empty entry to put K, it means that the table is full and we should report an error**

Hash tables: **exercise**

- Given an hash table that is initially empty and with:
  - size = 17
  - H(K) = k mod 17 and with linear probing
- Show the content of the hash table:
  - after inserting the keys:
    - B, I, O, I, N, F, O, R, M, A, T, I, C, S
  - after deleting the keys:
    - I, S
  - and finally after inserting the keys:
    - C, O, O, L

Hash tables: **open addressing, clustering**

- **Quadratic probing**
  - **$f(i) = i^2$**
    - **$hi(K) = (hash(K) + i^2) \bmod m$**

- A *block of contiguously occupied table entries* is a **cluster**
- On the average, when we insert a new key K, we may hit the middle of a cluster. Therefore, the time to insert K would be proportional to half the size of a cluster. That is, **the larger the cluster, the slower the performance**
- Linear probing and quadratic probing are both affected by problems of clustering (primary and secondary clustering)

Hash tables: **open addressing, double hashing**

- To alleviate the problem of clustering, the sequence of probes for a key should be independent of its primary position => use **two hash functions**: *hash()* and *hash2()*

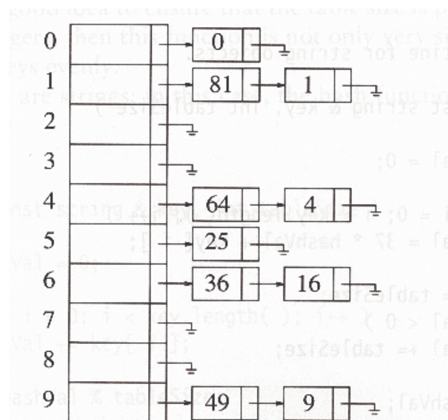
- **$f(i) = i * hash2(K)$**
  - E.g. $hash2(K) = R - (K \bmod R)$, with R is a prime number smaller than m

Hash tables: **deletion** in **open addressing**

- How is it possible to implement **deletion** in open addressing?
- In open addressing the deletion of a key can not be implemented simply erasing the key from the table, **otherwise this will isolate records further down the probe sequence**
- **SOLUTION:** add an extra bit to each table entry, and mark a deleted slot by storing a special value DELETED
- Using this approach, how is it possible to implement the **Find**(), **Insert**() and **Delete**() operations?

Hash tables: **separate chaining**

Instead of a hash table, we use a **table of linked lists:** keep a **linked list of keys** that hash to the same value



Hash tables: **separate chaining**

- **To insert a key K**
  - compute **h(K)** to determine which list to traverse
  - if **T[h(K)]** contains a null pointer, initialize this entry to point to a linked list that contains K alone
  - if **T[h(K)]** is a non-empty list, we add K at the beginning of this list
- **To delete a key K**
  - compute **h(K)**, then search for K within the list at **T[h(K)]**
  - delete **K** if it is found

Hash tables: **separate chaining**

- Assume that we will be storing n keys. Then we should make **m** the next larger prime number. If the hash function works well, the number of keys in each linked list will be a **small constant**
- Therefore, we expect that each **search**, **insertion**, and **deletion** can be done in **constant time**. The value of this constant depends on the average length of lists
- **Disadvantage**: memory allocation in linked list manipulation will slow down the program
- **Advantage**: deletion is easy

Hash tables: **separate chaining, exercise**

- **EXERCISE**: in a dictionary some keys are accessed much more frequently than others (i.e. in Italian some words are much more frequently used than others)
  - The dictionary is implemented using and hash table with separate chaining
  - Is it possible to modify the hash table to reduce the cost of the **Find**() operation in the "average case"?
- **SUGGESTION:** implement a very simple heuristic to modify the list management and to speed up accesses to the data structure

References

- Part of this material is inspired / taken by the following freely available resources:
  - http://www.cs.ust.hk/~huamin/COMP171/index.htm

Algorithms and Data Structures 2010 - 2011

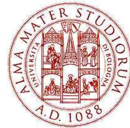
Lesson 4: *Sets, Dictionaries and Hash Tables*

Luciano Bononi

<bononi@cs.unibo.it>

http://www.cs.unibo.it/~bononi/

(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)



International Bologna Master in Bioinformatics

University of Bologna

29/04/2011, Bologna