Appendice C
Hashing

Introduzione

In questa appendice viene fornita un'illustrazione del concetto dell'*hashing*. Si tratta di un'importante nozione di programmazione che, sebbene molto utilizzata ai giorni d'oggi, risale agli albori dell'informatica. La teoria deriva dagli studi eseguiti da Hans Peter Luhn presso i laboratori di ricerca della IBM e pubblicati nel 1953. L'obiettivo consisteva nell'individuare una funzione in grado di manipolare chiavi di ricerca al fine di produrre un insieme uniformemente distribuito di variabili randomiche. Una trattazione approfondita dell'argomento può essere trovata nel libro [ARTCP3].

La teoria dell'hashing è correntemente utilizzata in diversi ambiti dell'informatica e non solo: dalle strutture dati, agli algoritmi di crittografia, alla verifica della correttezza delle trasmissioni e così via. In questa appendice, però, l'attenzione è focalizzata esclusivamente sull'utilizzo dell'*hashing* nel contesto delle strutture dati (*hash table*), che permette di organizzare i dati in modo tale che la relativa ricerca sia estremamente efficiente (accesso diretto nei casi migliori).

Nel linguaggio di programmazione Java, il concetto dell'*hashing* è costantemente utilizzato, sia indirettamente (per esempio attraverso il ricorso a collezioni di dati come `Hashtable` e `HashMap`, sia direttamente, *overriding* del metodo `hashCode` ereditato dalla classe `java.lang.Object`. Specialmente questo secondo utilizzo, sebbene presente sin dagli albori del linguaggio Java, continua a generare una certa mescolanza di concetti e dubbi anche tra programmatori non più junior.

Obiettivo di questo capitolo, pertanto, è presentare una panoramica generale del concetto di *hashing*, corredata da esempi pratici, al fine di fugare dubbi e perplessità che circondano il concetto.

Il capitolo è concluso con la presentazione di un algoritmo generale utile per l'implementazione del metodo `hashCode`.

Un po' di teoria

L'*hash table* è una particolare struttura dati utilizzati per memorizzare insieme di dati in modo tale che le operazioni di ricerca dei singoli elementi presentino una complessità asintotica teorica del tipo $O(1)$. Ciò equivale ad affermare che le operazioni di ricerche possano avvenire direttamente, ossia senza eseguire particolari cicli. Come si vedrà di seguito, ciò è possibile solo in condizioni molto particolari, mentre, nel peggiore dei casi (situazione assolutamente poco probabile) si ottiene una complessità asintotica pari a $O(n)$ che, nelle implementazioni più ricorrenti, equivale allo scenario in cui tutti gli elementi generano il medesimo indirizzo relativo! Questa, fortunatamente, è una situazione assolutamente inconsueta, mentre in casi normali le performance di questi algoritmi sono molto elevate. In effetti sia il caso migliore sia quello medio danno luogo a una complessità $O(1)$, anche se spesso a discapito di un'elevata occupazione di memoria.

Come termini di paragone, si ricordi che ricerche in vettori non ordinati richiedono una complessità asintotica pari a $O(n)$; ciò significa che, nel caso peggiore (elemento non presente nel vettore), è necessario scorrere l'intero vettore prima di poter addivenire a tale conclusione. Mentre ricerche di dicotomie in array ordinati e ricerche in alberi binari (bilanciati) consentono di ottenere complessità asintotiche del tipo $O(\log n)$. Analogamente al caso delle *hash table*, anche gli alberi binari possono presentare, in situazioni desuete in cui gli alberi degenerino in liste, complessità asintotiche pari a $O(n)$.

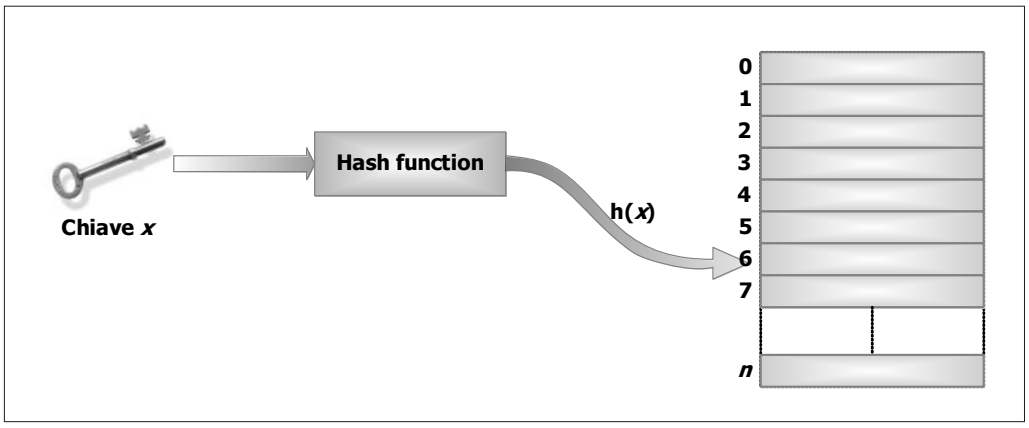

Figura C.1 – Rappresentazione grafica del concetto di hash table.

L'idea alla base delle tabelle *hash* consiste nel sostituire il concetto di comparazione delle tradizionali ricerche con quello di determinazione della posizione dell'elemento ricercato. In particolare, si prendono in considerazione funzioni in grado di trasformare la chiave degli elementi da memorizzare nel relativo indirizzo. Queste funzioni sono per l'appunto chiamante funzioni di hash (Figura C.1).

In termini più formali, le funzioni *hash*, `h`, sono funzioni tali che

$h(\div) \rightarrow N \quad$ per ogni $\quad \div \in U$

dove ÷ è un elemento appartenente all'insieme dei possibili elementi (`U`) da dover memorizzare nella struttura dati e `N` è l'insieme dei numeri naturali. `h(÷)`, pertanto, è una funzione che deve essere in grado di associare un numero naturale (l'indirizzo) a ciascun elemento ÷.

Più precisamente, indicando con la lettera `U` l'universo degli elementi da memorizzare, e con la lettera `u` la relativa dimensione (si assume che tale insieme sia finito, questa è un'assunzione necessaria solo in teoria, mentre nella pratica può essere semplicemente trascurata assumendo `u` molto grande), e `T[1..m]` (con `m` uguale alla dimensione del vettore), ne segue:

$h: U \rightarrow \{0,1,2, \ldots, m-1\}$

Pertanto la funzione `h` associa a ogni possibile $\div \in U$ una ben definita posizione all'interno del vettore `T`.

Si consideri, per esempio, il caso semplice in cui si abbia a che fare con insiemi di dati le cui chiavi possano variare in un insieme ben definito, per esempio considerando una chiave di un solo carattere alfabetico, si avrebbe una variazione nell'intervallo: `A..Z`. La realizzazione della funzione *hash*, in questo caso, sarebbe piuttosto immediata: sarebbe sufficiente eseguire una semplice sottrazione tra il carattere considerato e il valore ASCII del carattere `A`: `65`.

Nei casi più frequenti, invece, l'insieme dei *possibili* elementi è di qualche ordine di grandezza superiore alla dimensione del vettore, tale da non rendere possibile il ricorso a un *array* la cui dimensione `m` sia uguale a `u`. Pertanto si rende necessario considerare funzioni *hash* più complesse e non esenti da problemi.

La funzione *hash* inclusa nella classe Java `String`, per esempio, implementa l'algoritmo mostrato nel listato 1. Il valore *hash* è ottenuto sommando il valore ASCII di ciascun carattere per il prodotto del numero primo `31` per il valore *hash* calcolato nell'iterazione precedente:

$$h(x) = \sum_{i=0}^{i=x.length} 31 * h(x)_{(i-1)} + x.char(i)$$

dove $h(x)_{(0)} = 0$

Non a caso il verbo *hash* significa "tagliare del cibo in piccole parti", "triturare" e, per estensione "creare confusione".

Listato 1 – Implementazione della funzione hash da parte della classe String.

```
public int hashCode() {
  // l'attributo hash è utilizzato per effettuare il cash del valore hash
  int h = hash;

  if (h == 0) {
    int off  = offset;
    char val[] = value;
    int len  = count;

    for (int i = 0; i < len; i++) {
      h = 31*h + val[off++];
    }

    hash = h;
  }
  return h;
}
```

Come si può notare, l'implementazione del metodo hashCode utilizza l'attributo hash della classe String per memorizzare il valore precedentemente calcolato. Questa tecnica è molto utile in generale (come si vedrà di seguito le operazioni di inserimento, ricerca ed eliminazione di elementi da strutture dati *hash* possono richiedere di calcolare il valore *hash* degli elementi diverse volte) e in particolare per oggetti immutabili come quelli di tipo stringa. In effetti, una volta che un oggetto di tipo stringa è stato impostato non è possibile variarne il contenuto a meno di non creare un nuovo oggetto.

Collisioni

Situazioni in cui sia possibile dichiarare un *array* la cui dimensione (m) equivalga a quella dell'universo dei *possibili* valori (u) è piuttosto rara. Anche qualora ciò fosse possibile, tipicamente, si preferisce ricorrere ad *array* di dimensioni minori onde evitare di dover allocare rilevanti spazi di memoria destinati a rimanere prevalentemente inutilizzati. Ciò, unito ad altri fattori, rende impossibile individuare, in casi non semplici, funzioni *hash* in grado di generare una corrispondenza biunivoca tra l'insieme dei possibili elementi da memorizzare e l'insieme finito degli indirizzi. Per esempio (cfr. [ARTCP3]), se si considera un insieme di 31 chiavi da memorizzare in una tabella *hash* di 40 elementi, è possibile individuare $41^{31} \approx 10^{50}$ funzioni *hash*, di cui solo $41 \cdot 40 \cdot 39 \cdot 38 \ldots 11/10! \approx 10^{43}$ (una ogni dieci milioni di funzioni) permettono di realizzare una corrispondenza biunivoca. Pertanto, funzioni che permettono di evitare duplicati (anche conoscendo a priori l'insieme delle chiavi) sono piuttosto rare da individuare, anche per grandi dimensioni della tabella di *hash*.

Un esempio molto interessante è dato dal paradosso del compleanno (I.J. Good, *Probability and Weighing of Evidence*, Griffin 1950), il quale afferma che se si raggruppa un insieme di 23 o più persone in una stanza, ci sono ottime possibilità che due o più persone condividano il giorno del compleanno (chiaramente si considera esclusivamente il giorno e il mese). Pertanto, considerando una funzione *hash* che esegue il mapping di 23 elementi in una tavola di 365 elementi (15.87 volte più grande dell'insieme degli elementi) la probabilità che non esistano due elementi con il medesimo valore *hash* è pari a 0.4927 (meno della metà)!

Pertanto, dati due elementi diversi, è possibile che la funzione *hash* `h` generi uno stesso indirizzo:

```
h(x) = h(y), x != y
```

In termini Java:

```
(e.hash == f.hash ) && ( !e.key.equals(f.key) )
```

Questo scenario, denominato conflitto, è facilmente riscontrabile anche dall'analisi della funzione utilizzata nel listato 1: la funzione *hash* implementa un algoritmo randomico e la chiave restituita è un intero (`int`).

Pertanto, le strutture dati *hash table* necessitano di stabilire e implementare opportune politiche per la gestione dei conflitti. Le strategie disponibili sono essenzialmente due e si differenziano per via della struttura dati utilizzata per memorizzare i conflitti. In particolare è possibile ricorrere a:

1. gestioni interne alla tabella *hash* (tecnica utilizzata nelle prime implementazioni della `Hashtable`);

2. gestioni esterne alla tabella *hash* (tecnica utilizzata nelle versioni recenti della classe `Hashtable` e nella classe `HashMap`);

Gestione interna alla tabella

L'idea alla base di questa politica di gestione delle collisioni è molto semplice e consiste nell'utilizzare uno stesso vettore, di dimensioni finite, per memorizzare tutti gli elementi, indipendentemente dal fatto che generino o meno conflitti. Pertanto, qualora l'indirizzo assegnato a un elemento (detto posizione *home*, casa) sia già occupato, ossia si abbia un conflitto, l'algoritmo di *hash* tenta di individuare una posizione libera tra quelle successive alla posizione inizialmente assegnata (*home*). Questa strategia, pertanto, non richiede l'utilizzo di apposite strutture esterne per memorizzare elementi che hanno dato luogo a un conflitto: si ha un solo *array* di dimensioni finite. Un algoritmo molto utilizzato per determinare la posizione successiva disponibile (detto comunemente *linear probing*, "esplorazione lineare") consiste nell'aggiungere un fattore fisso (tipicamente 1) all'indirizzo inizialmente assegnato.

L'algoritmo utilizzato è riportato nel listato 2.

Listato 2 – Pseudocodice dell'inserimento di un elemento in una tabella hash le cui collisioni sono gestite internamente.

1. genera valore hash *h* (posizione home);
2. accedi alla posizione home, *v[h]*;
3. assegna *i = 0*;
4. finché la posizione non è libera (*v[h+i] == null*) e non si è giunti alla fine del vettore (*h+i < m*);
   4.1. verifica che l'elemento non sia uguale a quello di inserire *! v[h+1].equals(x);*
   4.2. se è uguale allora termina
   4.3 aggiungi 1 all'indice *i += 1*
5. se la posizione non è libera (*v[h+1] = null*)
   5.1 assegna *i = -1*;
   5.2 finché la posizione non è libera (*v[h+i] == null*) e non si è giunti all'inizio del vettore (*h-i = 0);*
      5.2.1 verifica che l'elemento non sia uguale a quello di inserire *! v[h+1].equals(x)*
      5.2.2 se è uguale allora termina
      5.2.3 sottrai 1 all'indice i '= 1;
6. se la posizione è libera allora memorizza l'elemento *v[h+i] = x*
7. altrimenti genera eccezione.

Una variante di questo algoritmo consiste nello scorrere il vettore in un solo verso e, qualora tutte le posizioni successive a quella home risultino occupate, procedere con l'ingrandimento del vettore di base (*rehash*).

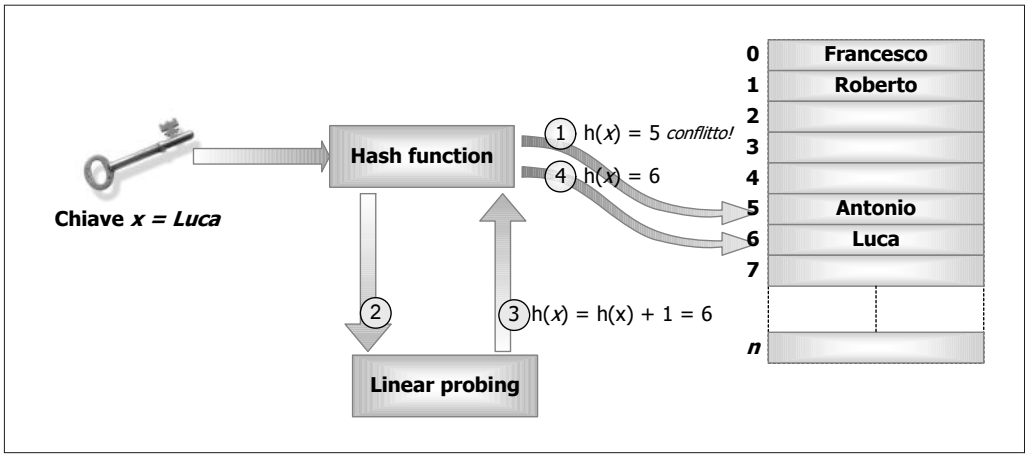

Figura C.2 – Funzionamento della gestione interna di conflitti utilizzando l'esplorazione lineare

Dall'analisi dello pseudocodice del listato 2 è possibile notare che la complessità asintotica dell'algoritmo, nel caso peggiore, non è più O(1), bensì dipende dal maggiore raggruppamento di conflitti. Pertanto, al fine di mantenere questa complessità più prossima possibile al valore desiderato, un'efficace gestione delle tavole *hash* deve prevedere meccanismi in grado di individuare possibili casi di degenerazione e quindi intraprendere opportuni provvedimenti, come per esempio ristrutturare l'intera tabella.

La gestione della struttura di dati *hash* presenta una discreta complessità non solo per l'inserimento degli elementi, ma anche per le operazioni di reperimento e, addirittura, per la cancellazione. In questo caso l'eliminazione di un elemento richiede di traslare tutti gli elementi successivi con lo stesso valore di *hash* della chiave.

Un algoritmo che tenta di introdurre una prima astuzia nell'individuazione della successiva posizione disponibile consiste nell'assegnare un incremento uguale alla radice quadrata del valore *hash*. Questo algoritmo è detto *squared probing* ("esplorazione quadratica").

Questa tecnica chiaramente non risolve il problema di elementi diversi che generano la stessa chiave. Invece tenta di ridurre il problema di conflitti "artificiosi", ossia di conflitti dovuti a elementi che trovano la propria posizione *home* occupata da un elemento finito in quella posizione dopo essersi imbattuto, a sua volta, in una serie di conflitti. Pertanto, sebbene la situazione non migliori per elementi che generano lo stesso indirizzo iniziale (questi generano lo stesso pattern di indirizzi), tenta di risolvere situazioni in cui i raggruppamenti prolificano, cercando di distribuire le aree dei conflitti. Il problema di queste due strategie, infatti, consiste nella generazione di gruppi di elementi intorno ad aree occupate (*cluster*) creando problemi di performance. In altre parole non si ha una distribuzione uniforme degli elementi. Questa situazione ha luogo dopo che un certo numero di inserimenti si siano risolti in una allocazione contigua (priva di elementi disponibili) per via dei conflitti. Con il crescere di queste aree cresce anche la probabilità che la posizione *home* dei nuovi elementi finisca all'interno dei vari raggruppamenti, accrescendo ulteriormente il raggruppamento stesso, e quindi creando nuovi problemi di performance. Inoltre la crescita dei raggruppamenti tende ad aumentare rapidamente non appena raggruppamenti continui comincino a fondersi creando raggruppamenti di dimensioni ancora maggiori.

Una strategia utilizzata per risolvere parzialmente questo problema consiste nell'assegnare al valore *hash* un fattore che non dipende dall'*hash* stesso bensì dalla chiave che lo ha generato. In questo caso si parla di *double hash* (doppio *hash*). La funzione tipicamente utilizzata è:

```
h = ( h + h(x) ) mod m
```

Pertanto il nuovo valore di *hash* è ottenuto dal modulo della somma del valore attuale di *hash* e di quello originario. Da notare che il valore attuale di *hash* non è necessariamente uguale a quello originario, soprattutto dopo la reiterazione di conflitti.

L'ultima variante di gestione delle collisioni interne alla tabella considerata in questa appendice è nota con il nome di *overflow*. Questa consiste nel suddividere l'intera tabella in due aree: una primaria destinata alla normale allocazione degli elementi, e un'altra, detta secondaria o di *overflow*, destinata a memorizzare gli elementi che danno luogo a collisioni. In particolare, quando un elemento genera un conflitto, questo viene memorizzato in un'opportuna

zona dell'area di *overflow* il cui indirizzo è memorizzato nell'elemento che ne occupa la posizione *home*. Se poi questo elemento già puntava a un altro elemento, allora si procede con l'aggiornare la catena, aggiungendo il nuovo arrivato in testa alla coda (si fa riferire l'elemento nella lista primaria al nuovo elemento inserito, al quale viene assegnato il riferimento all'elemento precedentemente riferito dall'elemento nella lista primaria). Questa tecnica è molto simile a quella delle liste concatenate, ampiamente illustrate di seguito, con la variante che le liste sono memorizzate all'interno dell'unica tabella. Pertanto si hanno liste logiche, con riferimenti a posizioni del vettore piuttosto che riferimenti a posizioni in memoria. Quantunque con questa tecnica sia necessario mantenere delle liste concatenate, la dimensione della tabella risulta limitata e l'accesso ai dati dovrebbe risultare più efficiente giacché i conflitti non inficiano l'area principale e si accede sempre alla stessa tabella.

Gestione esterna alla tabella

La gestione esterna, come suggerisce il nome, prevede che la struttura dati utilizzata per memorizzare gli elementi non sia più un semplice vettore, bensì una lista di liste in cui ogni elemento della lista principale contiene il riferimento in memoria ad una lista di elementi accomunati dallo stesso valore *hash* della chiave (queste liste sono tipicamente denominate liste dei conflitti). Pertanto, elementi diversi le cui chiavi danno luogo allo stesso valore *hash* sono memorizzate in una medesima lista di conflitti (figura C.3).

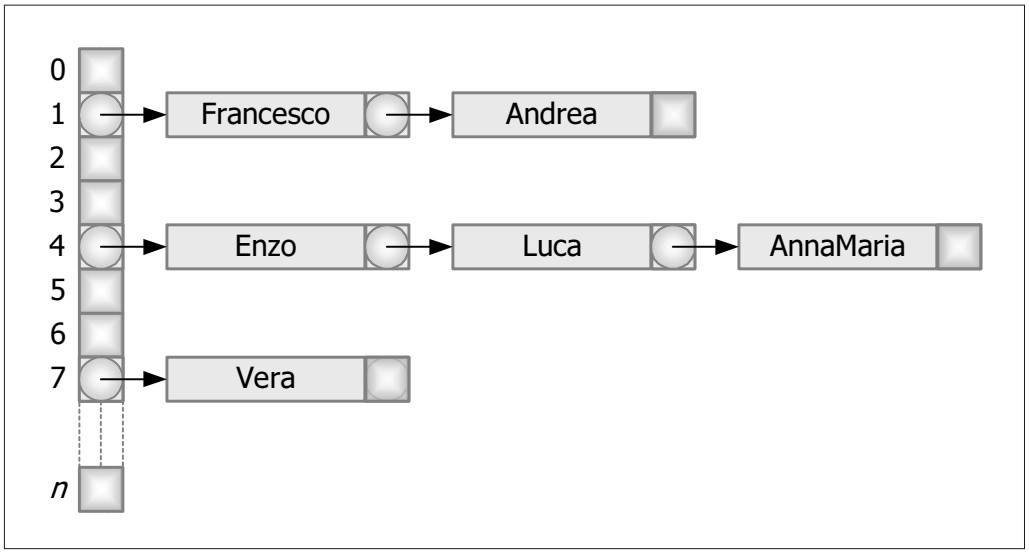

Figura C.3 – Gestione esterna dei conflitti. Il diagramma mostra l'inverosimile situazione in cui h(Francesco) = h(Andrea), h(Enzo) = h(Luca) = h(AnnaMaria).

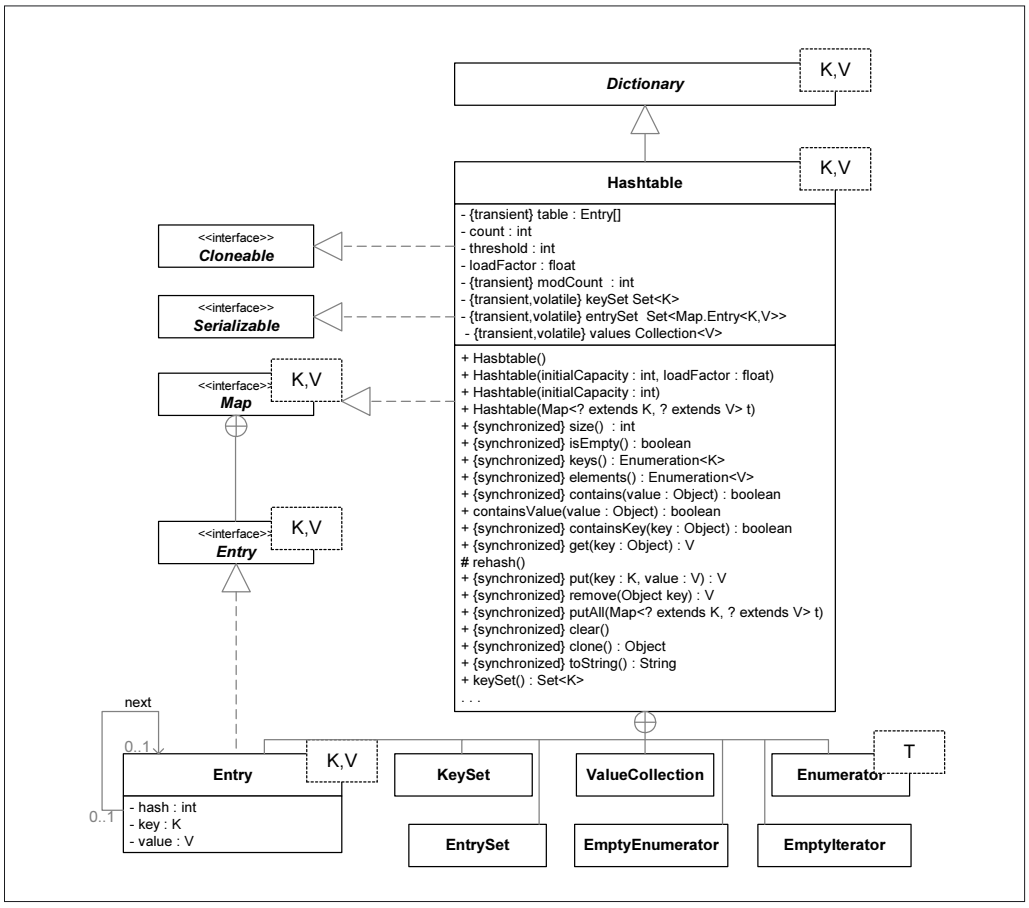

Figura C.4 – Diagramma delle classi della classe java.util.Hashtable.

In questo caso, la complessità asintotica dell'algoritmo è data dalle dimensioni della lista di conflitti di maggiori dimensioni. Nel diagramma precedente, il caso peggiore consiste nella ricerca di un elemento non presente nella tabella la cui chiave generi un valore *hash* uguale a 4.

Per comprendere più chiaramente il funzionamento di questa struttura, si consideri come la classe java.util.Hashtable implementa il metodo di get. Prima di procedere però è necessario fornire alcuni dettagli circa la struttura di questa classe (Figura C.4).

Per chiarimenti circa la notazione UML si rimanda al Capitolo 7 del libro *UML e ingegneria del software*.

Quello che interessa in questo contesto è che la classe java.util.Hashtable dispone di diverse classi annidate (queste sono mostrate nel diagramma attraverso linee spezzate unite alla classe Hashtable per mezzo di una circonferenza con il simbolo + inscritto). Una di queste è la classe

Entry, utilizzata per incapsulare gli elementi inseriti nelle istanze della classe Hashtable (table : Entry[]). In particolare, ogni istanza di questa classe mantiene una copia del valore dell'*hash* dell'elemento, la chiave (key), l'elemento stesso (value) e un riferimento al l'elemento Entry successivo, se presente. Questa relazione di associazione è utilizzata per creare le liste concatenate dette liste dei conflitti.

Per comprendere l'utilizzo degli elementi Entry, si consideri il listato 3 relativo al metodo put che permette di inserire nuovi elementi all'interno della collezione. Come mostrato nella figura C.4, uno degli attributi principali della collezione Hashtable è un *array* di oggetti di tipo Entry (table : Entry[]). Le istanze di questo tipo servono per incapsulare gli elementi da inserire nella lista aggiungendovi, contestualmente, caratteristiche strutturali e comportamentali necessarie per poterli organizzare in liste concatenate necessarie per gestire i conflitti. La classe Entry (listato 4), pertanto, memorizza una copia del valore *hash* al fine di migliorare le performance, la chiave, l'elemento e l'indirizzo dell'elemento successivo:

Listato 3 – Metodo put *della classe* java.util.Hashtable.

```
1 public synchronized V put(K key, V value) {
2   // Make sure the value is not null
3   if (value == null) {
4     throw new NullPointerException();
5   }

6   // Makes sure the key is not already in the hashtable.
7   Entry tab[] = table;
8   int hash = key.hashCode();
9   int index = (hash & 0x7FFFFFFF) % tab.length;
10    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
11      if ( (e.hash == hash) && e.key.equals(key) ) {
12        V old   = e.value;
13        e.value  = value;
14        return old;
15      }
16    }

17    modCount++;
18    if (count >= threshold) {
19      // Rehash the table if the threshold is exceeded
20      rehash();

21      tab = table;
22      index = (hash & 0x7FFFFFFF) % tab.length;
23    }
```

```
24    // Creates the new entry.
25    Entry<K,V> e = tab[index];
26    tab[index] = new Entry<K,V>(hash, key, value, e);
27    count++;
28    return null;
29  }
```

Si proceda con l'esaminare il listato 3. Le istruzioni 3, 4, 5 servono per verificare che non si stia tentando di inserire un elemento nullo. Da notare che invece ciò è possibile con la classe java.util.HashMap. Le istruzioni 8 e 9 servono per generare il valore *hash* della chiave e per normalizzarlo in funzione alla dimensione del vettore delle liste di collisione. In particolare la variabile index contiene l'indirizzo della lista dei conflitti dove il nuovo elemento dovrà essere memorizzato. Il ciclo for (istruzione 10) è utilizzato per scorrere l'intera lista dei conflitti al fine di individuare un elemento con la stessa chiave. Se ciò accade (istruzione 11) allora l'elemento individuato viene restituito al metodo chiamante subito dopo che al suo posto è stato inserito il nuovo elemento. Qualora non esista un duplicato, si incrementa il contatore di variazioni strutturali (modCount, istruzione 17). Se questo valore supera il fattore di soglia (threshold = capacità * fattore di caricamento), allora il metodo procede con il riorganizzo della tabella interna (istruzione 20). In particolare, il metodo di rehash incrementa la dimensione della tabella interna e quindi riorganizza i vari elementi presenti. Ciò determina anche la necessità di normalizzare nuovamente il valore di *hash* della chiave del nuovo elemento (istruzione 22). Il compito delle istruzioni dalla 25 poi è di memorizzare il nuovo elemento nella prima posizione della lista. A tal fine, viene memorizzato il riferimento al primo elemento della lista di collisioni. Quindi al suo posto è inserito il nuovo elemento, subito dopo averlo immesso in un nuovo oggetto di tipo Entry, il cui riferimento all'elemento successivo contiene proprio l'indirizzo dell'elemento che precedentemente occupava la prima posizione.

Si consideri ora l'implementazione del metodo get riportato nel listato 4. In particolare, dopo aver ottenuto il valore *hash* dell'elemento da cercare, il metodo esegue una sua normalizzazione e quindi accede alla relativa lista di conflitti. Questa viene scorsa fino a quando o si trova l'elemento cercato (in questo caso si utilizza la funzione equals: e.key.equals(key)), oppure si giunge alla fine della lista.

Listato 4 – Implementazione del metodo get *della classe* java.util.Hashtable.

```
public synchronized V get(Object key) {
  Entry tab[] = table;
  int hash = key.hashCode();
  int index = (hash & 0x7FFFFFFF) % tab.length;

  for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
    if ( (e.hash == hash) && e.key.equals(key) ) {
      return e.value;
    }
```

```
    }
    return null;
  }
```

Listato 5 – Attributi della classe **Entry**.

```
  private static class Entry<K,V> implements Map.Entry<K,V> {
    int hash;
    K key;
    V value;
    Entry<K,V> next;
    . . .
```

I listati mostrati evidenziano come questa gestione dei conflitti permette di avere una tabella degli elementi di dimensioni prefissate e liste di conflitti in grado di crescere, teoricamente, all'infinito.

Funzioni hash e numeri primi

Nell'analizzare la realizzazione di una serie di funzioni *hash* è possibile notare la presenza di numeri quali: …, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, … e soprattutto 31 e 37 utilizzati come fattori moltiplicativi. In effetti, si tratta di numeri primi. Quantunque non ci sia una prova matematica certa in grado di dimostrare l'efficacia dei numeri primi nella generazione (calcolo) di numeri psudorandomici, studi empirici mostrano che il loro utilizzo permette di generare migliori distribuzioni di numeri randomici.

Pertanto utilizzare numeri primi per la realizzazione delle funzioni *hash* permette di raggiungere migliori risultati. I numeri primi selezionati però non devono essere eccessivamente grandi, altrimenti si corre il rischio che moltiplicando i vari numeri si finisca con l'azzerare i bit meno significativi del valore *hash* calcolato. Questi bit giocano un ruolo molto importanti per via dell'operazione di modulo necessaria per normalizzare il valore intero calcolato alla dimensione dell'*hash table*.

Alcuni esempi di algoritmi di hash

Saranno adesso presentati alcuni algoritmi *hash* particolarmente interessanti. L'idea non è quella di fornire una panoramica esaustiva degli algoritmi di *hash* esistenti, ma di presentarne alcuni particolarmente interessanti utilizzati per le strutture dati. Questi, oltre a fornire un riscontro pratico alla teoria presentata sin qui, dovrebbero offrire interessanti spunti per ulteriori approfondimenti demandati però al lettore.

Per semplicità di trattazione, si è deciso di considerare esclusivamente funzioni *hash* che operano sul dominio delle stringhe (hash(string)). Come si vedrà al termine di questa appendice, la loro estensione agli altri tipi di dati è piuttosto immediata.

Caratteristiche di una buona funzione hash

Prima di procedere con la disamina dell'insieme di funzioni *hash* selezionato, è importante ricordare brevemente quali siano le caratteristiche di una buona funzione *hash*:

Capacità di evitare collisioni

Sebbene questa possa sembrare una caratteristica più teorica che pratica, e quindi non facilmente raggiungibile a meno di specifiche informazioni circa il dominio delle chiavi, non è infrequente il caso in cui sia possibile evidenziare, da una semplice ispezione, funzioni disegnate non accortamente e quindi destinate a generare un gran numero di conflitti.

Capacità di generare dei valori uniformemente distribuiti nel sottoinsieme dei numeri naturali considerato (dimensione dell'array)

Indicando con p*i* la probabilità che h(x) = *i*, una funzione *hash* che distribuisce uniformemente i valori ha la proprietà che 0 < *i* < M, p*i* = 1/M (i valori *hash* sono uniformemente distribuiti). Purtroppo, per poter eseguire delle valutazioni sulla distribuzione dei valori di *hash*, è necessario conoscere alcune informazioni circa la distribuzione delle chiavi. In assenza di queste informazioni, si può assumere che le chiavi siano equiprobabili. Indicando con K*i* l'insieme delle chiavi che generano un medesimo valore *i* : h(Ki) = *i*, il requisito di distribuire uniformemente i valori calcolati, implica che |K*i*| = |K|/|M|, ossia che un ugual numero di chiavi si riferisca a ciascun elemento dell'array.

Efficienza di calcolo (ridotta complessità)

Si tratta di un'ulteriore, importante caratteristica di una funzione *hash*.

Passiamo ora ad analizzare diversi esempi di algoritmi di hash.

Additive Hash (hash additivo)

Uno dei primissimi algoritmi di *hash* elaborato è il cosiddetto *hash* additivo. Il nome è dovuto alla sua idea di base che prevede semplicemente di addizionare il valore ASCII dei caratteri che costituiscono la stringa. Del risultato ottenuto viene eseguito il modulo per la dimensione del vettore, per la quale, come spesso avviene, è consigliabile selezionare un numero primo. Valori che invece devono essere evitati sono potenze di due. In questo caso l'operazione di modulo restituisce gli ultimi bit del numero calcolato ignorando completamente i restanti. Per evitare questo fenomeno, in generale è necessario evitare valori tali che siano divisibili per un valore del tipo $r^k \pm a$, dove a e k sono due numeri, è la radice dell'insieme dei caratteri. Pertanto la soluzione più semplice consiste nel selezionare un numero primo.

Listato 6 – Additive hash.

```
public int additiveHash(String source, int size) {

  if ( (source == null) || (source.length() == 0) ) {
    throw new NullPointException();
  }
  long hash = source.length();
  for (int i=0; i < source.length(); i++) {
    hash += source.charAt(i);
  }

  return (int) (hash % size);
}
```

Quantunque questo algoritmo sia molto semplice e di efficiente esecuzione, soffre del problema di generare valori molto simili e quindi, in ultima analisi, di dar luogo a un numero significativo di conflitti.

Rotative Hash (hash rotativo)

Questa funzione permette di generare valori di *hash* eseguendo operazione al livello di bit sui caratteri che costituiscono la stringa. In particolare, si imposta il valore iniziale di *hash* uguale alla lunghezza della stringa, quindi, per ogni carattere, si eseguono le seguenti operazioni: si sposta a sinistra di 4 posizioni il corrente valore *hash* (in sostanza lo si moltiplica per un fattore pari a 16), quindi lo si mette in xor con il valore *hash* questa volta spostato a sinistra di 28 posizioni, il tutto viene posto nuovamente in xor con il valore ASCII del carattere corrente.

Listato 7 – Rotative hash.

```
public int rotativeHash(String source, int size) {

  if ( (source == null) || (source.length() == 0) ) {
    throw new NullPointException();
  }
  long hash = source.length();
  for (int i=0; i < source.length(); i++) {

    hash = (hash << 4 ) ^ (hash >> 28) ^ source.charAt(i);
  }

  return (int) (Math.abs(hash) % size);
}
```

Questo algoritmo permette di individuare buoni valori di *hash*, senza aggravare il costo computazionale che rimane decisamente contenuto: si eseguono esclusivamente operazioni binarie.

Bernstein's Hash

L'algoritmo di Bernstein è molto semplice e, al contempo, permette di ottenere interessanti risultati. In questo caso il valore di *hash*, inizialmente impostato uguale a zero, viene sommato al valore ASCII di ogni carattere componente della stringa, dopo essere stato moltiplicato per il numero 65.

Listato 8 – Bernstein's Hash.

```
public int bernsteinHash(String source, int size) {

  if ( (source == null) || (source.length() == 0) ) {
    throw new NullPointException();
  }
  long hash = 0;
  for (int i=0; i < source.length(); i++) {
    hash += hash * 65 + source.charAt(i);
  }

  return (int) (Math.abs(hash) % size);
}
```

Anche in questo caso si hanno risultati molto buoni con ottime performance.

CBU hash

Il nome di questo algoritmo è dovuto alla Università di Canterbury, di Christchurch, Nuova Zelanda (**C**anter**B**ury **U**niversity) dove è stato ideato. Anche in questo caso il valore *hash* è azzerato inizialmente, e poi, per ogni carattere delle stringa, è spostato di 2 posizioni a sinistra e addizionato al valore ASCII del corrente carattere. Questo algoritmo, oltre ad essere molto efficiente, ha dimostrato di funzionare bene soprattutto con stringhe di caratteri. Queste caratteristiche lo rendono molto interessante, tanto che sue variazioni sono utilizzate in altri algoritmi come il Revision Control System.

Listato 9 – CBU Hash.

```
public int cbuHash(String source, int size) {
```

```
    if ( (source == null) || (source.length() == 0) ) {
      throw new NullPointException();
    }
    long hash = 0;
    for (int i=0; i < source.length(); i++) {

      hash = (hash << 2) + source.charAt(i);
    }

    return (int) (Math.abs(hash) % size);
  }
```

PKP hash

Questo algoritmo deve il proprio nome alle iniziali del suo inventore: Peter K. Pearson il quale pubblicò nel 1990 un algoritmo in grado di generare numeri pseudorandomici nell'intervallo da 0 a 255 (byte). L'idea base era molto semplice e consisteva in un primo passo di inizializzazione necessario per generare un vettore con i primi 255 numeri inseriti in ordine casuale. Dopodiché, il codice ASCII di ciascun carattere della stringa, dopo essere stato posto in xor con il corrente valore hash, veniva usato come indice del vettore al fine di prelevare il numero corrispondente.

L'algoritmo è strutturato come mostrato nel listato 10.

Listato 10 – PKP Hash originale.

```
  public int pkpHash(String source, int size) {

    if ( (source == null) || (source.length() == 0) ) {
      throw new NullPointException();
    }

    pTab[] = {3, 12, 176, 34, … // 256 elementi

    long hash = 0;
    for (int i=0; i < source.length(); i++) {

      hash = pTab[ hash ^ source.charAt(i)];
    }

    return hash;
  }
```

Questo algoritmo funzionante al livello di byte, per essere esteso al condominio dei long richiede di considerare ben otto contributi (uno per ciascun byte che forma il long). Pertanto, si può immaginare di eseguire in parallelo 8 PKP hashing, ognuno relativo ad uno dei byte del tipo long. La versione estesa di questo algoritmo (listato 11) necessita pertanto di

- generare 8 pTabs di valori casuali, sempre compresi tra 1 e 256;
- consideare 8 *hash* parziali;
- utilizzare il codice ASCII di ciascun carattere per aggiornare gli 8 *hash* di cui sopra;
- effettuare il merge degli 8 *hash* parziali in un unico *hash* finale.

Listato 11 – PKP Hash esteso.

```
public class PKPHashFunction {
  // this table is used to generate the 8 per 256 pseudo-random numbers
  private  int pTabs[][]  = null;
  private static int numBytes  = 8;

  /**
   * Default constructor
  */
  public PKPHashFunction() {
    initPTabs();
  }

  /**
   * Initialises the table filled with random number
   * Each row contains the number from 1 to 256 stored in a random order
  */
  private void initPTabs() {
    pTabs = new int[numBytes][255];
    for (int ind = 0; ind < pTabs.length; ind++) {
      pTabs[ind] = generateRandomByteArray(256);
    }
  }

  /**
   * Generates a pseudo-random array filled with number
   * from 1 to size, stored in a random order
```

```
 * @param size vector dimension
 * @return int array of size dimension with number form 1 to size randomly stored
*/
private int[] generateRandomByteArray(int size) {
  int[] randomArray = new int[size];
  // initialise the array
  for (int ind = 0; ind < size; ind++) {
    randomArray[ind] = ind;
  }
  // randomise the array
  Random randomGenerator = new Random();
  for (int ind = 0; ind < size; ind++) {
    // select the position to swap with the current one
    int pos = randomGenerator.nextInt(256);
    // swap bytes
    int currByte  = randomArray[ind];
    randomArray[ind] = randomArray[pos];
    randomArray[pos] = currByte;
  }
  return randomArray;
}

/**
 * Generates the PKP Hash value
 * @param source String to use for the hash generation
 * @param size   address space size
 * @return  the PKP hash value
*/
public int pkpHash(String source, int size) {
  if ( (source == null) || (source.length() == 0) ) {
    throw new NullPointException();
  }
  int hash[] = new int[numBytes];
  for (int i=0; i < source.length(); i++) {
    for (int j=0; j < numBytes; j++) {
      hash[j] = pTabs[j][ hash[j] ^ source.charAt(i)];
    }
  }
  // accumulate the different contributions
  int resultHash = 0;
  for (int j=0; j < numBytes; j++) {
    resultHash += ( hash[j] << (8*j) );
  }
```

```
        return (int) (Math.abs(resultHash) % size);
    }
}
```

Come si può notare, questo algoritmo presenta qualche problema di complessità. In particolare è necessario allocare 8 tabelle di 256 caratteri e quindi eseguire ogni passo 8 volte. Comunque, i risultati ottenuti sono del tutto apprezzabili.

Un test informale

Gli algoritmi presentati precedentemente sono stati sottoposti a una serie di verifiche. In particolare, si sono considerati tre insiemi di input e, per dimensioni crescenti della tabella di *hash*, si è misurato il numero dei conflitti generati. Sebbene questo valore da solo non sia molto significativo (sarebbe necessario analizzarlo congiuntamente con altri dati quali dimensioni dei cluster, distribuzione dei conflitti, etc., ma ci vorrebbe un testo solo per questo), è comunque utile per fornire un iniziale apprezzamento circa la bontà delle varie funzioni. Poiché una qualità importante richiesta agli algoritmi hash consiste nel distribuire uniformemente le chiavi di ingresso sulla relativa tabella, sono riportati due grafici con le distribuzioni generate da due ben definiti insiemi di input su due tabelle di dimensioni ben definite.

I tre vettori di input utilizzati sono, rispettivamente, un insieme di nomi di personaggi celebri, i codici ISIN di alcuni strumenti finanziari dell'India e, per terzo, i codici di svariate compagnie aree.

Di seguito sono riportati i valori degli array utilizzati.

Vettore con nominativi celebri (228 elementi)

```
private static String elementNames[] = {
"Curtis Jackson", "Vincent Furnier", "Alizée Jocotet", "Anastacia Newkirk", "Sasmi Anggun Cipta", "Jimmy
McShane", "Barry Pinkus", "Hansen Beck", "William Michael Albert Broad", "Björk Gudmundsdottir", "Robert
Zimmermann", "Paul Hewson", "George Alan O'Dowd", "Steven Demetre Georgiou", "Charles Aznavourian",
"Cherilyn Sarkasian La Pierre", "Francisco Repilando", "Olga De Souza", "Deadrie Crozie", "David Robert Hayward
Jones", "Dino Paul Crocetti", "Dido Armstrong", "Raymond Donnez", "Adrian Donna Gaines", "Andre Young",
"Edith Giovanna Gassion", "Reginald Kenneth Dwight", "Declan Patrick McManus", "Marshall Bruce Mathers III",
"Eithne Ni Bhraonain", "Derek William Dick", "Michael Balzary", "Mammola Sandon", "Frank Sinatra", "Farookh
Pluto Bulsara", "Luis Romero", "Georgios Kyriacos Panayiotou", "Geraldine Estelle Halliwell", "Josè Angel Hevia
Velasco", "O'Shea Jackson", "James Newell Osterberg", "John Robert Cocker", "John Francis Bongiovi", "Jean
Philippe Smet", "Henry Wayne Casey", "Keith Richards", "Robert James Ritchie", "Christopher Hamill", "Richard
Wayne Penniman", "Steve Van Zandt", "Lewis Firbank Reed", "Louise Veronica Ciccone", "Oscar Tramor", "Maria
Sofia Kalogeropoulos", "Brian Warner", "Melanie Brown", "Melanie Jayne Chrisholm", "Michael Bolotin", "Miguel
Dominguin", "Richard Melville Hall", "Achinoam Nini", "Noelia Lorenzo Monge", "Winston Hubert McIntosh",
"Alicia Moore", "Roger Jouret", "Prince Roger Nelson", "Sean Combs", "Richard Bresch", "Enrique Martin
Morales", "George Albert Tabett", "Richard Starkey", "Helen Folasade Adu", "Terence Trent D'Arby", "Sandra Ann
Goodrich", "Sandra Lauer Cretu", "Orville Richard Burrell", "Shakira Ripoli Mebarak", "Norman David Shapiro",
```

"Deborah Ann Dyer", "Steven Tallarico", "Steveland Hardaway Judkins", "Gordon Matthew Sumner", "David Evans", "Anna Mae Bullock", "Thomas Beecher", "Thomas Jones Woodward", "Antonio De La Cuesta", "Tracy Louise Freeman", "George Ivan Morrison", "Wesley Johnson", "Ivo Livi", "Salvatore Adamo", "Albano Carrisi", "Aleandro Civai", "Alessia Aquilani", "Carla Bissi", "Roberta Mogliotti", "Andrea Fumagalli", "Anna Hoxha", "Antonio Muraccioli", "Nicola Balestri", "Roberto Gatti", "Domenico Di Graci", "Antonio Calò", "Mariano Rapetti", "Carla Quadraccia", "Davide Civaschi", "Carlo Marchino", "Alfredo Rapetti", "Cristiano Rossi", "Claudia Moroni", "Claudio Pica", "Yolande Christina Gigliotti", "Stefano Zanchi", "Eugenio Zambelli", "Vito Luca Perrini", "Donato Battaglia", "Aldo Caponi", "Giuliano Illiani", "Giampiero Anelli", "Stefano Belisari", "Elisa Toffoli", "Vincenzo Jannacci", "Tommaso Zanello", "Nicola Fasani", "Paolo Panigada", "Marina Fiordaliso", "Francesco Di Gesù", "Roberto Antoni", "Alfredo Buongusto", "Gabriele Ponte", "Renato Abate", "Luigi Giovanni Maria Panceri", "Paul Mazzolini", "Giampiero Scalamogna", "Federico Renzulli", "Dante Luca", "Giorgia Todrani", "Giorgio Gaberschick", "Giovanna Nocetti", "Giuseppina Romeo", "Federico Monti Arduini", "Carmela Iannetti", "Ivan Lenin Graziani", "Alessandro Aleotti", "Gianfranco Randone", "Giovanna Bersola", "Enrico Sbricolli", "Giovanna Coletti", "Edoardo Bennato", "Giorgio Guidi", "Lorenzo Cherubini", "Massimo Jovine", "Marco Messina", "Lorena Biolcati", "Annalisa Panetta", "Antonio Ciacci", "Paul Bradley Couling", "Giuseppe Mango", "Marco Armenise", "Marina Restuccia", "Mario Macciocco", "Giovanni Calone", "Maurizio Lobina", "Maria Di Donna", "Marcello Modugno", "Domenica Bertè", "Gian Michele Maisano", "Daniela Miglietta", "Christian Meyer", "Maria Ilva Biolcati", "Anna Maria Mazzini", "Beniamino Reitano", "Mirella Fregni", "Giulio Rapetti", "Umberto Giardini", "Monica Bragato", "Marco Castoldi", "Nada Malanima", "Natale Codognotto", "Giovanni Pellino", "Angela Cacciola", "Filippo Neviani", "Nico Tirone", "Domenico Colarossi", "Michele Scommegna", "Adionilla Pizzi", "Agostino Ferrari", "Ombretta Comelli", Orietta Galimberti", "Pierpaolo Pelandi", "Paola Iezzi", "Chiara Iezzi", "Nicoletta Strambelli", "Giuseppe Faiella", "Barbara D'Alessandro", "Enzo Ghinazzi", "Raffaele Riefoli", "Bruno Canzian", "Renato Brioschi", "Renato Ranucci", "Renato Fiacchini", "Riccardo Sanna", "Roberto Concina", "Roberto Loreti", "Roberto Castiglione", "Camillo Facchinetti", "Sergio Conforti", "Rosalino Cellamare", "Fabio Staccotti", "Giovanni Scialpi", "Gaetano Scardicchio", "Ivana Spagna", "Stefano Rossi", "Susanna Quattrocchi", "Cecilia Cipressi", "Giovanni Giacobetti", "Ferruccio Ricordi", "Antonio Lardera", "Elio Cesari", "Tiziana Donati", "Salvatore Cutugno", "Guido Lamberti", "Monica Zucchi", "Virginia Minetti", "Giuditta Guizzetti", "Adelmo Fornaciari", "Luca Persico" };

Vettore dei codici (ISIN) di alcuni prodotti finanziari (178 elementi)

private static String elementISINs[] = {
"INA05A000079", "INA12A000120", "INA15A000150", "INB10A002614", "INE253B01015", "INA03A730016", "INA14A830045", "INA12A840053", "INA14A840101", "INA15A850075", "INE761C01015", "INE762C01013", "INE673C01012", "INE130C07010", "INE130C07044", "INE130C07051", "INE130C01013", "INE226B01011", "INE208A14014", "INE208A14022","INE208A14030", "INE208A07018", "INE208A07026", "INE208A07034", "INE208A07042", "INE208A01011", "INE293C01019", "INE046C01011", "INE441A01018","INE363A01014", "INE237D01014", "INE787D14029", "INE294A14022", "INE294A14030", "INE294A14048", "INE294A14055", "INE294A07018", "INE828A01016", "INE972C01018", "INE436C01014", "INE160C01010", "INE279C01018", "INE824B01013", "INE604B01019", "INE069C01013", "INE986A01012", "INE605B01016", "INE819C01011", "INE702C01019", "INE829B01012", "INE751C01016", "INE151D01017", "INE084D01010", "INE419D01018", "INE085A01013", "INE194C01019", "INE713D01014", "INE368D01017", "INE974C01014", "INE442C01012", "INE953B01010", "INE777B01013", "INE088A01017", "INE488A01019", "INE178A01016", "INE217C01018", "INE132B01011", "INE974B01016", "INE489A01017", "INE937D14012", "INE937D14020", "INE820A14016", "INE820A14024", "INE820A01013", "IN9820A01011", "INE086A01011",

```
"INE095B01010", "INE319D01010", "INE285A01019", "INE671B01018", "INE652C01016", "INE541A01015",
"INE353B01013", "INE311B01011", "INE106B01015", "INE426D01013", "INE038A08017", "INE353A01015",
"INE074C01013", "INE310C01011", "INE549A14011", "INE549A14029", "INE549A01018", "INE551A01014",
"INE005A11887", "INE005A11895", "INE005A11903", "INE005A11911", "INE005A14048", "INE005A14055",
"INE005A14063", "INE005A14071", "INE005A14089", "INE005A14097", "INE005A08BW8", "INE005A11853",
"INE005A08BN7", "INE849D14019", "INE444C01018", "INE008A08101", "INE008A08127", "INE069A07030",
"INE069A07048", "INE069A07055", "INE786B01022", "INE569A01024", "IN9569A01030", "IN9569A01048",
"INE966C01010", "INE058D01014", "INE575A01013", "INE916B01017", "INE307C01017", "INE028C01019",
"INE802B01019", "INE635B04017", "INE770B01018", "INE166B01019", "INE980A01015", "INE946A01016",
"INE206D08014", "INE206D08022", "INE699B01019", "INE843A01015", "INE787B01012", "INE844A01013",
"INE855B01017", "INE165C01019", "INE269D01017", "INE700B01015", "INE179A01014", "INE002A08062",
"INE002A08104", "INE002A08112", "INE002A08120", "INE002A08138", "INE266C01015", "INE640C01011",
"INE455D01012", "INE075B01012", "INE001C01016", "INE298B01010", "INE710B01014", "INE461D01010",
"INE311D01017", "INE295B01016", "INE722B01019", "INE017A07369", "INE017A07377", "INE017A07385",
"INE188D01019", "INE017B01014", "INE822C01015", "IN9965A01014", "IN9965A01022", "IN9965A01030",
"IN9965A01048", "IN9965A01055", "INE216B01012", "INE551D01018", "INE650C01010", "INE429B01011",
"INE718A01019", "INF725B01060", "INF725B01136", "INF725B01144", "INF725B01102", "INF725B01110" };
```

Vettore con un insieme di codici di compagnie aeree (198 elementi)

```
private static String elementsAirlineCodes[] = {
 "JP", "AEA", "RE", "EI", "VV", "SU", "AR", "4L", "EUK", "BT", "AB", "AC", "CA", "AF", "AI", "JM", "UL", "KM",
"MK", "SW", "NZ", "A7", "AIRS", "GRE", "HM", "WOW", "TS", "JY", "PB", "6G", "UM", "AZ", "NH", "4J", "AA",
"OZ", "5W", "AEU", "RC", "GR", "AO", "OS", "J2", "BD", "LZ", "PG", "B2", "BG", "WW", "BA", "BNW", "BW",
"8B", "ZQ", "CX", "KX", "C0", "MU", "C9", "CF", "WX", "X9", "CO", "SS", "OU", "OK", "CT2", "CU", "CY", "DWT",
"DL", "T3", "U2", "EZ", "DS", "MS", "LY", "JEM", "EK", "EOS", "OV", "ET", "EY", "VE", "EUR", "3W", "EA", "BR",
"JN", "AY", "DP", "BE", "GC", "GA", "4U", "B4", "JN2", "GF", "ZU", "T4", "2L", "8H", "X3", "IB", "TY", "FI", "IC",
"IR", "JL", "9W", "LS", "JFK", "KQ", "YK", "KL", "KE", "KU", "LI", "LN", "TE", "LO", "LH", "LG", "DM", "W5",
"MH", "MA", "MY", "IG", "ME", "ZB2", "ZB", "MYT", "VZ", "CE", "HG", "NW", "DY", "OL", "OA", "WY", "OHY",
"PK", "PMA", "9R", "NI", "QF", "QR", "ROC", "AT", "BI", "RJ", "FR", "S4", "SV", "SK", "SKY", "CB", "MI", "SQ",
"5P", "RU", "NE", "JZ", "ALXX", "QS", "SN", "SA", "JK", "SD", "WV", "LX", "RB", "DT", "TP", "RO", "TG", "MT",
"TOM", "BY", "HV", "5A", "TU", "TK", "T5", "T7", "UA", "US", "HY", "RG", "VIK", "VS", "VG", "WF", "WM", "W6",
"IY", "Z4" };
```

L'analisi del numero di conflitti, come detto sopra, considerato da solo, non è un fattore molto significativo. In effetti, le varie funzioni *hash* sembrerebbero fornire le stesse prestazioni in termini del numero di conflitti. Tuttavia, è possibile trarre alcune considerazioni iniziali. In primo luogo, è possibile notare che con tabelle di *hash* di dimensioni circa uguali a quelle dell'insieme dei vettori di input si ottiene circa il 35-40% dei conflitti. Mentre per scendere ad un numero di conflitti oscillanti tra il 20-25% è necessario ricorrere a una tabella *hash* di dimensioni doppie rispetto al numero dei valori di input. La presenza di conflitti, però ancora non permette di trarre delle conclusioni circa le prestazioni. Per poter valutare questa caratteristica è necessario considerare la distribuzione delle chiavi sulla tabella *hash*. A tal fine si è deciso di

Vettore nomi (228 elementi)						
Size	Rotating	Bernstein	Default	SBU	PKP	Media
97	139	143	141	137	136	139.2
109	135	133	137	127	133	133
127	122	116	117	124	116	119
149	107	107	115	113	111	110.6
173	98	101	99	99	103	100
223	83	79	93	75	88	83.6
239	82	83	77	87	73	80.4
263	69	74	78	74	73	73.6
317	65	63	58	67	63	63.2
503	52	41	50	41	48	46.4
719	33	35	34	26	31	31.8
1019	16	29	23	33	19	24
1187	11	21	18	28	22	20
Media	77.84615	78.84615	80	79.30769	78.15385	

Tabella C.1 – Numero di conflitti generati con il vettore dei nomi per dimensioni crescenti della tabella di hash.

vettore ISIN (178 elementi)						
Size	Rotating	Bernstein	Default	SBU	PKP	Media
97	91	96	94	96	100	95.4
109	88	91	93	89	92	90.6
127	84	80	80	79	86	81.8
149	71	80	73	79	79	76.4
173	62	64	63	67	77	66.6
223	53	63	55	49	51	54.2
239	52	58	55	52	63	56
263	53	51	52	50	55	52.2
317	45	36	38	43	45	41.4
503	30	29	25	31	29	28.8
719	17	17	20	24	26	20.8
1019	8	11	13	10	12	10.8
1187	8	19	11	10	16	12.8
Media	50.92308	53.46154	51.69231	52.23077	56.23077	

Tabella C.2 – Numero di conflitti risultanti con il vettore dei codici dei prodotti finanziari per dimensioni crescenti della tabella di hash.

vettore codici vettori aerei (198 elementi)						
Size	Rotating	Bernstein	Default	SBU	PKP	Media
97	113	116	113	114	115	114.2
109	109	96	103	100	111	103.8
127	95	92	96	97	94	94.8
149	90	86	82	89	85	86.4
173	70	74	69	81	79	74.6
223	64	63	67	81	70	69
239	58	55	51	77	62	60.6
263	54	74	51	79	62	64
317	45	39	45	75	45	49.8
503	25	34	22	71	35	37.4
719	30	21	15	72	31	33.8
1019	22	7	10	72	18	25.8
1187	26	18	7	77	20	29.6
Media	61.61538	59.61538	56.23077	83.46154	63.61538	

Tabella C.3 – Conflitti risultanti con il vettore dei codici delle linee aeree per dimensioni crescenti della tabella di hash

- considerare i primi 40 elementi dell'array dei nomi e studiarne la distribuzione su una tabella di 43 elementi (tabella C.4 e figura C.5);

- considerare i primi 40 elementi dell'array dei codici dei vettori aerei e di studiarne la distribuzione su una tabella di 89 elementi (Figura C.5).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Rotating	1	0	2	3	1	1	0	1	0	0	0	1	0	0	0	1	0	2	1	1	1	1	0	1	0	0	2	1	3	0	0	4	2	0	2	2	0	2	1	1	0	1	1
Bernstein	1	0	0	2	2	0	0	0	0	1	0	1	0	0	2	2	2	0	0	0	2	4	1	0	1	0	0	1	1	0	2	0	2	2	1	0	2	0	0	2	1	3	2
Default	4	1	0	1	3	2	0	0	0	2	0	0	2	2	0	1	0	0	0	0	1	0	1	0	0	0	2	3	2	1	1	1	0	2	1	0	0	1	0	2	0	1	3
SBU	1	1	3	0	1	0	0	1	0	2	2	1	1	1	0	1	1	1	1	5	2	0	0	0	3	0	2	0	3	1	0	1	1	0	1	1	0	1	1	0	0	0	0
PKP	1	1	2	0	1	1	1	0	2	0	3	0	2	2	0	0	2	0	0	1	0	1	0	0	1	2	1	0	1	2	1	1	1	1	1	1	3	0	2	0	1	0	1

Tabella C.4 – Distribuzione dei primi 4 elementi dell'array dei nomi in una tabella di 43 posizioni.

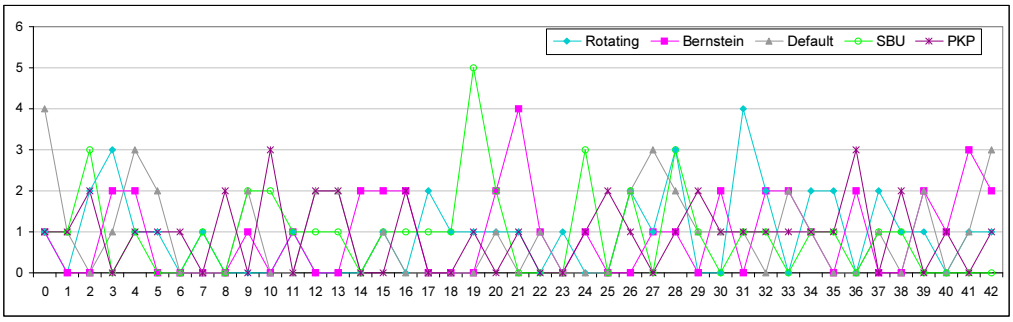

Figura C.5 – Grafico delle distribuzioni dei primi 40 elementi dell'array dei nomi in una tabella di 43 posizioni.

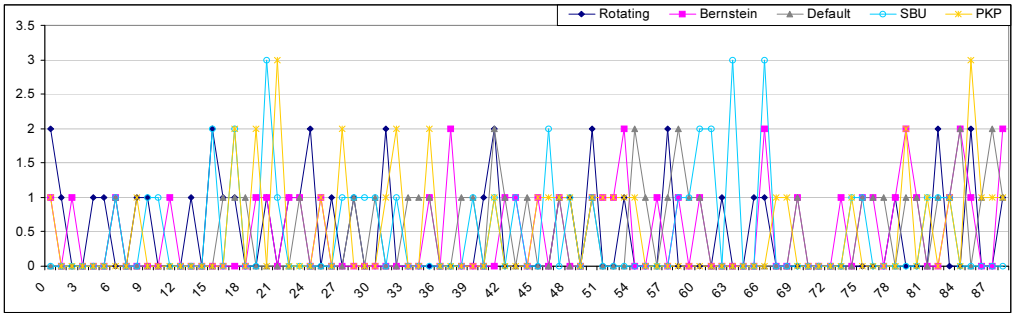

Figura C.6 – Grafico delle distribuzioni dei primi 40 elementi dell'array dei codici dei vettori aerei memorizzati in una tabella di 89 posizioni.

Dall'analisi delle distribuzioni di tabella C.4 è possibile notare che, nel caso in cui la dimensione della tabella di *hash* sia circa uguale a quella dell'insieme delle chiavi di input, la ricerca di un elemento tende a presentare una complessità media veramente del tipo O(1). Il caso peggiore è dato dall'algoritmo CBU che alloca 5 elementi nella posizione 19. Pertanto, utilizzando tale funzione, nel caso in cui si cerchi un elemento non presente nella tabella il cui valore *hash* della chiave sia 19 la complessità è O(5). Sempre dall'analisi della tabella C.4, è possibile notare che le varie funzioni, per quanto concerne la distribuzione degli elementi, hanno comportamenti molto differenti. Tra questi si evidenzia negativamente la distribuzione generata dall'algoritmo CBU. Mentre, sempre da questo test, sembrerebbe che la funzione *hash* in grado di generare un comportamento maggiormente uniforme sia la PKP.

In figura C.6 è mostrata la distribuzione generata da 40 elementi distribuiti su una tabella *hash* di dimensioni poco superiori al doppio di tali elementi. In questo caso è possibile notare che la complessità asintotica è più marcatamente di tipo O(1). Questo test, inoltre, mostra un pessimo comportamento delle funzioni *hash* CBU (3 posizioni con 3 elementi e 5 con 2) e PKP (2 posizioni con 3 elementi e 6 con 2) e un ottimo comportamento della funzione *hash* standard e di quella di Bernstein.

Metodo generico per generare valori hash delle proprie classi

Dopo aver discusso della teoria delle tabelle *hash*, nel listato 12 presentiamo una classe, HashCodeBuilder, molto utile per l'implementazione del metodo hashCode ereditato da tutte le classi Java a partire dalla classe antenata java.lang.Object. Da notare che un'altra versione della stessa classe (org.apache.commons.lang.builder.HashCodeBuilder) è fornita da Apache Jakarta (http://jakarta.apache.org/commons/lang/).

La classe presentata si presta ad essere utilizzata in due modalità diverse

- utilizzo dei soli metodi statici per la generazione del valore hash di elementi di tipo specifico: byte, int, long, float, array, object, etc. Per esempio hash += HashCodeBuilder.hash(name); hash += HashCodeBuilder.hash(name);

- utilizzo canonico: si crea un'istanza e quindi vi si aggiungono gli attributi da considerare per la generazione del valore *hash*. Alla fine si ottiene il valore *hash* desiderato invocando il classico metodo hashCode.

Listato 12 – Metodo generico per generare valori hash delle proprie classi.

```
import java.io.Serializable;
import java.lang.reflect.Array;


/**
 * This class manages the operations necessary to generate hashCode values
```

```
 * Furthermore it provides other objects with a number of static
 * methods that can be directly invoked to generate the hashcode for some
 * specific data types.
 * Therefore, it can be used creating an instance and then appending values,
 * or invoking directly the static methods <code>hash</code>
 * Example use case:
 * <pre><code>
 *   HashCodeBuilder hcb = new HashCodeBuilder();
 *   hcb.append(field1).append(filed2);
 *   myHash = hcb.hashCode()
 * </code>
 * or
 * <code>
 *   myHash += HashCodeBuilder.hash(field1);
 *   myHash += HashCodeBuilder.hash(field2);
 * </code></pre>
 * @author L.V.T.
 * @since JDK1.3
 */
public class HashCodeBuilder implements Serializable {

  // ———————————— CONSTANTS ————————————

  // Universal version identifier. Deserialization method uses this number to ensure
  // that a loaded class corresponds exactly to a serialized object.
  static final long serialVersionUID = 7526472295622776147L;

  /** Hashcode value to use for null */
  private static final int HASH_NULL = 0;

  /** Hashcode for a 'false' boolean */
  private static final int HASH_PRIME_NUMBER = 37;

  // ———————————— ATTRIBUTES ————————————

  /** This is the value of the hashcode */
  private int hashCodeValue;

  // ———————————— METHODS ————————————

  /**
   * Construct method
   *
   * @param value int to use to initialise the hash code value
   */
```

```
public HashCodeBuilder(int value) {
  hashCodeValue = value;
}


/**
 * Default constructor.
 * It sets the internal hash code value to the prime number
 */
public HashCodeBuilder() {
  this(HASH_PRIME_NUMBER);
}


/**
 * Add a boolean contribution to the hashCode value.
 * @param b  Boolean value to consider
 * @return this object
 */
public HashCodeBuilder append(boolean b) {

  this.hashCodeValue ^= hash(b);

  return this;
}


/**
 * Add a byte contribution to the hashCode value.
 * @param by Byte value to consider
 * @return this object
 */
public HashCodeBuilder append(byte by) {

  this.hashCodeValue ^= hash(by);

  return this;
}


/**
 * Add a char contribution to the hashCode value.
 * @param c Char to add to the hashCode
```

```
     * @return this object
     */
    public HashCodeBuilder append(char c) {

        this.hashCodeValue ^= hash(c);

        return this;
    }


    /**
     * Add a short contribution to the hashCode value.
     * @param sh Short to add to the hashCode
     * @return this object
     */
    public HashCodeBuilder append(short sh) {

        this.hashCodeValue ^= hash(sh);

        return this;
    }


    /**
     * Add the given integer contribution to the hashCode value.
     * @param i Integer value to add to the hashCode
     * @return this object
     */
    public HashCodeBuilder append(int i) {

        this.hashCodeValue ^= hash(i);

        return this;
    }


    /**
     * Add the given long contribution to the hashCode value.
     * @param l Long value to add to the hashCode
     * @return this object
     */
    public HashCodeBuilder append(long l) {

        this.hashCodeValue ^= hash(l);
```

```
    return this;
}


/**
 * Add the given float contribution to the hashCode value.
 * @param f float value to add to the hashCode
 * @return this object
 */
public HashCodeBuilder append(float f) {

    this.hashCodeValue ^= hash(f);

    return this;
}


/**
 * Add the given double contribution to the hashCode value.
 * @param d Double value to add to the hashCode
 * @return this object
 */
public HashCodeBuilder append(double d) {

    this.hashCodeValue ^= hash(d);

    return this;
}


/**
 * Add the given object hashCode contribution to this object hashCode value.
 * @param o Object whose hashCode has to be added
 * @return this object
 */
public HashCodeBuilder append(Object o) {

    this.hashCodeValue ^= hash(o);

    return this;
}
```

```
/**
 * Return the current hash code value
 * @return Current hash code value.
 */
public int hashCode() {

   return this.hashCodeValue;
}


/**
 * Verify whether or not this object is equal to the given one
 * @param o Object to test equality with.
 * @return <code>true</code> if object is equal; <code>false</code>otherwise
 */
public boolean equals(Object o) {
   boolean ret = false;

   if ( (o != null) && (o.getClass() == getClass()) ) {

      HashCodeBuilder obj = (HashCodeBuilder) o;
      ret = (this.hashCodeValue == obj.hashCodeValue);
   }

   return ret;
}


/**
 * Return a string representation of the state of this object
 * @return A string representation of the state of this object
 */
public String toString() {
   return this.hashCodeValue+"";
}


/**
 * Return a cloned copy of this object
 * @return This object clone
 */
public Object clone() {
   return new HashCodeBuilder(this.hashCodeValue);
```

```
}


/**
 * Generate a hash code for a boolean
 * @param b Boolean value to hash
 * @return hash code generated.
 */
public static int hash(boolean b) {
   return ( b ? HASH_PRIME_NUMBER : (HASH_PRIME_NUMBER*3) );
}


/**
 * Generate a hash code for an integer
 * @param i Integer value to hash
 * @return hash code generated.
 */
public static int hash(int i) {
   return i + HASH_PRIME_NUMBER;
}


/**
 * Generate a hash code for a char
 * @param c Char value to hash
 * @return hash code generated.
 */
public static int hash(char c) {
   return c + HASH_PRIME_NUMBER;
}


/**
 * Generate a hash code for a long value.
 * @param l  Long value to hash
 * @return hash code generated.
 */
public static int hash(long l) {
   return (int) (HASH_PRIME_NUMBER + (l ^ (l >> 32)) );
}
```

```
/**
 * Generate a hash code for a double value.
 * @param d  Double value to hash
 * @return hash code generated.
 */
public static int hash(double d) {
   return hash(Double.doubleToLongBits(d));
}


/**
 * Generate a hash code for a float value.
 * @param f  Float value to hash
 * @return hash code generated.
 */
public static int hash(float f) {
   return Float.floatToIntBits(f);
}


/**
 * Generate a hash code for the given object.
 * Java considers array as objects
 * @param o  Object to use for the hash code
 * @return hash code generated.
 */
public static int hash(Object o) {
   int hash = HASH_PRIME_NUMBER;

   if (o == null) {  // is it null?
      hash = HASH_NULL;

   } else if ( ! o.getClass().isArray() ) { // is it a simple object ?

      hash += hash(o.hashCode());

   } else {         // is it an array?

      int length = Array.getLength(o);

      for ( int i = 0; i < length; i++ ) {

         Object currObj = Array.get(o, i);
```

```
            // hash is given by the current obj hash: recursive invocation
            hash = hash(currObj);
          }
        }

        return hash;
      }

} // — End of class —
```